



Osaamista  
ja oivallusta  
tulevaisuuden  
tekemiseen

Juho Hölsä

# Sovellusarkkitehtuuristandardien käyttö nykyaikaisten sovellusten suunnittelussa

Metropolia Ammattikorkeakoulu

Medianomi

Viestinnän tutkinto-ohjelma

Opinnäytetyö

15.11.2018

Tekijä(t) Otsikko	Juho Hölsä Sovellusarkkitehtuuristandardien käyttö nykyaikaisten sovel- lusten suunnittelussa
Sivumäärä Aika	45 sivua 15.11.2018
Tutkinto	Medianomi
Tutkinto-ohjelma	Viestinnän tutkinto-ohjelma
Suuntautumisvaihtoehto	Digitaalinen viestintä
Ohjaaja(t)	Lehtori Juhana Kokkonen
<p>Tämän opinnäytetyön aihe on sovellusarkkitehtuuristandardien käyttö nykyaikaisten appli- kaatioiden suunnittelussa. Sovellusarkkitehtuuri on rakennettu ja dokumentoitu malli sovel- luksen rakenteesta ja toiminnasta, jolla pyritään edesauttamaan suunnittelijoiden ja ohjel- moijien työtä sovelluksen koko elinkaaren aikana. Työssä käytetyt standardit ovat UML (uni- fied modeling language) ja MVC (model-view-controller). Kirjoittaja on asettanut kaksi hypo- teesia ohjaamaan tutkimusta: (1) sovellusarkkitehtuurimetodeista MVC-metodia käytetään useammin kuin muita arkkitehtuurimetodeja ja (2) vaikka UML-mallinnuskieli on alan stan- dardi, sen käyttö on epäolennaista nykyaikaisten sovellusten suunnittelussa. Hypoteesit pohjataan jo julkaistuun tutkimukseen sekä arkkitehtuurimetodien suosioon ohjelmistoalalla. Opinnäytetyön aihe syntyi kirjoittajan omista kokemuksista ja havainnoista työelämässä.</p> <p>Työ koostuu sovelluksia ja sovellusarkkitehtuuria tarkastelevasta teoriaosuudesta sekä tut- kimusosuudesta, jossa teemahaastattelun avulla haastateltiin ohjelmistokehittäjiä arkkiteh- tuurien käytöstä. Opinnäytetyön tutkimuskysymys on: millä tavoin sovellusarkkitehtuuristan- dardeja käytetään nykyaikaisten sovellusten suunnittelussa? Teemahaastattelu jaettiin kol- meen osaan. Ensimmäisessä osassa haastateltavaa pyydettiin piirtämään sovellusarkkiteh- tuuri kuvanjakopalvelulle. Toisessa osassa kysyttiin UML-mallinnuskielen käytöstä ja kol- mannessa osassa MVC-metodin käytöstä. Haastatteluissa ilmeni, että standardien käyttö on luovaa ja sovellettua.</p> <p>Haastattelujen perusteella voidaan tulkita, että vaikka MVC-arkkitehtuuria ei nähdä kestä- vänä metodina, käytetään sitä silti soveltavasti nykyaikaisten sovellusten suunnittelussa. Li- säksi haastatteluissa ilmeni, että vaikka UML-mallinnuskieli on alan standardi, sitä käytetään löyhästi osana arkkitehtuurin rakentamista.</p> <p>Opinnäytetyössä todetaan, että alusta asti mietitty, suunniteltu ja toteutettu arkkitehtuuri on hyödyksi suunnittelijoille ja ohjelmoijille sovelluskehityksen kaikissa vaiheissa. Lisäksi opin- näytetyössä ehdotetaan sovelluskehityksen vaiheiden tehostamista antamalla suunnitteli- joille vapauksia käyttää omaa luovuuttaan osana arkkitehtuurin rakentamista. Tämän työn pohjalta saatua tietoa sovellusarkkitehtuureista ja standardien luovasta käytöstä voi hyödyn- tää tulevien sovellusten suunnitteluprosessin ja sovellettujen arkkitehtuurimetodien kehittä- misessä, esimerkiksi avaamalla keskusteluja metodien luovasta ja sovelletusta käytöstä or- ganisaation sisällä.</p>	
Avainsanat	Sovellusarkkitehtuuri, sovelluskehitys, MVC, UML

Author(s) Title	Juho Hölsä Software Architecture Standards in Modern Application Development
Number of Pages Date	45 pages 15 Nov 2018
Degree	Bachelor of Culture and Arts
Degree Programme	Media
Specialisation option	Digital media
Instructor(s)	Juhana Kokkonen, Senior Lecturer
<p>The aim of this Bachelor's thesis is to find out how software architecture standards are used in the modern application development. In this study, software architecture means the steps behind the application development process. The standards used in the study are as follows: UML (Unified Modeling Language) and MVC (Model-View-Controller) architecture. The author has stated two hypothesis to guide the research: (1) MVC method is more effective than other software architecture methods and (2) even though the UML language is the standard it is nonessential to use like it was designed. The hypothesis are based on earlier studies about software architectures standards, usage statistics and the author's personal experience. The subject for the study was inspired by the author's personal experience in the field of software development.</p> <p>The study consists of a theoretical and research section. The theoretical section describes what software architecture is and why it is important. The research was conducted by using a qualitative research method and by interviewing developers. The research question of the thesis examined the methods of software architecture standards used in the development process. The semi-structured interviews had a framework of themes to be discussed: (1) designing architecture for an application, (2) using MVC as a part of the architecture and (3) using UML as a part of the architecture.</p> <p>Based on the interviews, it can be interpreted that MVC is still used widely but the usage is mostly adaptive. The same can be suggested for UML: language use is not based on the original documentation but rather lean utilization of the standard.</p> <p>It was concluded that architecture that is researched, designed and implemented from the start is more useful for both designers and programmers in each step of the development process, and that you could come to a better outcome by using architecture methods adaptively. The knowledge about adaptive usage of software architectures acquired from this thesis can be utilized in e.g. remodeling the application development process of the organization.</p>	
Keywords	Software Architecture, Software Development, MVC, UML

## Sisällys

1	Johdanto	1
2	MVC ja UML osana sovellusarkkitehtuuria	2
2.1	Ohjelmistot ja sovelluskehitys	3
2.2	Mitä sovellusarkkitehtuuri on?	4
2.3	MVC ja sovellusarkkitehtuuri	5
2.3.1	Model	6
2.3.2	View	7
2.3.3	Controller	7
2.4	UML-mallinnuskieli	8
2.4.1	Luokkakaavio	9
2.4.2	Luokkien suhteet ja polut	11
3	Metodit	13
3.1	Hypoteesit	13
3.2	Toteutus ja rakenne	14
3.3	Aineiston analysointi	16
4	Aineisto	17
4.1	Yksi mahdollinen tapa esittää kuvanjakopalvelu	18
4.2	Vilin haastattelu	20
4.3	Jaakon haastattelu	23
4.4	Sepen haastattelu	27
4.5	Juhan haastattelu	31
5	Tutkimustulokset ja analysointi	34
5.1	MVC-arkkitehtuurin osien rooli nähdään epäselvänä	34
5.2	UML-mallinnuskieltä käytetään soveltavasti	35
5.3	MVC-arkkitehtuuria ei nähdä kestäväenä metodina	37
5.4	Kehittäjän rooli organisaatiossa vaikuttaa MVC-arkkitehtuurin käyttöön	38
6	Johtopäätökset	39
7	Lopuksi	41
	Lähteet	43

## 1 Johdanto

Olen valinnut opinnäytetyöni aiheeksi sovellusarkkitehtuuristandardien käytön nykyaikaisten sovellusten suunnittelussa. Sovellusarkkitehtuuri (engl. *software architecture*) on malli sovelluksen rakenteesta ja toiminnasta. Sillä pyritään edesauttamaan suunnittelijoiden ja ohjelmoijien työtä ohjelmiston koko elinkaaren aikana. Sovelluskehitykseen kuuluu olennaisesti applikaatioiden rakenteen kirjaaminen ja dokumentointi. Se miten applikaation toimintalogiikan saa ymmärrettävästi ja tehokkaasti esitettyä on erittäin kiehtova aihe. Haluan tutkia opinnäytetyössäni sovellusarkkitehtuurien käyttöä ja sitä, kuinka laajasti siihen liittyviä standardeja käytetään. Kun organisaatiot rakentavat kestäviä arkkitehtuureja sovelluksille, pitkäaikainen tuotto maksimoituu (Booch 2006).

Toimiessani freelancerina ja ohjelmistokehittäjänä ketterät (engl. *agile*) metodit ovat nousseet esille applikaatioiden sovelluskehitysvaiheessa. Esitän työssäni kaksi hypoteesia sovellusarkkitehtuuriin liittyen, jotka pohjaan julkaistuun tutkimukseen ja selkeisiin perusteltuihin trendeihin ohjelmistotalalla (ks. luku 3). Ensimmäinen hypoteesini on, että sovellusarkkitehtuurimetodeista MVC (*model-view-controller*) on nykyään kaikkein tehokkain ja sitä käytetään useammin kuin muita arkkitehtuurimetodeja. Toinen hypoteesini on, että vaikka UML (*unified modeling language*) on alan standardi, sen käyttö on epäolennaista nykyaikaisten sovellusten suunnittelussa. Luvussa kolme perustelen hypoteesien käyttöä tarkemmin. Hypoteeseja varten tutkimuskysymykseni on: millä tavalla sovellusarkkitehtuuristandardeja käytetään nykyaikaisten sovellusten suunnittelussa? Opinnäytetyön teoriaosuudessa esitän MVC:n toimintalogiikan sekä UML:n version 2.0 käytön perusteet.

Rajaan tutkimuksen koskemaan pelkästään sovellusarkkitehtuuristandardien käyttöä osana nykyaikaisten sovellusten suunnittelua. Standardeiksi tutkimusta varten olen valinnut UML-mallinnuskielen ja MVC-arkkitehtuurin. UML-mallinnuskieli on alan standardi (OMG 2005), ja MVC-arkkitehtuuri on käytetyin arkkitehtuurimetodi (Google Trends n.d.). Tässä opinnäytetyössä en tutki muita sovellusarkkitehtuureja. Luvussa kaksi käyn kuitenkin sovelluskehityksen vaiheet lyhyesti läpi.

Opinnäytetyöni tavoitteena on tutustuttaa lukija sovellusarkkitehtuurin vaiheisiin sekä ottaa selvää, pitävätkö esittämäni hypoteesit sovellusarkkitehtuuristandardeista paikkansa. Tavoitteena ei ole muuttaa nykyisiä arkkitehtuurimetodeja. Sen sijaan tämän työn

pohjalta saatua tietoa nykyaikaisista arkkitehtuureista ja suunnittelumetodeista voi hyödyntää tulevien sovellusten suunnitteluprosessin ja sovellettujen arkkitehtuurimetodien kehittämisessä.

Luvussa kaksi määritellään lyhyesti, mikä sovellus on ja lisäksi esitellään sovelluskehityksen vaiheet. Olen liittänyt lukuun teoriaosuuden MVC-arkkitehtuurista sekä UML:n käytöstä. Työn analyysiosassa peilaan haastatteluja teoriaosuudessa esitettyihin tiivistelmiin arkkitehtuuristandardeista. Kolmannessa luvussa esittelen haastattelussa käytetyt metodit sekä niistä kootun aineiston. Luvut neljä ja viisi on omistettu haastattelujen, luonnosten sekä tutkimusaineiston analysointiin ja läpikäyntiin. Luvussa kuusi tehdään johtopäätöksiä pohjaten ne edellisten lukujen analyysiin. Lisäksi luvussa tehdään johtopäätöksiä sovellusarkkitehtuurin kehityksen suunnasta nykyaikaisessa sovelluskehityksessä. Luvussa seitsemän pohdin työn lopputulosta ja sitä, miten tutkimusta voisi jatkaa tulevaisuudessa.

Osana opinnäytetyötä olen haastatellut neljää ohjelmistokehityksen ammattilaista. Suurin haaste tutkimustyössä oli mahdollisuus haastatteluiden rakenteen hajoamiselle, jolloin saatu aineisto jäisi häilyvälle pohjalle. Osana ensimmäistä haastattelua testasin tutkimuskysymyksen toimivuutta, jotta haastatteluiden rakenteesta tulisi kestävä (ks. luku 3). Mielestäni haastattelun tärkein ja kiinnostavin osa oli kuvanjakopalvelun sovellusarkkitehtuurin rakentaminen ja suunnittelu. Applikaatioksi valitsin tunnetun sosiaalisen median kuvapalvelun Instagramin. Haastattelun aikana applikaation sovellusarkkitehtuurista piirrettiin yksinkertainen versio paperille.

## 2 MVC ja UML osana sovellusarkkitehtuuria

Ohjelmistoiksi kutsutaan niitä ohjelmia, joita operoidaan tietokoneella. Ohjelmistot ovat osa nykyaikaista länsimaista yhteiskuntaamme väestötietojärjestelmistä terveydenhuoltoon, koulutuksesta työelämään ja jokapäiväisen elämämme tarpeista vapaa-aikaan. Ohjelmisto ei ole vain ohjelma itsessään tai päätelaite, jolla ohjelmisto ajetaan. Siihen kuuluu lisäksi dokumentaatio ohjelmiston käytöstä ja hallinnasta sekä kirjattu konfiguraatio siitä, miten ohjelmisto käytetään ja toimii. (Sommerville 2010, 1-10.)

Ohjelmistokehitykseksi (engl. *software development*) kutsutaan koko sitä vaihetta, jonka aikana sovellusta ideoidaan, määritellään, suunnitellaan, ohjelmoidaan, dokumentoi-

daan ja testataan (O'Regan 2014, 42-43). Sovelluskehitys on synonyymi ohjelmistokehitykselle. Ohjelmistokehitykseen ei ole olemassa yhtä ainoata standardia. Siihen käytetään monia malleja, toimintatapoja, arkkitehtuureja ja kieliä. Liiallinen vapaus on pienten sovellusten suunnittelutyössä mahdollisuus, mutta se voi johtaa ohjelmistokehityksessä monimutkaisuuteen etenkin laajempien sovellusten kohdalla. Monimutkaisuus on organisaatioille kallista. (Babar & Abrahamsson 2008, 242-243.)

## 2.1 Ohjelmistot ja sovelluskehitys

Sovelluksen suunnitteluun käytetään ohjelmistokehityksen keinoja. Tarkempia termejä sovelluskehityksen käytöstä ovat esimerkiksi web-, Android- ja iOS-kehitys. Myös PWA-suunnittelu (*progressive web application*) kuuluu sovelluskehityksen piiriin. Kaikkeen sovelluskehitykseen tarvitaan ohjelmistokehitystä, mutta esimerkiksi graafisen pelin pelikehitys ei ole sovelluskehitystä. (Sommerville 2010, 1-10.)

Sovelluskehitys on prosessi, jossa suunnittelu- ja hallintaprosessin tehostamiseksi kehitys jaotellaan osiin. Prosessia kutsutaan myös nimellä SDLC (*software development life cycle*), ja se pitää sisällään niin ensimmäiset projektin suunnittelun vaiheet kuin itse sovelluksen kehitystyönkin (Ruparelia 2010, 8). SDLC helpottaa struktuurin ja arkkitehtuurin määrittelyä. SDLC:n vaiheet ja niiden tärkeysjärjestys vaihtelee kehitystiimin päätösten mukaan. Maciaszekin (2007, 26-30) mukaan SDLC:n useimmissa projekteissa käytetyt avainvaiheet ovat seuraavat: *Analysis, System Design, Implementation, Integration and Deployment ja Operation and Maintenance*.

Ketterä sovelluskehitys (engl. *agile software development*) on joukko sovelluksen suunnittelussa käytettäviä menetelmiä. Käytettyjä metodologioita ovat esimerkiksi:

- extreme programming
- scrum
- crystal family of methodologies
- feature driven development
- the rational unified process
- dynamic systems development method
- adaptive software development
- open source development (Abrahamsson, Salo, Ronkainen & Warsta 2002, 19-20.)

## 2.2 Mitä sovellusarkkitehtuuri on?

Sovellusarkkitehtuuri on kuvaus sovelluksen rakenteesta ja toimintatavoista. Se voidaan jakaa korkean tason rakenteisiin, joilla on oma tehtävänsä ja suhteensa toisiin rakenteisiin. Eriteltynä rakenneosia ovat esimerkiksi tietokanta ja sovelluslogiikka. Sovelluslogiikan voi jakaa alikomponentteihin eli itsenäisesti toimiviin osiin. Alikomponenteilla on yhteisesti sovittuja kommunikointiväyliä eli rajapintoja, joilla ne keskustelevat keskenään. Komponentit ovat se osa sovelluksesta, joka suunnitellaan arkkitehtuurin avulla. (Bass, Clements & Kazman 2003, 43-46.)

Richardsin (2015, 1, 11, 21, 27) mukaan sovellusten suunnitteluun käytettävät arkkitehtuurimetodit voidaan jakaa viiteen kategoriaan:

- Layered architectures
- Event-driven architectures
- Microkernel architectures
- Microservices architectures
- Space-based architectures

Sovellusarkkitehtuuri on rakennearkkitehtuurista kumpuava metafora (Perry & Wolf 1992, 41). Se pitää sisällään suunnitelman ja tehtävät, joita suunnittelijat ja ohjelmistokehittäjät käyttävät. Hyvin suunniteltu ja dokumentoitu sovellusarkkitehtuuri antaa mahdollisuuden aikaiseen kommunikointiin ja suunnitteluun ennen sovelluksen julkaisua (Bass ym. 2003, 37). Siitä on siis hyötyä läpi sovelluksen elinkaaren. Hyvin tehty arkkitehtuuri maksaa itsensä takaisin jo sovelluksen aikaisessa vaiheessa.

Sommerville (2010, 10) jakaa arkkitehtuurin kahteen eri tasoon: järjestelmäarkkitehtuuriin (*architecture in large*) ja sovellusarkkitehtuuriin (*architecture in small*). Puhekielessä voisi karkeasti samalla tavalla jakaa sovellusarkkitehtuurin olevan front-end-kehitystä, järjestelmäarkkitehtuurin back-end-kehitystä. Huomioitavaa on, että sovellusarkkitehtuuri sekoitetaan usein järjestelmäarkkitehtuuriin, mutta niillä on itsenäiset roolit sovelluksen arkkitehtuurissa.

Arkkitehtuuria käytetään eniten niissä organisaatioissa, joissa hallinnoidaan isoja sovellus- ja järjestelmäkokonaisuuksia. Sovellukset, joiden hallinta ja kehittäminen on jaettu



useamman eri osaston tai ryhmän välillä, tarvitsevat järjestelmä- ja sovellusarkkitehtuuria toiminnan hahmottamiseen kehityksen eri vaiheiden aikana. (O'Regan 2014, 45.)

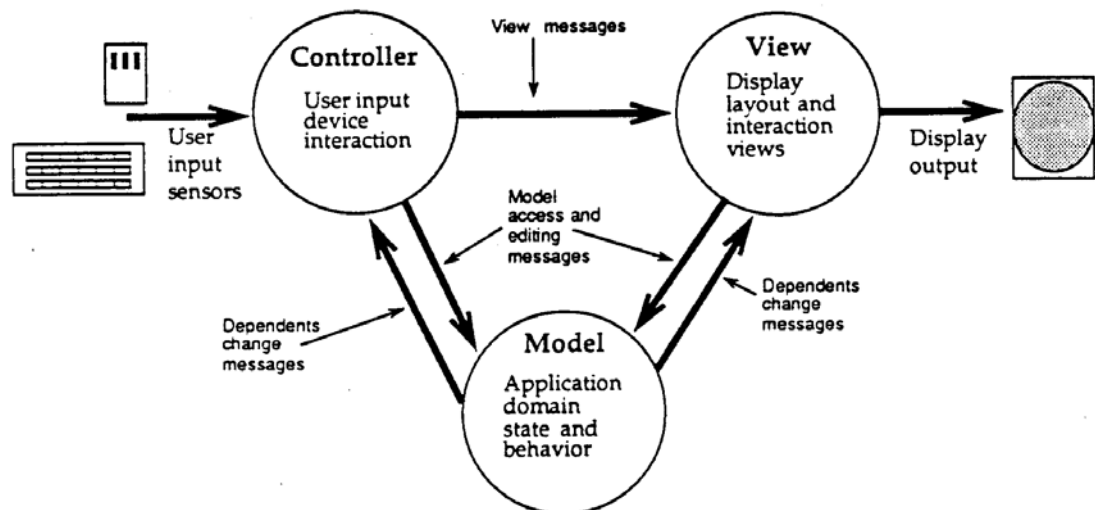
### 2.3 MVC ja sovellusarkkitehtuuri

MVC on tunnettu sovellusarkkitehtuurimetodi, ja se kuuluu kerrosarkkitehtuureihin (engl. *layered architecture*). Vuonna 1979 Trygve Reenskaug loi MVC:n selkeäksi tavaksi ratkoa yleisiä ongelmia antamalla käyttäjille mahdollisuuden hallita informaatiota monesta perspektiivistä. Reenskaug kutsui MVC:tä yleiseksi suunnittelusysteemiksi (*planningsystem*). Sitä ei siis alun perin luotu arkkitehtuurimetodiksi. (Reenskaug 2007, 1-2.)

Kosketuspinta sovellusarkkitehtuuriin kuitenkin löytyi muutamia vuosia myöhemmin, kun Dortmundin yliopistolla Glenn E. Krasner ja Stephen T. Pope (1988) julkaisivat MVC:n käyttöohjeet Smalltalk-80-ohjelmointikielen graafiselle käyttöliittymälle. Krasnerin ja Popen (1988) artikkelia pidetään yleisesti MVC-sovellusarkkitehtuurin pohjatyönä.

MVC-sovellusarkkitehtuurilla on kaksi tavoitetta: (1) luoda moduuleja määrittämään sovelluksen toimintaa ja sitä kautta helpottamaan eri interaktioiden ymmärtämistä sovelluskehityksessä ja (2) luoda toistettavia komponentteja sovelluskehittäjien työn helpottamiseksi. (Krasner & Pope 1988, 26.)

Sovellusarkkitehtuurissa komponenttien modulaarisuudesta on hyötyä. Kun funktiot erotetaan toisistaan, suunnittelijan on helpompi ymmärtää applikaation toimintaa sekä rakentaa uusia toiminnallisuuksia tarvitsematta hallita muiden funktioiden sisäistä toimintaa yksityiskohtaisesti. Tätä kutsutaan modulaarisuudeksi. (Krasner & Pope 1988, 48.)



Kuvio 1. Alkuperäinen kolmivaiheinen Model-View-Controller (Krasner & Pope 1988, 27)

Kuviossa 1 on Krasnerin ja Popen piirtämä diagrammi MVC:n kiertokulusta. Kiertokululla tarkoitetaan sovelluksen sisällä informaation liikkumista osien välillä. Esimerkiksi kuviossa 1 model-osasta lähtee kaksisuuntaiset viivat sekä view-osaan että controller-osaan. Vasemmalla kuviossa on esitetty sovitimet, joita controller-osa kuuntelee. Oikealla view-osasta lähtee käyttäjälle näkyvä informaatio.

Määritän seuraavissa alaluvuissa MVC:n osat Krasnerin ja Popen julkaisemaan artikkelin (1988) mukaan.

### 2.3.1 Model

Model-osa simuloi sovelluksen toimintaa. Se on siis eräänlainen toimintamalli ja kirjoitettu logiikka sovelluksen ytimessä. Alkuperäisen Smalltalk-80:n kanssa esitetyn mallin mukaan Model on se osa, josta MVC:n rakentaminen aina aloitetaan (Krasner & Pope 1988, 26-28).

Model-osa vastaa aina informaation käsittelystä ja sisällöstä. Se saa controller-osalta viestejä, joka taas vastaavasti seuraa käyttäjän interaktioita. Model-osan yksi tehtävistä on se, että sovelluksen sisäinen muutos johtaa kaikkien view-osien päivittämiseen yhden osan sijasta (Krasner & Pope 1988, 26). Tällä haetaan MVC-sovellusarkkitehtuurin toisen tavoitteen täyttymistä.

Model-osa sisältää sovelluksen keskeiset toiminnot, eli sen mitä se tekee (Krasner & Pope 1988, 28). Yksinkertaisimmillaan model-osa voi siis olla vaikkapa lukujono (engl. *integer*). Tällöin sovellus voisi olla yksinkertainen laskuri.

### 2.3.2 View

View-osa on se osa sovelluksesta, joka näkyy päätelaitteella. View-osan tehtävä on pyytää informaatiota model-osalta ja näyttää sen käyttäjälle (Krasner & Pope 1988, 29).

Sovelluksen view-osa ei pidä sisällään vain niitä sovelluksen osia, jotka sillä hetkellä näkyvät vaan myös niin sanottuja subviews-osia, jotka ovat osa isompaa superview-osastoa (Krasner & Pope 1988, 27-29). Superview voi olla esimerkiksi kerätty kokoelma DOM-elementtejä web-applikaatiossa. DOM-elementti (engl. *document object model - element*) on tapa jolla selain lukee ja käsittelee informaatiota (Robie 1998). View-osaa voidaan siis kutsua sovellusarkkitehtuurin graafiseksi osaksi, joka sisältää useampia hierarkkisesti rakentuneita, sisäkkäisiä view-osia.

View-osan tehtävä ei ole lukea ja välittää käyttäjän interaktioita sovelluksen sisällä toisin kuin joskus esitetään. Interaktioiden lukeminen on yksinomaan controller-osan tehtävä (Krasner & Pope 1988, 27).

### 2.3.3 Controller

Sovelluksen controller-osa vastaa ja välittää käyttäjän suoria interaktioita sovelluksen sisällä. Se on suorassa yhteydessä model-osan ja view-osan välillä. (Krasner & Pope 1988, 29).

Asynkroniset kutsut kuuluvat controller-osan keskeisimpiin käytäntöihin. (Krasner & Pope 1988, 29-30). Asynkronisilla kutsuilla tarkoitetaan ajastettuja interaktioita, joiden aikana käyttäjällä on mahdollisuus jatkaa sovelluksen käyttöä prosessin kulun aikana. Esimerkiksi hiiren klikkaukset view-osassa kuuluvat tähän: käyttäjän ei tarvitse odottaa hiiren klikkauksen jälkeen sovellusta vaan sen käyttöä voidaan jatkaa samaan aikaan. (Brecht 2006, 1.)

Controller-osa vastaa aina sovittimien kuuntelemisesta. Sovittimia ovat esimerkiksi hiiri, näppäimistö ja kello.

## 2.4 UML-mallinnuskieli

UML (engl. *unified modeling language*) on standardoitu graafinen ohjelmistojen ja tietojärjestelmien mallinnuskieli (Sommerville 2010, 155). Se on joukko sovittuja graafisia merkintöjä, joita käytetään ohjelmien ja sovellusten suunnittelussa sekä niiden mallintamisessa (Fowler 2004, 14). UML on suunniteltu ja kehitetty toimimaan mahdollisimman monen ohjelmistokehitysmenetelmän rinnalla. Se ei ole sidoksissa ohjelmointikieliin, -tapaan tai -kulttuuriin, jolloin sitä voidaan käyttää mallinnuskielenä missä tahansa prosessissa sovellusalueesta riippumatta (Pender 2003, 9-12).

Object Management Group (OMG) standardoi UML:n marraskuussa vuonna 1997, kun OMG:n jäsenet hyväksyivät UML:n mallinnuskielien standardiksi sekä sopivat kielen jatkokehityksestä. UML:n versio oli tällöin 1.1, mutta työryhmät sopivat sen tippuvan takaisin versionumeroon 1. (Fowler 2004, 155.)

UML:n kehitys alkoi jo vuonna 1995, kun UML-kieltä edeltäneestä Unified method -mallinnuskielestä julkaistiin ensimmäinen versio. Myöhemmin Rational Software -nimisessä organisaatiossa työskentelevät Rumbaugh, Booch ja Jacobson antoivat työlleen nimen Unified Modeling Language. (Fowler 2004, 14.)

Vuonna 2005 julkaistiin toinen pääversio 2.0 UML-mallinnuskielestä (Unified Modeling Language 2005). Tämän jälkeen kieleen on tullut pieniä korjauksia. Esimerkiksi vuonna 2007 julkaistiin versio 2.1.2 ja versio 2.2 vuonna 2009. UML:a kehitetään jatkuvasti olemaan ajan tasalla ohjelmistokehityksessä. Versioon 2.0 lisättiin esimerkiksi käyttäytymistä ja suhteita kuvaavia kaavioita olio-ohjelmointikielien yleistyessä. Uusin UML:n versio on opinnäytetyön kirjoittamisen aikana 2.5.1, joka julkaistiin joulukuussa vuonna 2017. (OMG 2017.)

UML:n sisältämät diagrammit kuuluvat joko rakennekaavioihin (engl. *structure diagrams*) tai käyttäytymiskaavioihin (engl. *behavior diagrams*). Näiden rakennekaavioiden tehtävä on näyttää ohjelman staattinen rakenne. Kuvatut diagrammit sisältävät ohjelman luokat (engl. *class*), komponentit (engl. *components*) ja oliokaaviot (engl. *object diagrams*). (Fowler 2004, 74, 89.)

Esittelen seuraavaksi UML:n kaaviotyyppit sekä käytetyn struktuurin. Kuvaan myös elementit ja niiden käyttökohteet. Esimerkkeihin ja kaavioihin käytän kuvitteellista sosiaali-

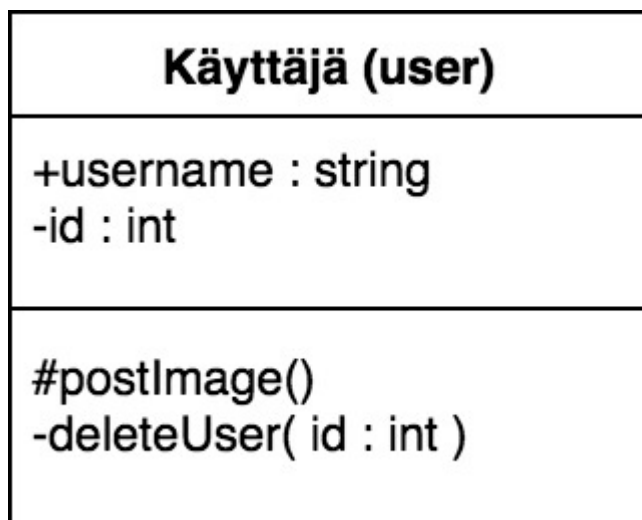
sen median palvelua, jossa käyttäjät voivat lähettää ja vastaanottaa kuvia. UML-dokumentaation laajuudesta johtuen olen rajannut tutkimuksen kannalta epäolennaiset asiat pois esittelystä.

#### 2.4.1 Luokkakaavio

Luokkakaavio (engl. *class diagram*) on tapa visualisoida luokkia ohjelmistossa ennen kuin sovelluksen tuotantovaihe alkaa. Se on aina staattinen malli ohjelmiston rakenteesta.

Luokkakaavio voi sisältää enintään kolme osaa. Luokan nimi (engl. *name*) esitetään aina kaavion yläosassa keskitettynä ja vahvennettuna. Luokan nimi on ensimmäinen osa kaavioita. Attribuutit (engl. *attribute*) ovat seuraava osa luokkakaaviota. Siinä esitetään ne luokan attribuutit, jotka vain tämä luokka sisältää. Alimpana kaaviossa ovat metodit (engl. *methods*). Metodi voi olla esimerkiksi luokalle kuuluva toiminto. Attribuutteja ja metodeja kutsutaan myös luokan jäseniksi (engl. *members*). (OMG 2005, 47-51.)

Kuviossa 2 esitellään luokkakaavio käyttäjästä. Voimme lukea kuviosta, että luokalla on kaksi attribuuttia ja kaksi metodia.



Kuvio 2. UML:n mukainen luokkakaavio käyttäjästä

UML:n mukaan attribuutilla on kaksi osaa: nimi ja tyyppi (engl. *type*). Esimerkiksi kuviossa 2 ensimmäisen attribuutin nimi on username, ja sen tietotyyppi on merkkijono (engl. *string*).

Luokkakaavion metodi on kaksiosainen. Ennen sulkuja kerrotaan metodin nimi. Kuvio 2 sisältää esimerkiksi metodin nimeltä *deleteUser*. Sulut lopettavat aina metodin nimen. Sulkujen sisään voidaan asettaa parametreja. Kuviossa 2 *deleteUser* metodilla on parametri *id*, ja sen tyyppi on integraali eli numero. Kyseisen metodin parametri viittaa saman luokan attribuuttiin.

Luokkakaavion attribuuteille ja metodeille voidaan asettaa haluttu näkyvyys (engl. *visibility*). Näkyvyydellä tarkoitetaan tässä tapauksessa oikeutta lukea ja käyttää kyseistä jäsentä luokassa tai alaluokissa. (OMG 2005, 133-134.)

Näkyvyys	
+	<b><u>public</u></b>
-	<b><u>private</u></b>
#	<b><u>protected</u></b>
~	<b><u>package</u></b>

Kuvio 3. UML luokkakaaviossa käytettävät näkyvyyden optiot

Luokan jäsenille on mahdollista asettaa yksi kuviossa 3 esitetyistä näkyvyyksistä. Jos näkyvyyttä ei aseteta, sen oletetaan olevan yksityinen (engl. *private*). (OMG 2005, 134.)

Kun jäsenelle halutaan asettaa julkinen näkyvyys, se merkitään aina merkillä '+'. Esimerkiksi kuviossa 2 attribuutille *username* on asetettu julkinen näkyvyys. Tämä tarkoittaa yksinkertaisimmillaan sitä, että käyttäjän nimi voidaan lukea mistä tahansa luokasta ohjelmiston sisällä. (OMG 2005, 134.)

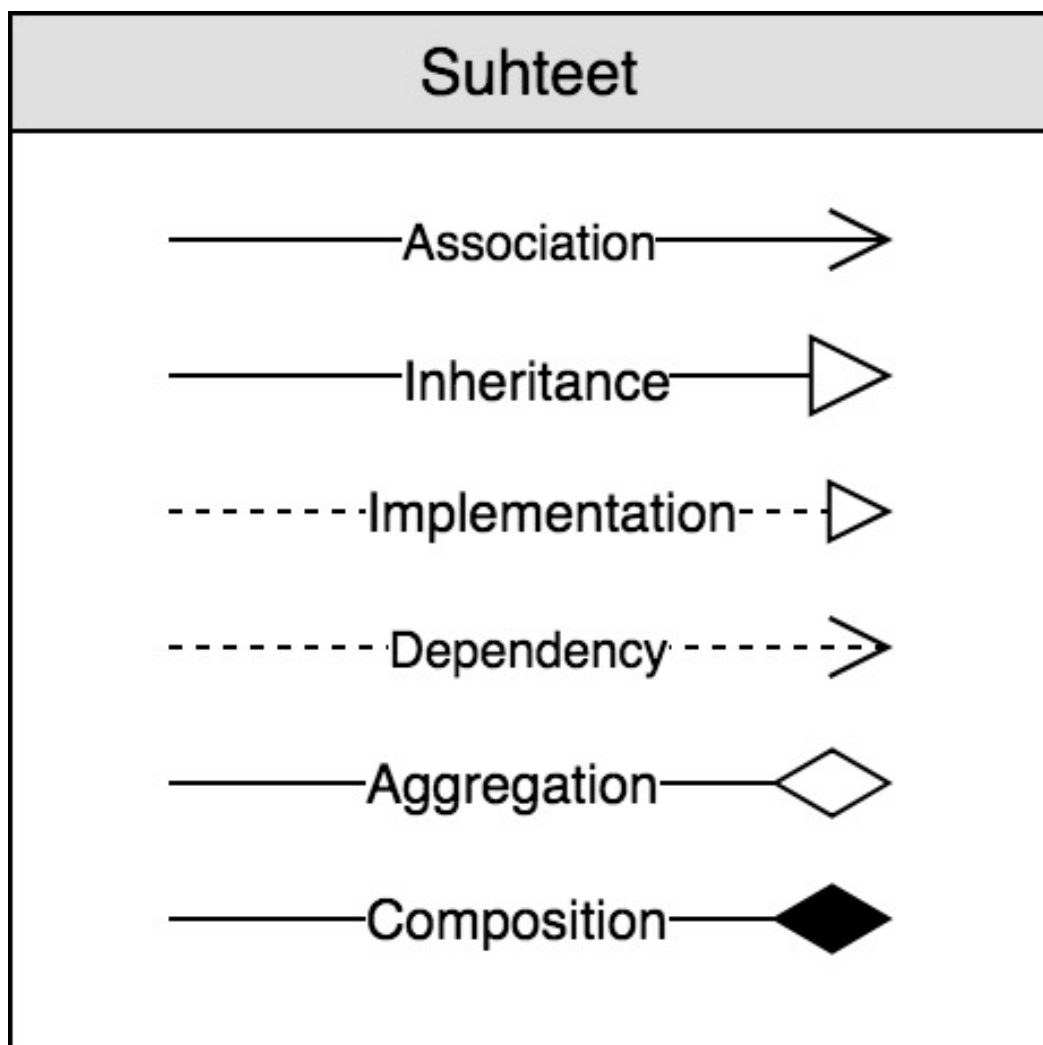
Yksityinen näkyvyys asetetaan jäsenelle merkillä '-', ja sillä merkitään esimerkiksi ne attribuutit luokasta, joiden ei anneta näkyä kuin sille luokalle, jonka osa se on. (OMG 2005, 134.)

Kun näkyvyys asetetaan merkillä '#', se tarkoittaa sitä, että kyseinen jäsen voidaan lukea sekä omasta luokasta että ylä- tai alaluokasta. (OMG 2005, 134.)

Näkyvyys saman tason (engl. *package*) muille luokille saadaan merkillä '~'. Tätä näkyvyyttä käytetään UML-kielessä harvemmin. (OMG 2005, 134.)

#### 2.4.2 Luokkien suhteet ja polut

Luokkakaaviossa luokkien välisiä suhteita (engl. *relationships*) kuvataan kahden luokan välissä kulkevalla viivalla. Esittelen nyt UML 2:ssa määritellyt luokkien suhteet ja sen, mihin niitä käytetään. Suhteita on yhteensä kuusi kappaletta, ja ne on esitetty kuviossa 4. (OMG 2005, 135-137.)



Kuvio 4. UML luokkien suhteet

## Assosiaatiosuhde

Assosiaatio (engl. *association*) on yleisin suhde UML-kielessä, ja sitä kuvataan mustalla viivalla, johon on liitettyä kuvion 4 kaltainen avoin kärki. Nuolenkärki on tärkeä osa suhdetta, ja sillä korostetaan mihin suuntaan luokkien välinen suhde osoittaa. Kärkeä ei kuitenkaan aina käytetä, jos suunnalla ei ole merkitystä tai se kulkee molempiin suuntiin (engl. *duo*). (OMG 2005, 36.)

## Perintäsuhde

Perintäsuhdetta (engl. *inheritance*) käytetään luokkien välillä silloin, kun alkuperäisestä luokasta (engl. *superclass*) vedetään kopioita (engl. *sub class*). Kopioilla voi olla spesiaaleja attribuutteja ja metodeja käytössä. Kopiot voivat lisäksi käyttää niitä jäseniä alkuperäisessä luokassa, joihin sille on annettu oikeudet. Perintäsuhdetta kuvataan suoralla viivalla, jossa on perässä suljettu kärki. (OMG 2005, 67.)

## Reaalisuhde

Reaalisuhdetta (engl. *implementation, realization*) käytetään kahden luokan välillä silloin, kun ensimmäinen luokka suorittaa toisessa luokassa spesifioidut tehtävät. Tätä suhdetta kuvataan katkoviivalla, jossa on päässä suljettu nuoli. (OMG 2005, 124.)

## Riippuvuus

Riippuvuussuhteella (engl. *dependency*) merkitään ne luokat, jotka ovat (tai joista toinen on) riippuvaisia toisistaan toimiakseen. Riippuvuutta merkitään assosiaatiosuhteen kaltaisella merkinnällä, missä suora viiva on korvattu katkoviivalla. (OMG 2005, 58.)

## Kompositio ja Kooste

Kompositio (engl. *composition*) ja koosteen (engl. *aggregation*) perässä käytetään salmiakin muotoista kärkeä kuvion 4 mukaisesti. Kärki osoitetaan siihen luokkaan, joka on tärkeämpi ja jonka alla on joitain muita olioita. Mustalla kärjellä kuvataan vielä astetta tärkeämpää ja spesifimpää suhdetta. Kompositiota ja koostetta käytetään silloin, kun halutaan korostaa ja selventää suhteiden painoarvoa. Jos korostusta ei tarvitse, käytetään usein tavallista assosiaatiosuhdetta. (OMG 2005, 35.)



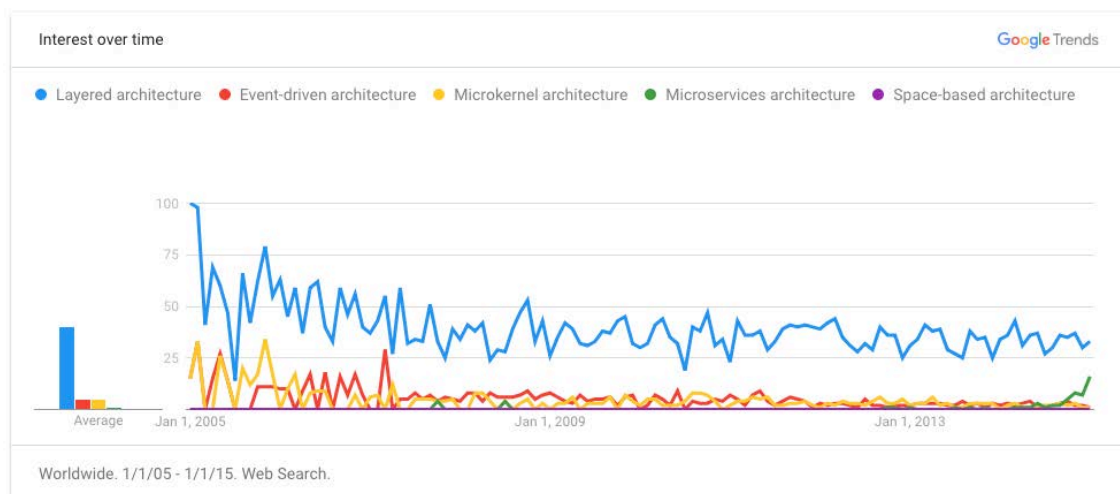
### 3 Metodit

Opinnäytetyössä on kerätty aineistoa teemahaastattelun avulla. Laadullisen tutkimuksen avulla pyrin selvittämään, miten MVC-metodia ja UML-kieltä käytetään. Yksi laadullisen tutkimuksen peruspiirteitä on tarkoituksenmukainen ja harkinnanvarainen otanta (Kananen 2008, 61-63). Tutkimuksen avulla pyrin antamaan johdonmukaisen tulkinnan sovellusarkkitehtuuristandardien käytöstä nykyaikaisten sovellusten suunnittelussa.

Laadullisessa tutkimuksessa lähtökohta on aina todellisen elämän kuvaaminen (Hirsjärvi, Remes & Sajavaara 2009, 151). Ennen tutkimuskysymyksen asettamista mietin tarkkaan, mitä oikeastaan haluan tutkia. Työskennellessäni kehittäjänä ohjelmistokehityksen menetot ja standardit tulivat tutuiksi. Tässä opinnäytetyössä haluan konkretisoida niitä olettamuksia ja käytäntöjä, joihin olen törmännyt sovellusarkkitehtuurin suunnittelussa. Laadullisen tutkimuksen avulla pyrin selvittämään sovellusarkkitehtuuristandardien käyttöä osana nykyaikaisten sovellusten suunnittelua. Esitän seuraavassa alaluvussa kaksi aiheeseen liittyvää hypoteesia. Pohjaan hypoteesini Baltesin ja Diehlin (2013) julkaisemaan tutkimukseen sekä arkkitehtuurimetodien suosioon ohjelmistotalalla (Google Trends n.d.). Lopuksi työn analysointivaiheessa pohdin, pitävätkö hypoteesit paikkansa.

#### 3.1 Hypoteesit

Nykyaikaisten sovellusten ja web-aplikaatioiden suunnittelu- ja arkkitehtuurityökaluja hakiessa esille nousee vahvasti MVC. Esimerkiksi hakemalla hakusanoilla "Web application architecture patterns", ensimmäiset hakutulokset suosittelivat MVC:n käyttöä arkkitehtuurissa. Samaan olen törmännyt toimiessani ohjelmistokehittäjänä: modulaaristen web-aplikaatioiden suunnitteluun käytetään ketteriä MVC-painotteisia arkkitehtuurin muotoja. Tämä on ensimmäinen hypoteesini: sovellusarkkitehtuurimodeista MVC (Model-View-Controller) on nykyään kaikkein tehokkain, ja sitä käytetään useammin kuin muita arkkitehtuurimodeja. Tämän opinnäytetyön puitteissa ei ollut mahdollista suorittaa laajamittaista kvantitatiivista tutkimusta eikä sovellusarkkitehtuurien käytöstä löytynyt aikaisempaa tutkimusta. Pohjaan MVC:n käyttöön liittyvän hypoteesin Googlen julkaisemiin hakutulostrendeihin. Ottamalla viisi yleisimmin käytettyä sovellusarkkitehtuurimetodia (Richards 2015, 1, 11, 21, 27), ja vertailemalla hakutulosten trendiä vuosien 2005 ja 2015 välillä, voidaan vetää selkeä johtopäätös. MVC on kerrosarkkitehtuureihin (engl. Layered architecture) kuuluvista arkkitehtuurimodeista käytetyin (Google Trends n.d.).



Kuvio 5. Sovellusarkitehtuurimetodien suosio Googlen hakutuloksissa (Google Trends n.d.)

Toinen hypoteesini on liittyy UML-mallinnuskieleen. Vaikka UML (Unified Modeling Language) on alan standardi, sen käyttö on epäolennaista nykyaikaisten sovellusten suunnittelussa. Tämä on kiistellympi kahdesta esitetystä hypoteesista ja herättää varmasti keskustelua. En kiistä UML:n tai muiden vastaavien mallinnuskielten hyötyä sovellusten suunnittelussa. Haluan tutkia, kuinka tosissaan sitä käytetään juuri web-aplikaatioiden tai muiden vastaavien sovellusten suunnittelussa. Baltes ja Diehl (2013) argumentoivat tekemässään tutkimuksessa sovellusten suunnittelukielten olevan lähtökohtaisesti luonnoksia ja piirroksia, jotka pohjaavat vain vähän OMG:n asettamiin standardeihin. Pohjaan toisen esittämäni hypoteesin UML:n käytöstä Baltesin ja Diehlin tutkimukseen.

Esitettyjen hypoteesien kautta pyrin vastaamaan tutkimuskysymykseen: millä tavalla sovellusarkitehtuurstandardeja käytetään nykyaikaisten sovellusten suunnittelussa.

### 3.2 Toteutus ja rakenne

Käytän aineiston hankintaan kohdennettua teemahaastattelua. Haastattelun aikana edetään ennalta suunniteltujen teemojen kautta (Hirsjärvi & Hurme 2000, 47-48). Jaoin haastattelun sisällön kolmeen osaan. Ensimmäisessä osassa haastateltavan tehtävänä oli suunnitella sovellusarkitehtuurin keinoin yksinkertainen web-aplikaatio kuvanjakopalvelusta. Toisessa osassa esitettiin kysymys MVC:n käytöstä ja kolmannessa UML:n käytöstä.

Haastattelun ensimmäisessä osassa suunniteltavaan applikaatioon otettiin mallia tunnetusta sosiaalisen median kuvanjakopalvelusta *Instagramista*. Applikaatioon tuli suunnitella vähintään seuraavat ominaisuudet: käyttäjäprofiili, kuvien lähettäminen ja kuvien lukeminen. Applikaatiolla oletettiin olevan täydellinen järjestelmäarkkitehtuuri. Tämä tarkoittaa sitä, että esimerkiksi API-kutsuihin ei tarvinnut kiinnittää huomiota, jolloin voitiin keskittyä kokonaan sovellusarkkitehtuurin suunnitteluun.

Kun haastateltava oli tyytyväinen suunniteltuun sovellusarkkitehtuuriin, haastattelun toinen osa lähti käyntiin. Toisen osan tavoitteena oli selvittää haastateltavan ajatuksia MVC:n käytöstä osana arkkitehtuuria. Jos haastateltava oli alusta asti käyttänyt tätä metodia, häneltä kysyttiin tarkentavia kysymyksiä. Jos taas haastateltava ei ollut käyttänyt MVC:tä, pyydettiin häntä miettimään, haluaako hän muuttaa sovellusarkkitehtuurista mitään, jos se pitäisi tehdä MVC:n ehdoilla.

Haastattelun viimeisessä vaiheessa selvitettiin UML:n käyttöä osana suunniteltua arkkitehtuuria. Haastateltava sai kertoa vapaasti omia ajatuksiaan mallinnuskielestä ja sen käytöstä osana piirrettyä arkkitehtuuria. On tärkeä huomioida, että haastateltavaa ei oltu pyydetty käyttämään mallinnuskieltä. Mallinnuskielen käyttö tai käytöstä jättäminen oli täten täysin omavaraista. Analyysissä otetaan tämä ja tilanteen osittainen hektisyys huomioon.

Tutkimusta varten haastattelin neljää ohjelmistoalan ammattilaista sovellusarkkitehtuurin käytöstä. Teemahaastattelun kannalta tärkeää on, että kaikilla haastateltavilla on aikaisempaa kokemusta teemasta (Hirsjärvi & Hurme 2002, 59), tässä tapauksessa arkkitehtuureista ja sovellusten suunnittelusta. Haastateltavat on valittu eliittiotannalla, eli haastatteluun valitaan vain henkilöitä joilla on parasta mahdollista tietoa tutkittavasta asiasta (Tuomi & Sarajärvi 2002, 88). Haastattelijoiden valinnan perusteena on ensisijaisesti ollut työkokemus arkkitehtuurien käytöstä, sekä taito esiintyä ja ilmaista itseään.

Dokumentoin haastattelut videokameralla sekä varalta audiotallentimella. Heti haastattelutilanteen jälkeen kirjoitin ylös ensimmäiset ajatukseni haastattelutilanteesta. Lisäksi seuraavana päivänä litteroin ja muutin haastatteluaineiston tekstimuotoon. Katsoin ja kuuntelin jokaisen haastattelun kolme kertaa läpi. Ensimmäisellä kerralla kävin haastattelut läpi ajatuksen kanssa. Toisella kerralla litteroin, eli kirjoitin ylös haastattelijan aja-

tukset ja suorat lainaukset. Kolmannella kerralla kirjoitin ylös haastattelun ensimmäisessä osassa rakennetun arkkitehtuurin työ- ja piirtovaiheet. Jokaisesta haastattelusta jäi myös analysoitavaksi piirretty sovellusarkkitehtuurisuunnitelma.

Ensimmäisessä haastattelussa testasin tutkimusasetelman ja -kysymyksen toimivuutta. Tässä haastattelussa on piirretty ja rakennettu kuvanjakopalvelulle arkkitehtuuri, mutta jatkokysymyksiä UML-kielen ja MVC-metodin käytöstä ei ole esitetty. Haastattelun jälkeen tutkimusasetelma tarkentui niin, että haastattelu jaettiin kolmeen osaan. Haastateltavana oli 25-vuotias Vili (nimi muutettu), joka työskentelee keskikokoisessa organisaatiossa nimikkeellä back-end ohjelmistokehittäjä. Tässä vaiheessa on hyvä huomioda, että kaikkien haastateltavien nimet on muutettu ja ikä pyöristetty yksityisyyden suojelemiseksi. Lisäksi jätän mainitsematta haastateltavien mahdollisen työpaikan sekä tarkan työnimikkeen.

Toinen haastateltavani oli Jaakko, 40-vuotias ohjelmisto- ja järjestelmäasiantuntija. Jaakko työskentelee suomalaisessa ohjelmistopalveluja tarjoavassa yrityksessä.

Kolmanneksi haastattelin Sepeä, 25-vuotiasta ohjelmistokehittäjää. Sepe työskentelee nimikkeellä web-kehittäjä isossa ohjelmisto- ja palveluyrityksessä.

Neljäs haastateltavani oli pitkän linjan ohjelmisto- ja web-kehittäjä, 25-vuotias Juha. Haastattelin Juhan videopuhelun ja etäpiirtoalustan avulla. Juha työskentelee isossa ohjelmistotalossa web-kehittäjänä.

Valitsin haastateltavat työkokemuksen perusteella. Kaikki haastateltavat ovat työskennelleet ohjelmistokehittäjänä ja käyttäneet aikaisemmin arkkitehtuureja osana sovelluksen suunnittelua. Kiinnostavan aiheen vuoksi jokainen haastattelu kesti vähintään 50 minuuttia ja useamman kohdalla keskustelu jatkui reilusti yli siihen budjetoidun ajan. Koin neljän haastattelun jälkeen, että niistä muodostuu mielekäs analyysi joka vastaa tutkimuskysymykseen.

### 3.3 Aineiston analysointi

Aineiston analyysimenetelmänä käytän teemoittelua. Teemoittelulla voidaan etsiä aineiston teemoja, toisin sanoen pääkohtia. Pääkohtien avulla voidaan vetää aineistosta johtopäätöksiä, mutta vain jos aineistoon on perehdytty kunnolla. Teemoittelulle yleinen haaste on se, että myös väärrien tai epäselvien johtopäätösten vetäminen on mahdollista.

(Moilanen ja Räihä 2010, 55–57.) Aineiston teemoittelun lisäksi tulkitseen haastattelijoiden rakentamia luonnoksia kuvanjakopalvelun arkkitehtuurista suoraan.

Etsin aineistosta teemoja kuuntelemalla haastatteluja läpi, litteroimalla ja kirjoittamalla haastattelut auki kronologisessa järjestyksessä (ks. luku 4). Lisäksi olen pyrkinyt hakemaan ymmärrystä teemoista selvittämällä miten, ja missä järjestyksessä haastateltava rakentaa kuvanjakopalvelun arkkitehtuurin. Aineistosta esiin nousseet teemat on esitetty taulukossa 1.

Taulukko 1. Aineistosta esiin nousseita teemoja

Teemat	Vili	Jaakko	Sepe	Juha
MVC-arkkitehtuurin osien rooli nähdään epäselvänä	x		x	x
UML-mallinnuskieltä käytetään soveltavasti	x	x	x	x
MVC-arkkitehtuuria ei nähdä kestäväenä metodina		x	x	x
Kehittäjän rooli organisaatiossa vaikuttaa MVC-arkkitehtuurin käyttöön	x	x	x	

Aineiston pohjalta nousee esille se, että MVC-arkkitehtuurin osien rooli (ks. luku 2) nähdään epäselvänä, vaikka arkkitehtuurimetodin käyttö muuten on sujuvaa. Tämä on ensimmäinen teema. Toinen esiin noussut teema on se, että UML-mallinnuskieltä käytetään soveltavasti osana arkkitehtuurin rakentamista, riippumatta siitä kuinka tuttu tämän standardi on. Kolmas teema on se, että MVC-arkkitehtuuria ei nähdä kestäväenä metodina. Neljäs teema on se, että vaikka puhutaan alan standardeista, kehittäjän rooli organisaatiossa vaikuttaa MVC-arkkitehtuurin käyttöön.

Luvussa 5 kerron ja esittelen teemat tarkemmin. Seuraavassa luvussa esittelen tutkimuksen aineiston.

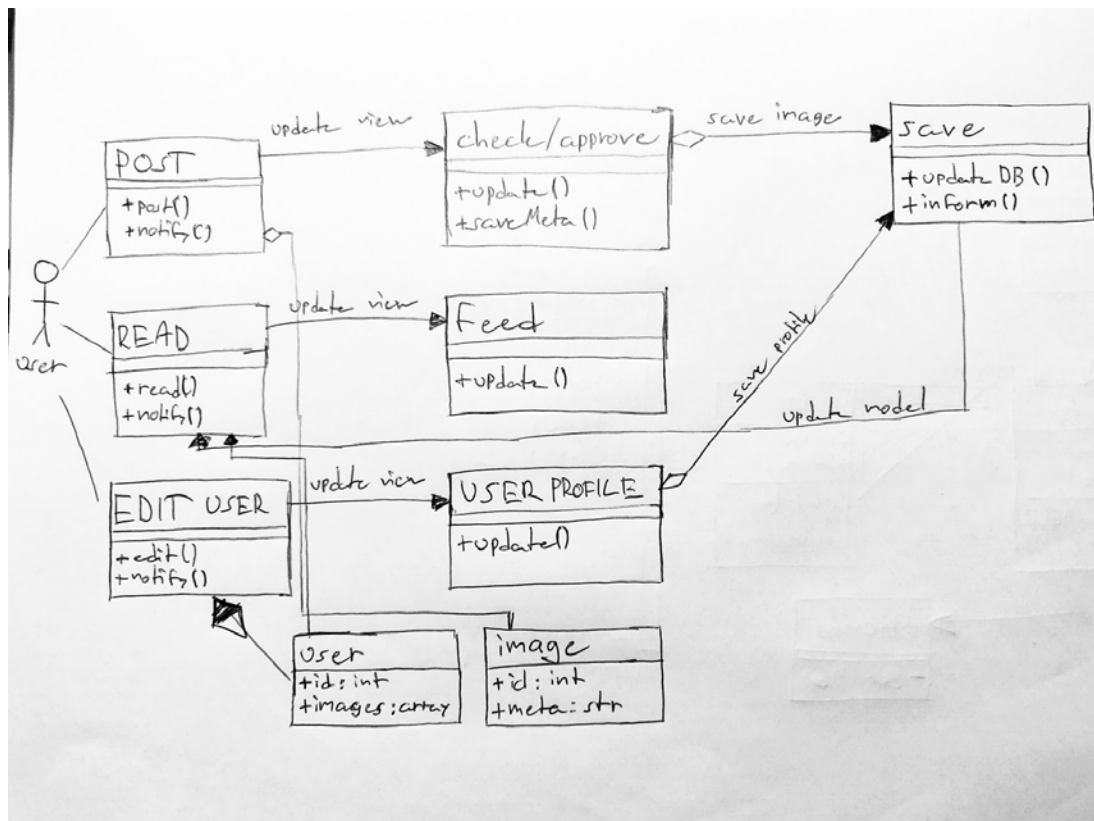
## 4 Aineisto

Tässä luvussa esittelen tutkimuksen aineiston. Esittelen ensin yhden mahdollisen standardeihin perustuvan arkkitehtuuriluonnoksen. Sen jälkeen käyn läpi neljä haastattelua.

Pyrin aineistoa läpikäydessä kuvaamaan haastateltavien päätöksiä, pohdintaa ja prosessia arkkitehtuurin rakentamisen aikana. Etenen jokaisen haastattelun kohdalla kromologisessa järjestyksessä.

#### 4.1 Yksi mahdollinen tapa esittää kuvanjakopalvelu

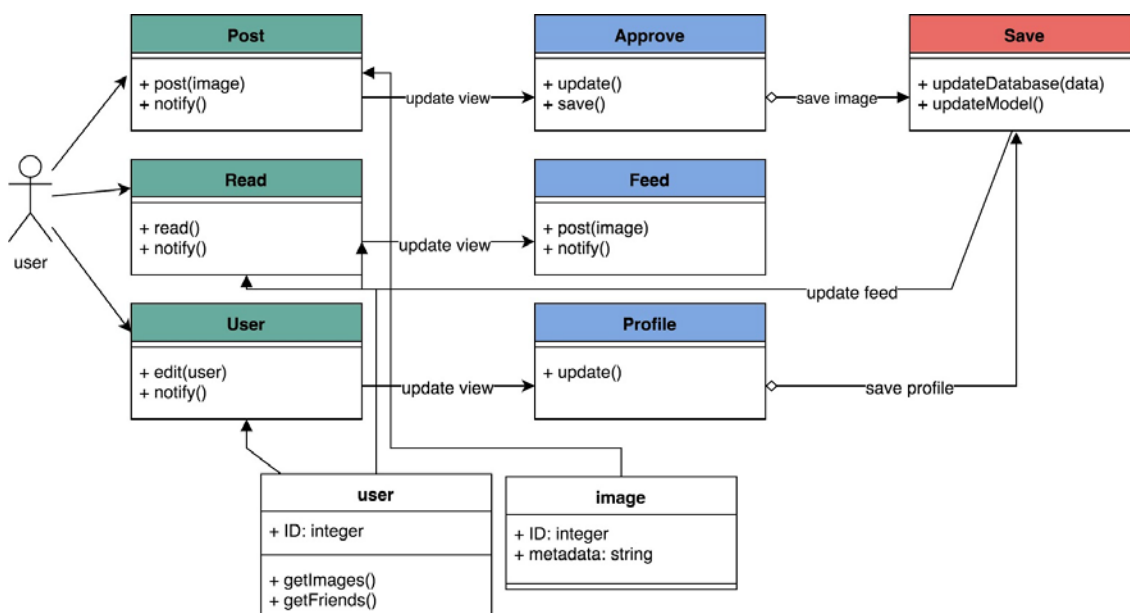
Tutkimusta varten rakensin kuvanjakopalvelu *Instagramille* sovellusarkkitehtuurin UML ja MVC-standardien mukaisesti. Rakennetun luonnoksella avulla pyrin avaamaan yhden mahdollisen tavan esittää kuvanjakopalvelun arkkitehtuuria. Lisäksi pyrin sen avulla pääsemään mahdollisimman lähelle luvussa 2 esitettyjen standardien todellista käyttöä. Olen tehnyt luonnoksen sitä varten, että haastatteluiden aikana rakennettuja arkkitehtuureja olisi mahdollista vertailla standardien todelliseen käyttöön. Kutsun tässä luvussa luonnosteltua sovellusarkkitehtuuria *applikaatioksi* lukemisen helpottamiseksi.



Kuvio 6. Piirretty luonnos applikaation arkkitehtuurista

Haastattelijoina pyydettiin luomaan kuvanjakopalvelusta yksinkertainen versio, jossa on vähintään kolme ominaisuutta: käyttäjäprofiili, kuvien lähettäminen ja kuvien lukeminen. Kuviossa 6 olen luonnostellut nämä ominaisuudet vasempaan laitaan: *post*, *read* ja *edit*

*user*. Näiden ominaisuuksien ympärille olen tämän jälkeen piirtänyt UML-mallinnuskieltä käyttäen vaadittavat toiminnallisuudet. Kuviossa 7 olen mallintanut applikaation arkkitehtuurin ja värikoodannut toimintalogiikan kuvaamisen helpottamiseksi.



Kuvio 7. Puhtaaksi piirretty luonnos applikaation arkkitehtuurista

MVC-arkkitehtuurissa on kolme osaa: model, view ja controller. Kuviossa 7 MVC:n osat on värikoodattu. Model-osa on vihreä, view-osa on sininen ja controller-osa on punainen. MVC:n osien kiertokulku esitettiin aikaisemmin kuviossa 1. Informaatio liikkuu osien välillä samalla tavalla kuviossa 7. Vihreistä model-osista lähtee assosiaatiosuhde sinisiin view-osiin. Sinisistä view-osista lähtee kompositiosuhde punaiseen controller-osaan. Ja punaisesta controller-osasta lähtee assosiaatiosuhde takaisin vihreään read-model-osaan.

Kuvion 7 alaosassa on kaksi luokkakaaviota: *user* ja *image*. Luokkakaaviolla *user* kuvataan yksittäistä käyttäjää, ja sillä on attribuuttina *id*. Metodeina *user*-luokkakaaviolla on *getImages* ja *getFriends*, joilla kuvataan käyttäjän mahdollisia toimia tämän luokkakaavion sisällä. Luokkakaaviolla *image* kuvataan yksittäistä kuvaa, ja sillä on attribuutteina *id* ja *metadata*. Metadata-attribuutti pitää sisällään kuvan yleiset tiedot. *User*-luokkakaaviosta on vedetty assosiaatiosuhde sekä vihreään *user*-modeliin, että vihreään *read*-modeliin. Tällä kuvataan niiden välistä suhdetta. Esimerkiksi käyttäjän lukiessaan kuvia (vihreä *read*-metodi), se voisi mahdollisesti käyttää *user*-luokkakaaviosta löytyviä *getFriends*- ja *getImages*-metodeita. Samalla tavalla luokkakaaviosta *image* on vedetty

assosiaatiosuhde vihreään post-modeliin. Post-model voisi käyttää tätä luokkakaaviota lähettäessään uutta kuvaa.

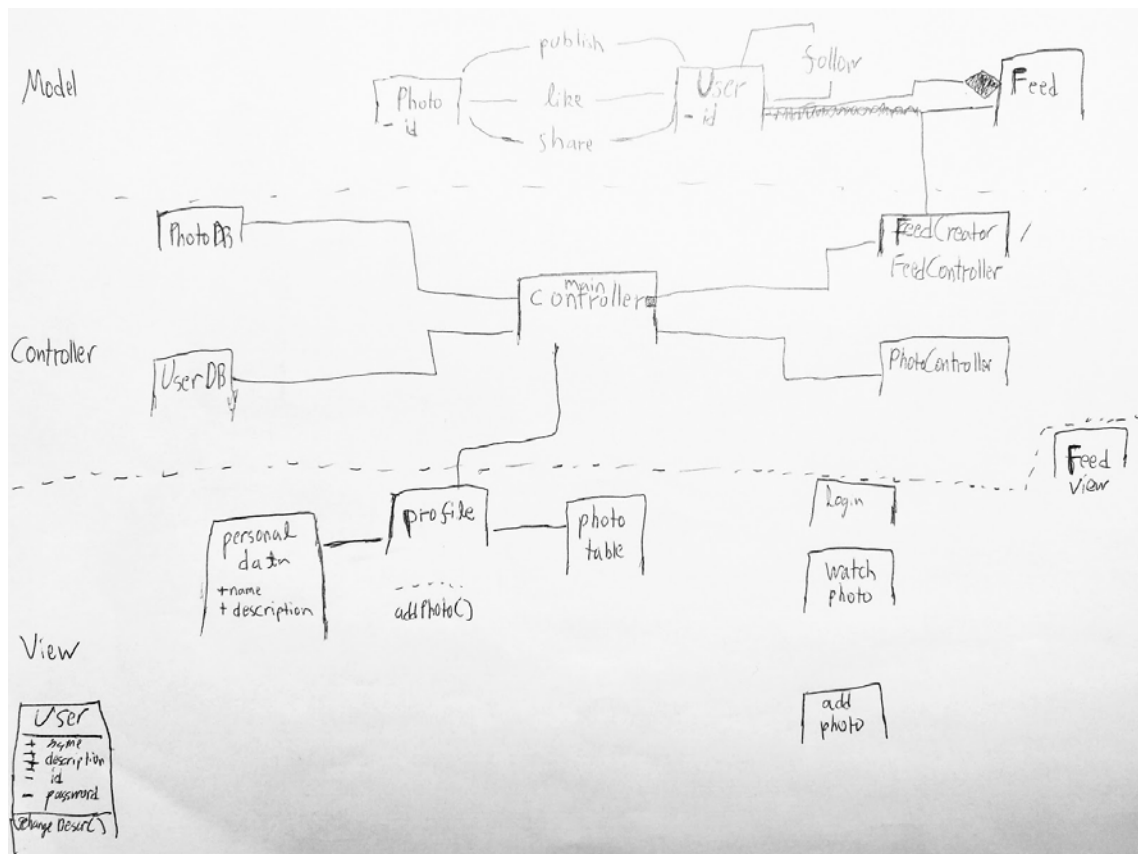
Kuvion 7 vasemmassa laidassa on käyttäjä *user*, josta lähtee kolme nuolta. Näillä assosiaatiosuhteilla kuvataan käyttäjän toimia (engl. *actions*). Käyttäjä voi siis tehdä kaikki kolme vaadittua toimenpidettä applikaatiossa. Käyttäjä voisi esimerkiksi lukea muiden käyttäjien kuvia. Kuviossa 7 tätä kuvataan vihreällä read-modelilla. Alhaalla olevaa *user*-luokkakaavion metodia *getFriends* voitaisiin käyttää käyttäjän ystävien hakemiseen ja *getImages*-metodia ystävien kuvien näyttämiseen. Vihreän read-modelin ja sinisen feed-viewin välillä on *update view* assosiaatiosuhde, joka kuvaa view-osan päivittymistä aina silloin kun model-osassa tapahtuu jotain. Punaista feed-controlleria voisi tarvita esimerkiksi, kun kuvaa tallennetaan.

Kuvioissa 6 ja 7 esitellyn arkkitehtuuriluonnoksen avulla on pyritty antamaan yksi näkökulma kuvanjakopalvelun sovellusarkkitehtuuriin. Seuraavassa luvussa esittelen haastattelujen aikana piirrettyjä arkkitehtuuriluonnoksia.

#### 4.2 Vilin haastattelu

Vilin haastattelu on ensimmäinen neljästä haastattelusta. Vilin haastattelun jälkeen tutkimusasetelma tarkentui niin, että haastattelu jaettiin kolmeen osaan. Viliä on pyydetty rakentamaan yksinkertainen arkkitehtuuri kuvanjakopalvelulle, mutta tarkentavia kysymyksiä MVC:n ja UML:n käytöstä ei esitetty haastattelun aikana.

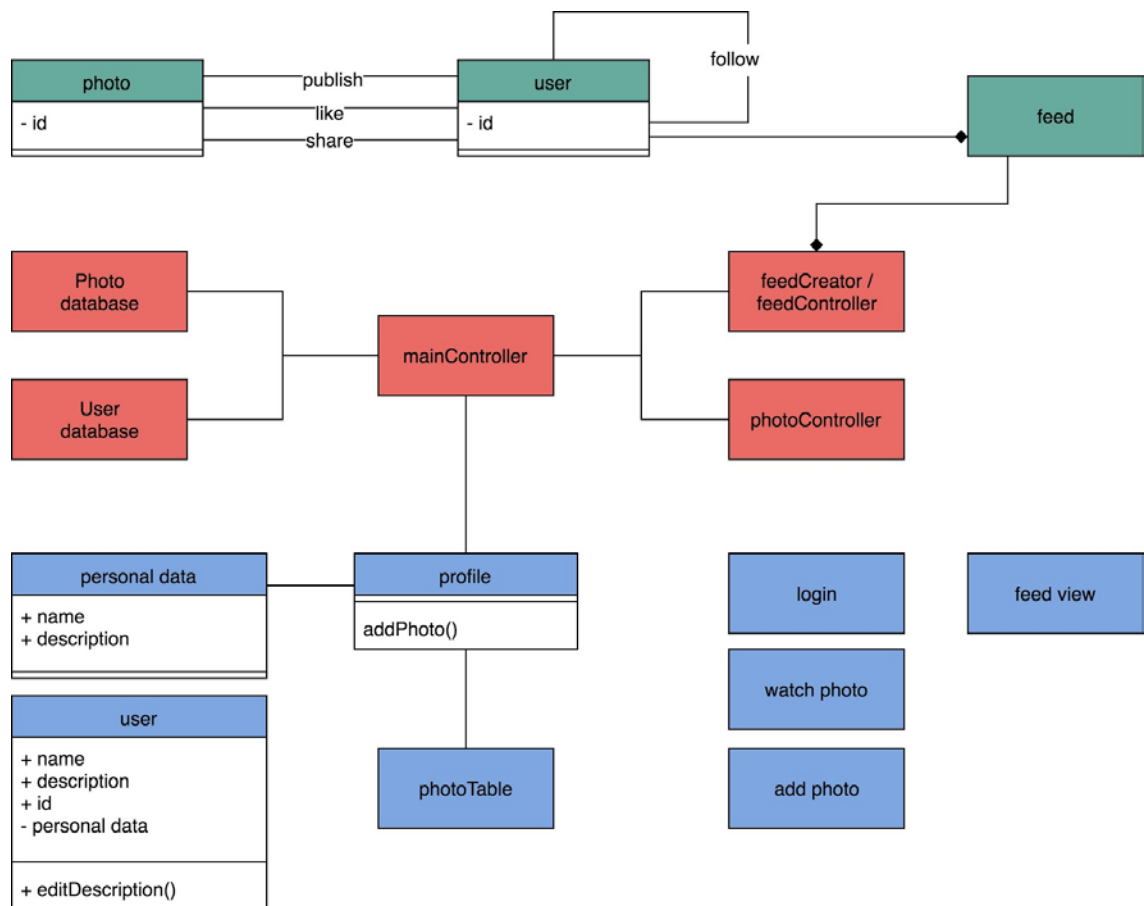




Kuvio 8. Vilin luonnos

Kuviossa 8 on Vilin rakentama sovellusarkkitehtuuri kuvanjakopalvelusta. Käyn tässä luvussa läpi Vilin haastattelun ja arkkitehtuurin rakentamisen kronologisessa järjestyksessä. Kuviossa 9 on mallinnettu ja värikoodattu versio Vilin luonnoksesta.

Ensimmäisenä Vili jakoi sovelluksen kolmeen osaan ja kirjoitti luonnoksen vasempaan laitaan MVC:n kolme osaa: model, controller ja view. Hän kertoi oppineensa arkkitehtuurimetodien käytöstä koulussa ja käyttävänsä MVC-metodia töissä. Vili mietti hetken aikaa mistä MVC:n osasta hän lähtee liikkeelle. Vili päätyi aloittamaan model-osasta. Kysyttäessä miksi hän aloittaa modelista, Vili kertoi sen olevan helppoa koska hän työskentelee back-end-ohjelmoijana. Vili pohti myös sitä, että hän on huomannut front-end-ohjelmoijien aloittavat arkkitehtuurin rakentamisen view-osasta ensin, eli sovelluksen visuaalisesta puolesta.



Kuvio 9. Vilin rakentama arkkitehtuuri

Vihreässä model-osassa Vili piirsi ensin auki kuvan ja käyttäjän suhteet. Vili veti UML:n standardin kaltaisilla assosiaatiosuhteilla kolme viivaa joihin hän kirjoitti *publish*, *like* ja *share*. Lisäksi laatikon *user* oikealle puolelle hän kirjoitti *follow* ja veti siitä kaksi viivaa takaisin laatikkoon. Näistä kahdesta laatikosta Vili veti vielä yhden painotetun assosiaatiosuhteen laatikkoon joka on nimetty *feed*. Näillä kaikilla hän kertoi kuvaavansa Instagramin toimintalogiikkaa applikaation ytimessä.

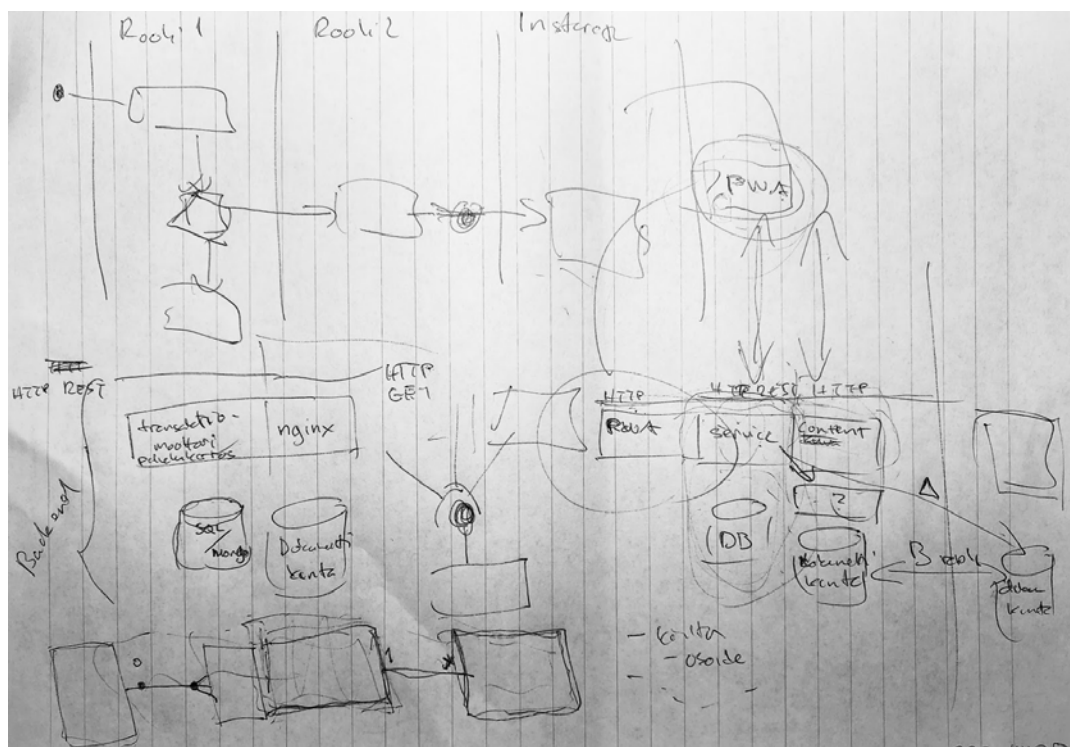
Seuraavaksi Vili rakensi arkkitehtuurin sinisen view-osan. Hän aloitti piirtämällä laatikon *profile*, ja veti siitä kaksi viivaa molemmille puolille: vasemmalle puolelle Vili piirsi *personal data* nimisen laatikon ja oikealle *photo table* nimisen laatikon. Vili sanoi miettivänsä tätä osaa käyttäjäprofiilin kautta. Lisäksi Vili piirsi luonnoksen oikealle puolelle kolme hänen mielestään tärkeää view-osaan kuuluvaa näkymää: *login*, *watch photo* ja *add photo*. Vili mainitsi jättävänsä joitain asioita piirtämättä, koska ne eivät vaikuta arkkitehtuurin ymmärtämiseen. Vielä erikseen Vili piirsi luonnoksen oikeaan laitaan *feed view* nimisen laatikon. Tämä laatikko vastasi Vilin sanojen mukaan sovelluksen keskeisintä toimea eli kuvien lukemista.

Viimeisimpänä MVC:n osista Vili piirsi luonnoksen keskelle punaisen controller-osan. Vili piirsi yhden ainoan laatikon ja nimesi sen *mainController*, joka sisältäisi kaikki controller-osat. Tässä kohtaa Vili sanoi pitävänsä controllerin tehtävää itsestään selvänä. Vili piirsi vielä vasemmalle kaksi controlleria, jotka vastasisivat informaation liikkumisesta tietokantoihin. Oikealle puolelle Vili piirsi *FeedCreator / FeedController* ja *PhotoController* controllerit. Lipuksi kaikki controllerit yhdistettiin assosiaatiosuhteen kaltaisella viivalla, ja *FeedController* ylös model-osaan.

Vili ei ollut piirtänyt yhdellekään laatikolle pohjaa. Hän kertoi sen olevan arkkitehtuurin piirtämistä tehostava keino, koska ikinä ei tiedä, kuinka paljon attribuutteja tai metodeja mihinkin kohtaan kirjoitetaan. Jättämällä alimman viivan piirtämättä työskentelyä voitaisiin jatkaa kunkin laatikon kohdalla. Lopuksi Vili piirsi vielä luonnoksen vasempaan alanurkkaan UML:n mukaisen luokkakaavion käyttäjästä.

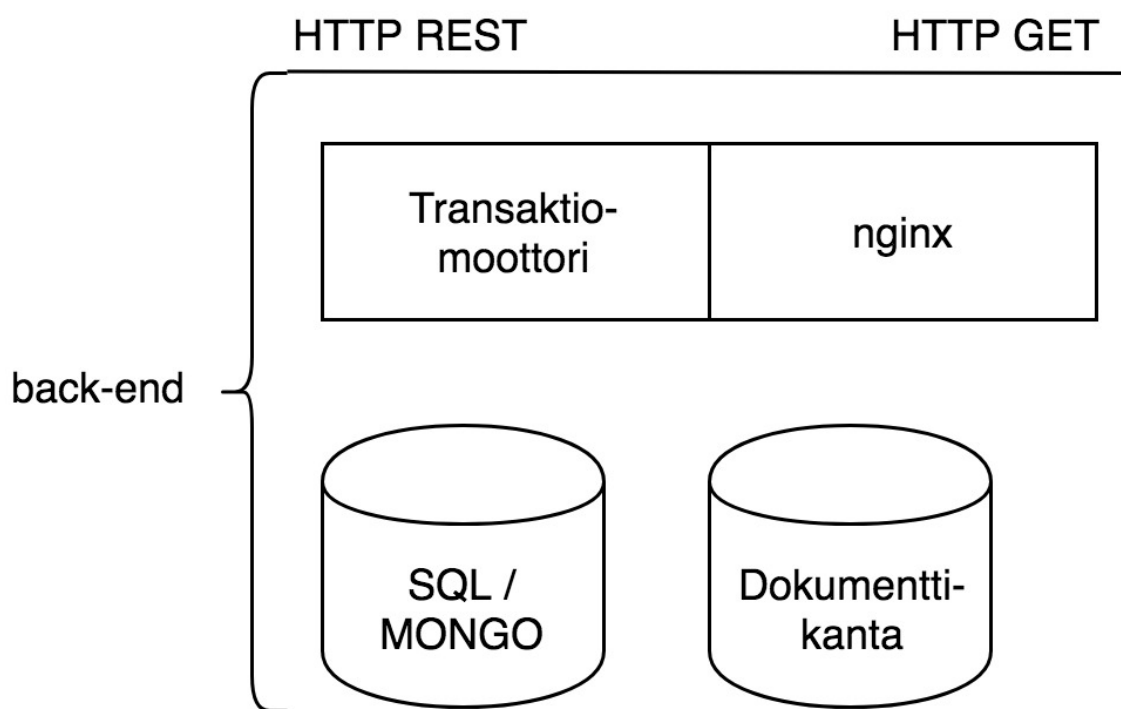
#### 4.3 Jaakon haastattelu

Jaakon haastattelu oli toinen neljästä haastattelusta. Jaakon kanssa käydyssä haastattelussa on kolme osaa: kuvanjakopalvelun arkkitehtuurin rakentaminen, kysymys MVC:n käytöstä ja kysymys UML:n käytöstä.



Kuvio 10. Jaakon luonnos

Käyn Jaakon haastattelun kronologisesti läpi. Kerron ensin miten Jaakko rakensi arkkitehtuurin kuvanjakopalvelulle. Sen jälkeen kerron haastattelun toisen ja kolmannen vaiheen keskustelussa esiin nousseita kohtia MVC:n ja UML:n käytöstä suhteessa rakennettuun arkkitehtuuriin. Jaakko rakensi järjestelmäarkkitehtuurin sovellusarkkitehtuurin sijasta, eli hän on piirtänyt informaation liikkumisen applikaation tasojen välillä yksittäisten toiminnallisuuksien sijasta.

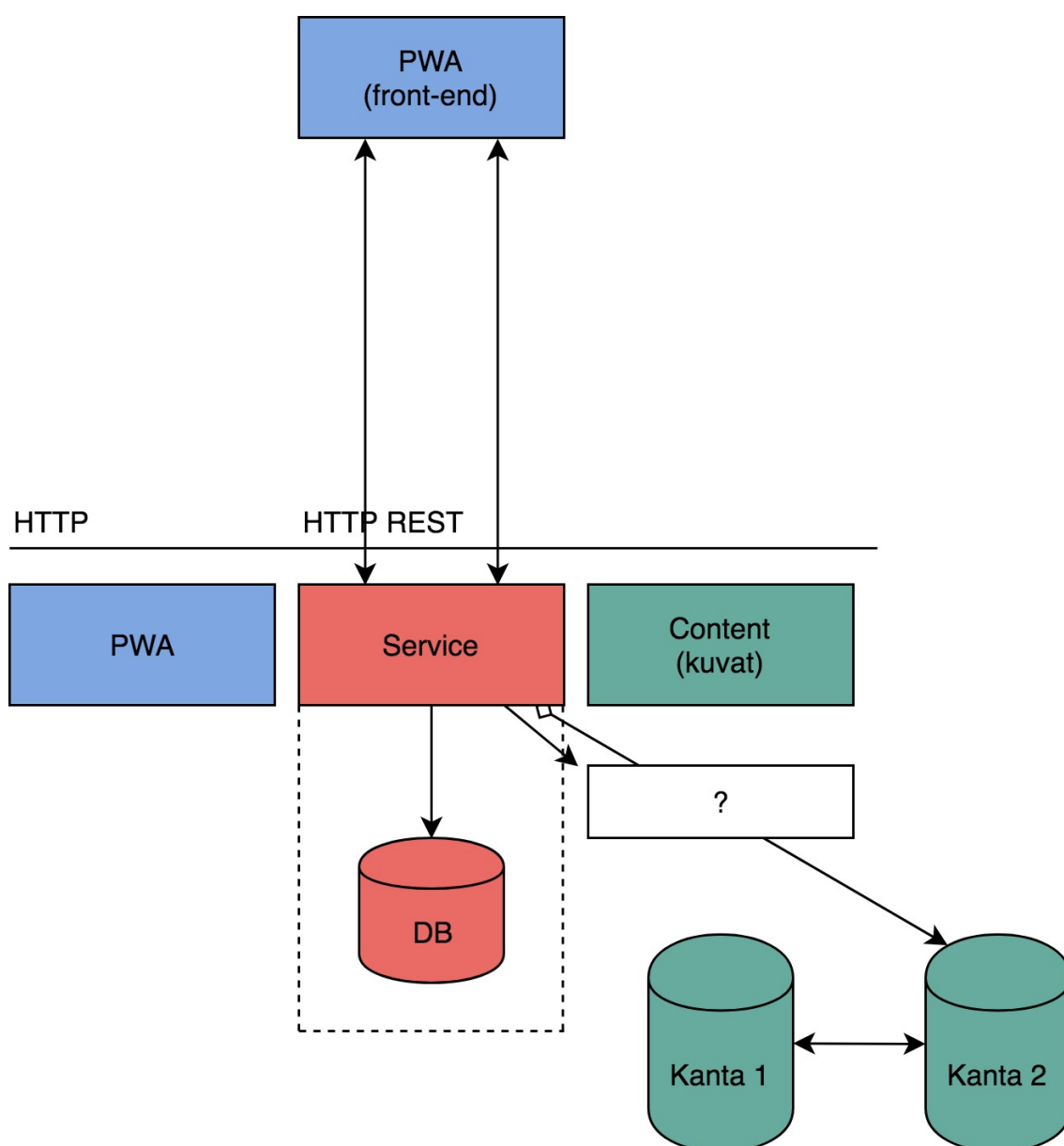


Kuvio 11. Jaakon piirtämä transaktiomoottori (tietokanta)

Jaakko aloitti arkkitehtuurin rakentamisen miettimällä, miten kuvanjakopalvelussa voidaan käsitellä ja siirtää niin paljon kuvia nopeasti ja tehokkaasti. Hänen vastauksensa oli transaktiomoottori (engl. *transaction engine*), joka vastaa kuvien tallennusmekaniikasta ja siirrosta. Transaktiomoottori on tässä kohden synonyymi tietokannalle ja tietokantarakenteelle. Jaakon omin sanoin: “homman ytimessä on valtava määrä kuvia ja käyttäjiä, skaalautuvuus on se juttu”. Jaakon luonnos tietokannasta on kuvion 9 vasemmassa laidassa. Kuviossa 11 sama moottori on mallinnettuna.

Jaakko kirjoitti ensin kuvion 11 mukaan *HTTP REST* ja *HTTP GET* -rajapinnan kuvion yläosaan. Jaakko selitti samalla tämän olevan back-end. Tästä rajapinnasta olisi mahdollista hakea vaadittu informaatio, tärkeimpänä kuvat. Jaakko sanoi nginx-moottorin vastaavan kuvien lähetyksestä eteenpäin.

Tämän jälkeen Jaakko pohti sovellusten elinkaarta: “Sovellusten arkkitehtuurissa on ongelmana se, että pitää pystyä arvioimaan tekniikoiden elinkaari. Pitää valita arkkitehtuuri niin, että sillä on tukea mahdollisimman pitkään.” Jaakko jatkoi kertomalla, että joitain hänen käyttämiään arkkitehtuureita ei enää käytetä: “Mä oon käyttänyt aikanaan joitain arkkitehtuureja, jotka on sittemmin kuollut pois, esimerkiksi xforms. Vähän kuin Yahoo UI. Frameworkit ja arkkitehtuurit menee ja tulee.”



Kuvio 12. Jaakon piirtämä arkkitehtuuri

Jaakon piirtämä kuvanjakopalvelun arkkitehtuuri on puhtaaksi mallinnettuna kuviossa 12. Aluksi Jaakko jatkoi rajapinnan piirtämistä (ks. kuvio 11), ja kirjoitti viivan yläpuolelle

*HTTP* ja *HTTP-REST*. Rajapinnan yläpuolelle piirrettiin laatikko, johon kirjoitettiin *PWA* (progressive web application). Jaakko kertoi tämän laatikon vastaavan arkkitehtuurin näkyvää osaa, eli front-end osaa. Sitten hän jatkoi jakamalla applikaation kolmeen osaan: "Tässä mulla on nyt kolme juttua. Eka *REST* esimerkiksi *reactilla*, sitten tulee *service-kerros*, ja sitten *content* eli *database*." Jaakko piirsi tässä kohden myös rajapinnan alle tulevat osat. Ensin hän piirsi laatikon jossa lukee *PWA*, sitten laatikon jossa lukee *service* ja sitten laatikon jossa lukee *content*. *Service*-laatikon alle hän piirsi sylinterin muotoisen laatikon, jossa lukee *DB* (database). Nämä kaksi hän yhdisti assosiaatiosuhdetta muistuttavalla viivalla, sekä ympyröi katkoviivalla.

Jaakko piirsi kaksi sylinterin muotoista laatikkoa kuvion 12 laitaan. Laatikot tulisivat olemaan osa tietokantaa. Jaakko yhdisti kanta 2 nimisen laatikon assosiaatiosuhteen kaltaisella viivalla *service*-osaan. Lisäksi Jaakko piirsi ensin yhden tyhjän laatikon viivan päälle, ja sitten salmiakkia muistuttavan kuvion viivan päälle. Tyhjään laatikkoon Jaakko kirjoitti kysymysmerkin, ja kertoi sen olevan osa tietokannan ja *service*-osan informaation liikkumista.

Kysyin Jaakolta *MVC*:n käytöstä osana hänen luonnostansa samalla siirtyen haastattelun toiseen osaan. Jaakko vastasi: "Puhtaan web-sovelluksen voisin piirtää käyttäen *MVC*:tä, mutta tässä mun piirtämässä *content managerissa* on nää kolme aluetta silti." Kuviossa 12 on värikoodattu Jaakon mainitsemat *MVC*-metodin osat: sinisellä *view*-osa, punaisella *controller*-osa ja vihreällä *model*-osa. Hän nauroi myös omaa metodiensa käyttöä: "Mitä mä just piirsin ei mennyt minkään metodin mukaan. Kutsun tätä laatikko-metodiksi."

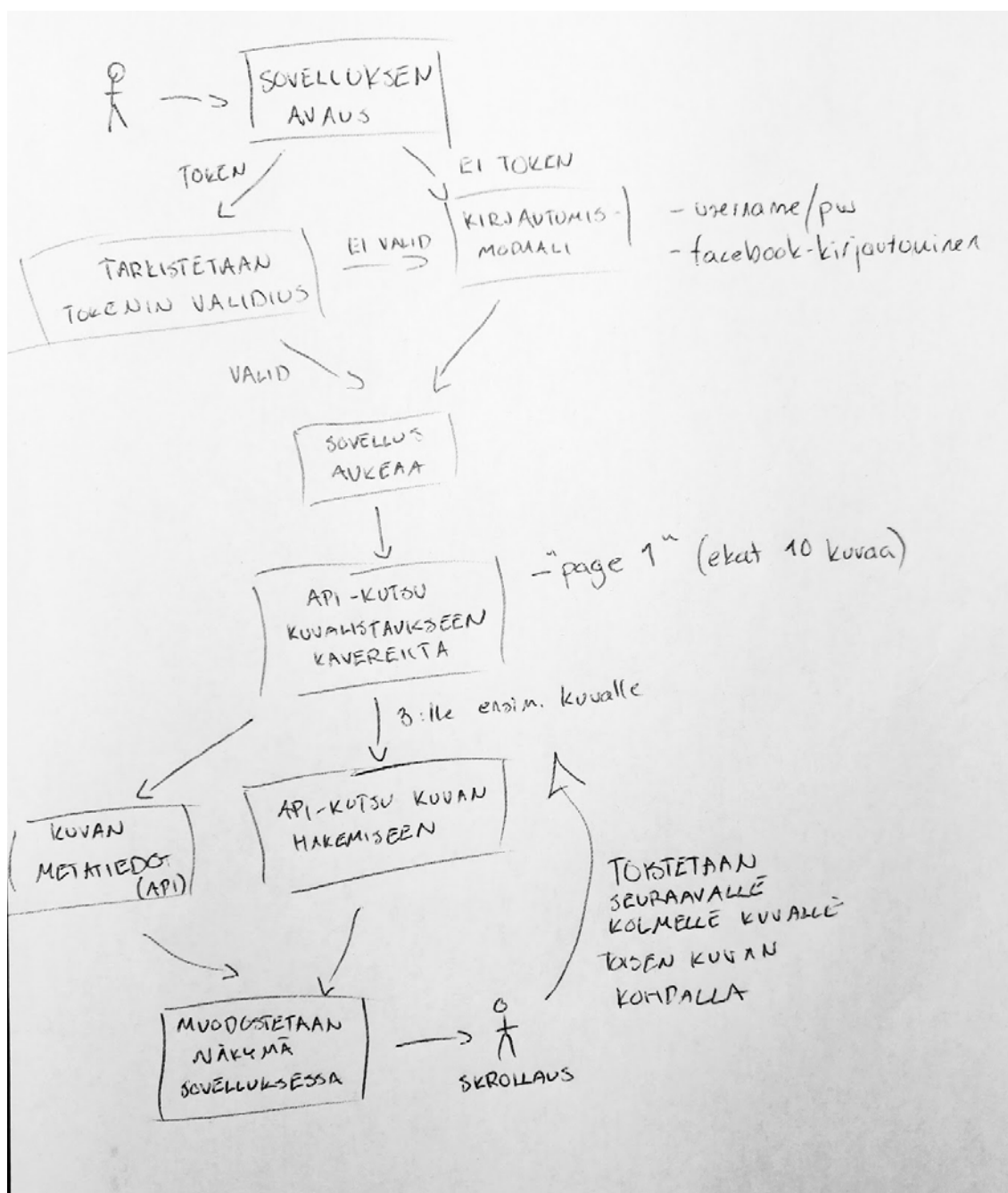
Haastattelun kolmannessa osassa Jaakko kertoi *UML*:n käytöstä seuraavasti:

On hyvä että on formaaleja tapoja (standardeja). Yksi asia mitä *UML*-kielestä tulee mieleen on käsitteet ja niiden väliset suhteet. Mulle on ihan se ja sama, mikä se käytetty työkalu on. Kaikissa on sama idea: on olemassa kaksi käsitettä ja niiden välissä on joku relaatio tai monta relaatiota.

Jaakon äänestä kuului huumori samalla hän kertoi *UML*-kielen käytöstä omassa työssään. Hän sanoi luopuneensa kuvien piirtämisestä kokonaan, ja käyttävänsä yleensä tekstiä kuvatessaan applikaation toimintaa.

#### 4.4 Sepen haastattelu

Sepe oli kolmas haastateltava. Myös Sepen haastattelu oli jaettu kolmeen osaan. Hän kertoi kuvanjakopalvelun toiminnasta rakentamalla sille arkkitehtuurin. Keskustelu jatkui rakentamisen jälkeen MVC:n ja UML:n käytön pohdiskeluun osana sovellusarkkitehtuuria.



Kuvio 13. Sepen ensimmäinen luonnos



Sepe aloitti arkkitehtuurin rakentamisen luonnoksen vasemmasta yläreunasta. Hän piirsi ensimmäisenä tikku-ukon, ja kertoi sen esittävän käyttäjää. Sepe pohti myös ääneen sitä, kuinka hän kertoisi applikaation toiminnallisuuksista selkeällä tavalla. Kuviossa 13 voidaan nähdä Sepen kirjoittaneen toiminnallisuudet laatikoihin, ja vetäneen sitten näiden välille assosiaatiosuhteen kaltaisia nuolia. Sepen luonnos on selkeä, ja sitä pystyy lukemaan ilman erillistä mallinnusta arkkitehtuurista.

Sepe pohti mitä vaiheita sovelluksen avautumiseen liittyy. Ensimmäisenä hänen mieleensä tuli käyttäjän identiteetin tarkistus eli autentikointi. Sepe piirsi ensimmäisestä laatikosta kaksi viivaa, jotka hän nimesi *token* ja *ei token*. Näillä Sepe kertoi kuvaavansa kahta vaihtoehtoa: joko käyttäjän kirjautuminen onnistuu, tai se ei onnistu. Sepe jatkoi piirtämällä rekisteröitymisen ja kirjautumisen vaiheita luonnoksen oikeaan laitaan. Kirjautumisen toiminnallisuuden Sepe esittää laatikolla, jossa lukee *kirjautumis-modaali*.

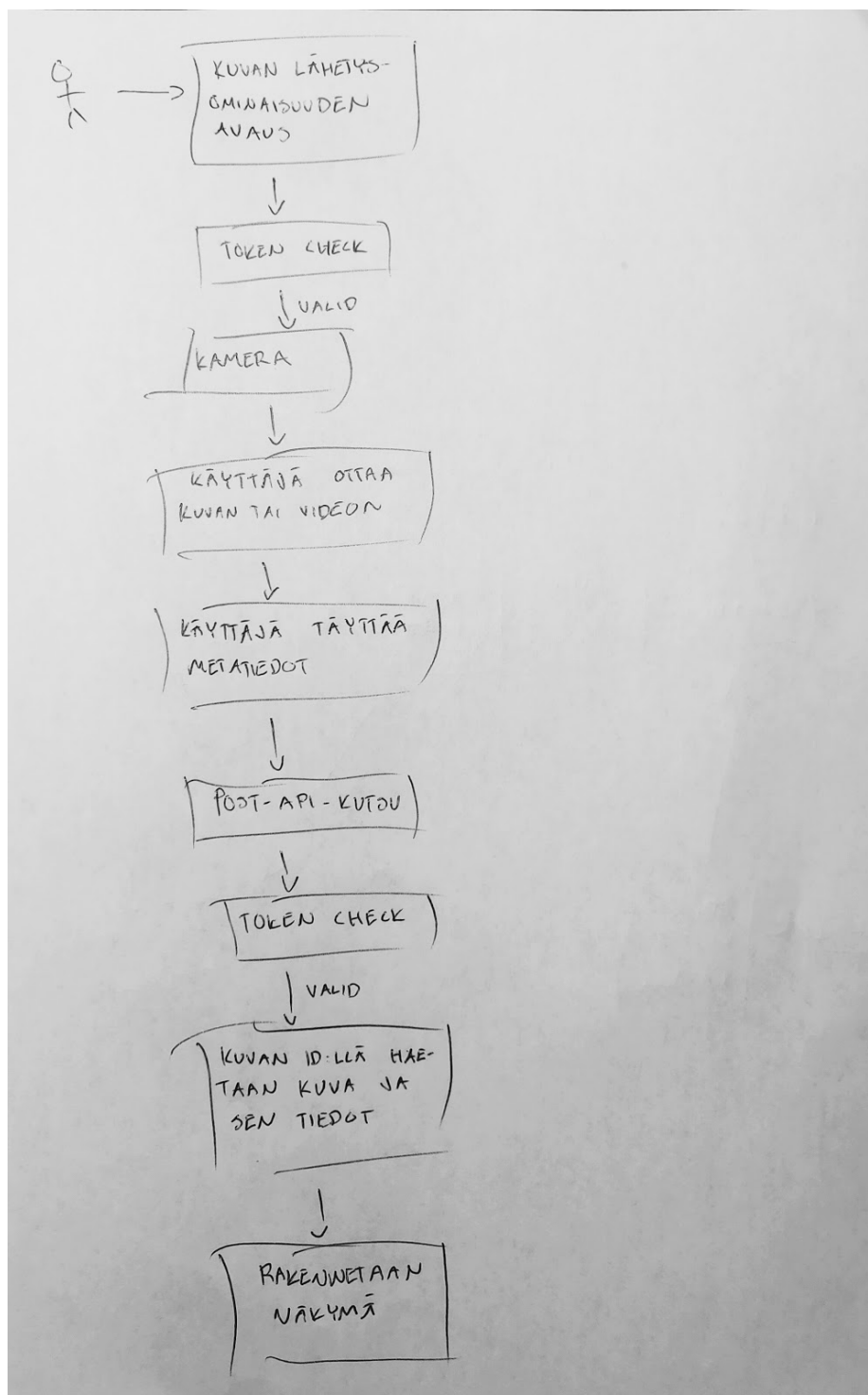
Sepe pohti arkkitehtuurin piirtämistä näin: "Ei oo helppoo eikä nopeeta tää piirtäminen. Ajattelen tän siksi aikajärjestyksessä, ja aloitan applikaation avaamisesta."

Sepe jatkaa applikaation avaamisesta sovelluksen aukeamiseen. Sovelluksen aukeamisesta hän jatkaa piirtämällä luonnoksen keskivaiheille ison laatikon, johon hän kirjoittaa *api-kutsu kuvalistaukseen kavereilta*. Sepe kertoi, että tämä laatikko kuvaa kuvanjako-palvelun *feed* ominaisuutta, missä käyttäjä voi lukea kavereidensa kuvia. Tästä laatikosta Sepe piirtää kaksi toiminnallisuutta alas ja yhdistää ne assosiaatiosuhteen kaltaisella nuolella: *kuvan metatiedot (api)* ja *api-kutsu kuvan hakemiseen*. Näiden kahden API-kutsuista kertovan toiminnallisuuden jälkeen Sepe piirtää vielä yhden laatikon, jonka hän nimeää *muodostetaan näkymä sovelluksessa*.

Sepe piirtää luonnoksen (ks. kuvio 13) lopuksi vielä yhden tikku-ukon ja kirjoittaa sen alle *skrollaus*. Tikku-ukon luota Sepe vetää vielä ison umpinaisen nuolen takaisin keskellä olevaan API-kutsu-laatikkoon, ja kirjoittaa nuolen viereen sen toiminnallisuuden: toistetaan seuraavalle kolmelle kuvalle toisen kuvan kohdalla. Sepen piirtämä nuoli muistuttaa kompositiosuhdetta. Kuten luonnoksesta voi nähdä, nuolet muodostavat kehän. Sepe kuvaa tätä *infinite scrolling* tekniikaksi, jota käytetään sosiaalisen median palveluissa paljon.



Kuvanjakopalvelulle luotiin haastattelun aikana kolme osaa: käyttäjäprofiili, kuvien lukeminen ja kuvien lähetys. Sepe sanoi tässä vaiheessa, että kuviossa 13 esitetty arkkitehtuuri kuvaa kuvan lukeminen -ominaisuutta. Tässä kohtaa Sepe nauraa ja sanoo: "Toellisuus on oikeasti hienompi kuin mun yksinkertaistus tästä applikaatiosta."

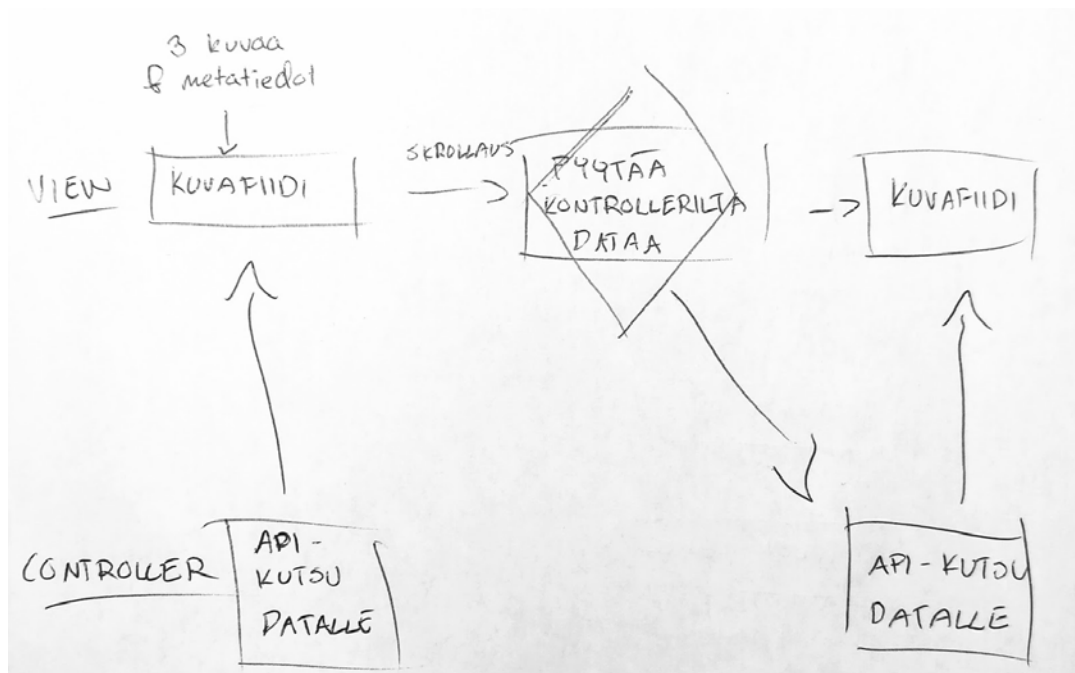


Kuvio 14. Sepen toinen luonnos

Kuviossa 14 on Sepeen luonnoksen jatkoa. Hän aloittaa piirtämällä tikku-ukon vasempaan ylälaitaan ja nauraa: "Käytän taas näitä hienoja tikku-ukkoja. Oispa olemassa joku standardi tähän." Sepe kertoo piirtävänsä nyt applikaation ominaisuuksista sen, mikä vastaa kuvan lähetyksestä. Sepe kirjoittaa laatikoihin seuraavat ominaisuudet ja vetää jokaisesta assosiaatiosuhteen kaltaisen nuolen alas kohti seuraavaa laatikkoa:

- Kuvan lähetysominaisuuden avaus
- Token check
- Kamera
- Käyttäjä ottaa kuvan tai videon
- Käyttäjä täyttää metatiedot
- POST-API-kutsu
- Token check
- Kuvan ID:llä haetaan kuva ja sen tiedot
- Rakennetaan näkymä

Sepe sanoo tämän luonnoksen (ks. kuvio 14) olevan yksinkertaistus siitä, miten kuva otetaan. Luonnoksessa on kaksi laatikkoa, joissa lukee token check, ja niiden jälkeisen viivan kohdalla lukee valid. Sepe kertoo, että se on lyhennetty mutta sama asia kuin token check edellisessä luonnoksessa (ks. kuvio 13).



Kuvio 15. Sepeen kolmas luonnos

Tässä vaiheessa haastattelua keskustelu siirtyi toiseen ja kolmanteen osaan MVC:n ja UML:n käytöstä osana sovellusarkkitehtuuria. Sepe piirsi vielä selkeyden vuoksi yhden luonnoksen kuvanjakopalvelun arkkitehtuurista MVC-metodin avulla.

Kuviossa 15 on Sepen luonnos applikaation arkkitehtuurista MVC-metodin mukaan. Sepe kertoo MVC-metodin käytöstä seuraavasti:

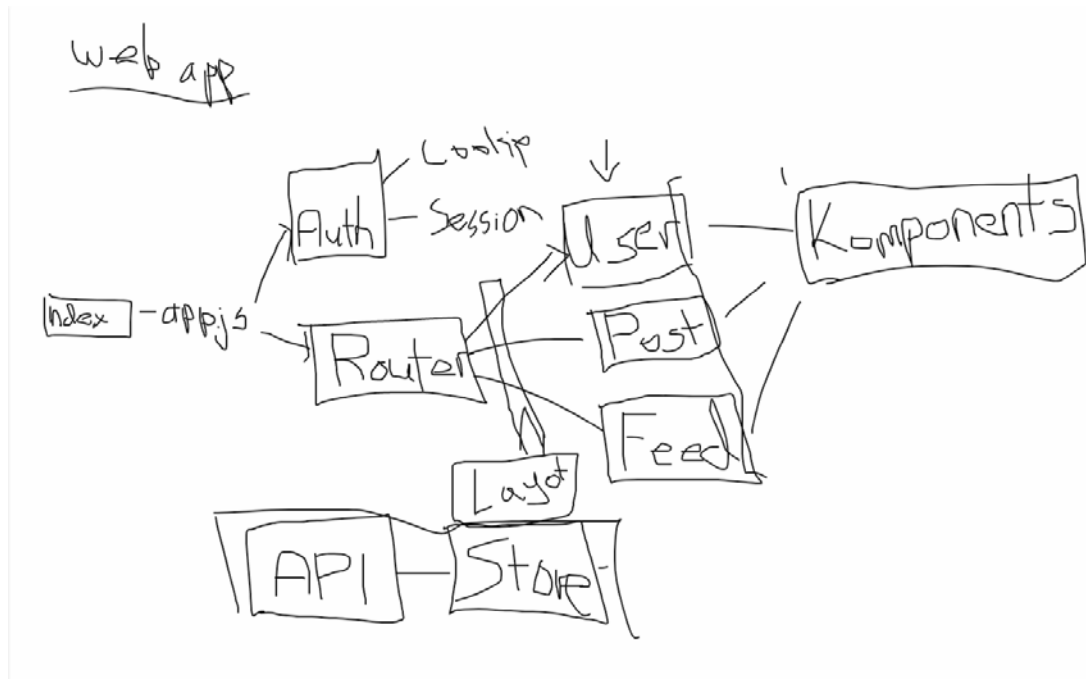
Teen tän MVC-mallin myös aikajärjestyksessä. Tiedän, että MVC:ssä irrotan käyttäjän tekemiset pois view-osasta. Mutta en tiä mihin oikeastaan ne erotan. Varmaan controller-osaan. Sitä en nyt ihan muista miten mallinnan sen, että controller pyytää jotain lisää tolta viewiltä.

Sepe aloitti MVC-luonnoksen piirtämisen view-osasta. Hän kirjoitti laatikkoon *kuvafiidi*, ja kertoi sen olevan sama asia kuin ensimmäisessä luonnoksessa (ks. kuvio 13). Siitä Sepe veti assosiaatiosuhteen kaltaisen viivan eteenpäin. Viivan yläpuolelle hän kirjoitti *skrollaus*. Sepe jatkoi piirtämällä seuraavat laatikot osaksi arkkitehtuuria: *pyytää controllerilta dataa*, *kuvafiidi* ja *API-kutsu datalle*. Viimeisen näistä laatikoista hän piirsi osaksi controlleria. Lopuksi hän piirsi arkkitehtuurin controller-osan vasempaan laitaan laatikon *API-kutsu datalle*, ja veti siitä assosiaatiosuhteen kaltaisen viivan takaisin view-osaan. Model-osan arkkitehtuurista Sepe jätti piirtämättä kokonaan. Kysyttäessä hän vastasi, että ei muista täysin controller-osan ja model-osan eroa.

Sepe kertoi käyttävänsä UML-mallinnuskieltä töissä jatkuvasti: "Siinä mielessä käytän jotain UML-kielen tapaista, mutta ei olla määritelty että se olis just UML-kieli. Käytetään neliöitä ja salmiakkibokseja."

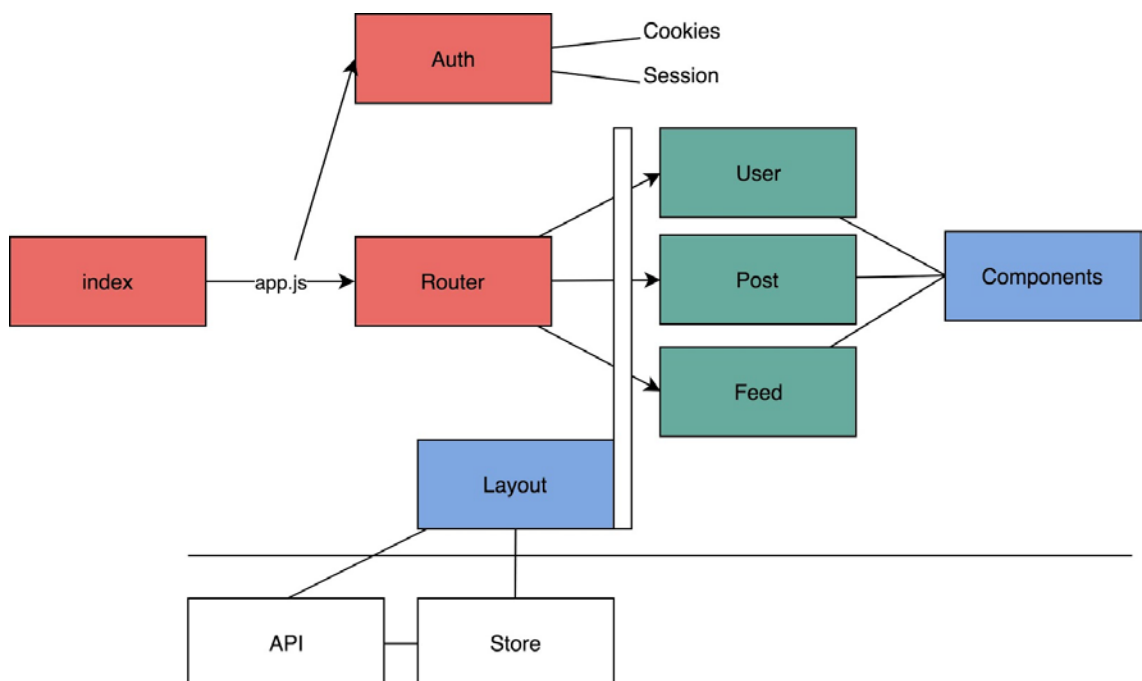
#### 4.5 Juhan haastattelu

Juha rakensi sovellusarkkitehtuurin etäpiirtoalustan avulla, ja kertoi MVC:n ja UML:n käytöstä videopuhelun välityksellä.



Kuvio 16. Juhan luonnos

Juha kertoi käyttävänsä arkkitehtuuria töissään. Arkkitehtuurin perusteet hän oppi koulussa. Aktiivisemman otteen arkkitehtuurien käytölle hän kertoi saaneensa pari vuotta aikaisemmin työskennellessään ohjelmistokehittäjänä startupissa.



Kuvio 17. Juhan piirtämä arkkitehtuuri

Kun haastattelun ensimmäinen osa alkoi, Juha mietti ääneen, mitä arkkitehtuurimetodeja hän voisi käyttää kuvanjakopalvelun arkkitehtuurin rakentamiseen. Juhalle muistui mieleen seuraavat metodit: “MVC, monoliittinen-arkkitehtuuri ja microservice-arkkitehtuuri.” Hän sanoi rakentavansa applikaation MVC-metodin avulla ja kertoi sen käytöstä seuraavasti:

MVC on sitä, että kuinka paljon eri osat puhuu toisilleen tai tietää toisistaan. Siinä on myös eri servicet kaikille asioille, eli ne tietää mahdollisimman vähän toisistaan, mutta tekee yhden asian tosi hyvin. Esimerkiksi front-end komponenteista nappi. Nappia painamalla se tekee yhen jutun hyvin, mutta ei mitään muuta.

Juha jatkoi piirtämällä ensimmäisen laatikon ja nimesi sen *index*. Laatikon viereen hän kirjoitti *app.js* tekstin. Juha kertoi nyt miettivänsä arkkitehtuuria react-kirjaston kautta. Toiseen laatikkoon Juha kirjoitti *router* ja veti assosiaatiosuhdetta muistuttavan viivan näiden välille niin, että *app.js* jäi sen väliin. Kysyttäessä tästä tekstistä Juha kertoi sen tarkoittavan applikaation ydintä.

Kuviossa 17 on värikoodattu MVC-metodin mukaisesti Juhan piirtämät komponentit. Punaisella controller-osa, vihreällä model-osa ja sinisellä view-osa. Haastateltava jatkoi piirtämällä applikaation toiminnallisuudet: *user*, *post* ja *feed*, ja veti assosiaatiota muistuttavan viivan jokaiseen näistä. Routerin toiminnasta Juha sanoi näin: “Applikaation router katsoo mitä käyttäjä on just nyt tekemässä ja ohjaa sen sitten oikeaan paikkaan.” Vihreiden model-osien jälkeen Juha piirsi laatikon, kirjoitti sen sisälle *components*, ja sanoi sen sisältävän kaikki visuaaliset elementit jotka käyttäjälle näytetään kulloisessakin ruudussa.

Juha piirsi myös ison paksun viivan joka päättyy siniseen layout-osaan. Juha kertoi tämän olevan se palikka applikaatiosta, joka pohjan sille mitä käyttäjä näkee jokaisella ruudulla. Sinisestä layout-osasta Juha veti vielä kaksi assosiaatiosuhteen kaltaista viivaa *API* ja *store* nimisiin laatikoihin.

Haastattelun siirtyessä toiseen osaan Juha kuvaa MVC:tä heti seuraavasti: “En tykkää MVC:n käytöstä. Se on vanhanaikainen tapa tehdä asioita. Etenkin jos puhutaan frontista.” Hän jakoi myös ajatuksia omasta historiasta ohjelmistokehittäjänä ja mainitsi yleensä käyttävänsä microservice-arkkitehtuuria. Haastattelun ensimmäiseen vaiheeseen Juha kuitenkin käytti MVC:tä, koska se on kaikista arkkitehtuurimetodeista hänelle tutuin.

Juha kertoi myös käyttävänsä UML-mallinnuskieltä osana arkkitehtuurin rakentamista, mutta suuressa ohjelmistoalan organisaatiossa jossa hän työskentelee, UML-kielestä ei puhuta ainakaan sen oikealla nimellä. Juha jatkoi vielä ja tarkensi tätä sanomalla, että hänen työpaikallaan UML-kieltä käytetään lähtökohtaisesti tietokantoihin liittyvä datan käsittelyn suunnittelemisessa.

## 5 Tutkimustulokset ja analysointi

### 5.1 MVC-arkkitehtuurin osien rooli nähdään epäselvänä

Ensimmäinen teema, joka nousee aineistosta on se, että MVC-arkkitehtuurin osien rooli nähdään epäselvänä, vaikka metodin käyttö muuten on sujuvaa.

MVC:n osat ovat model-osa, view-osa ja controller-osa (ks. luku 2). Jokaisessa haastattelussa sovellusarkkitehtuuri koettiin helposti lähestyttävänä teemana ja tuttuna metodina. Kuitenkin käytännössä, kun asioita luonnosteltiin paperille, moni joutui miettimään mihin kohtaan MVC-metodia tietyn toiminnon sijoittaisi.

Vili määritteli MVC:n osien roolit oikein, mutta controller-osa jäi yksinkertaisemmaksi suhteessa muihin osiin (ks. kuvio 9). Vili piirsi ensin controller-osaksi pelkästään yhden laatikon ja tarkensi piirtämällä muutaman laatikon lisää myöhemmin miettiessään, miten model-osa keskustele controller-osan kanssa.

Sepen luonnoksessa (ks. kuvio 15) model-osa sijoitettiin controller-osaan. Vaikka haastateltavalle MVC oli tuttu väline, hän päätti yhdistää model-osan ja controller-osan. Sepen luonnoksessa olisi riittänyt se, että nykyinen controller-osa siirretään model-osaan, ja controller-osaan luodaan MVC-metodin mukaan toimintoja jotka vastaavat käyttäjän suorista ja välillisistä interaktiosta luvussa 2 esitetyn teorian mukaan.

Jaakon luonnoksessa (ks. kuvio 12) MVC:n osat jaettiin oikein: REST-rajapinta (view-osa), Service-kerros (controller-osa) ja Content eli database (model-osa).

Juha määrittelee Controllerin roolin seuraavasti: *“Router (controller-osa) välittää informaation komponenteille (view-osa). Tässä mun applikaatiossa controller-osassa on API, ja jokaiselle API-kutsulle on oma Controller”* (ks. kuvio 17). Controller-osan ja view-osan

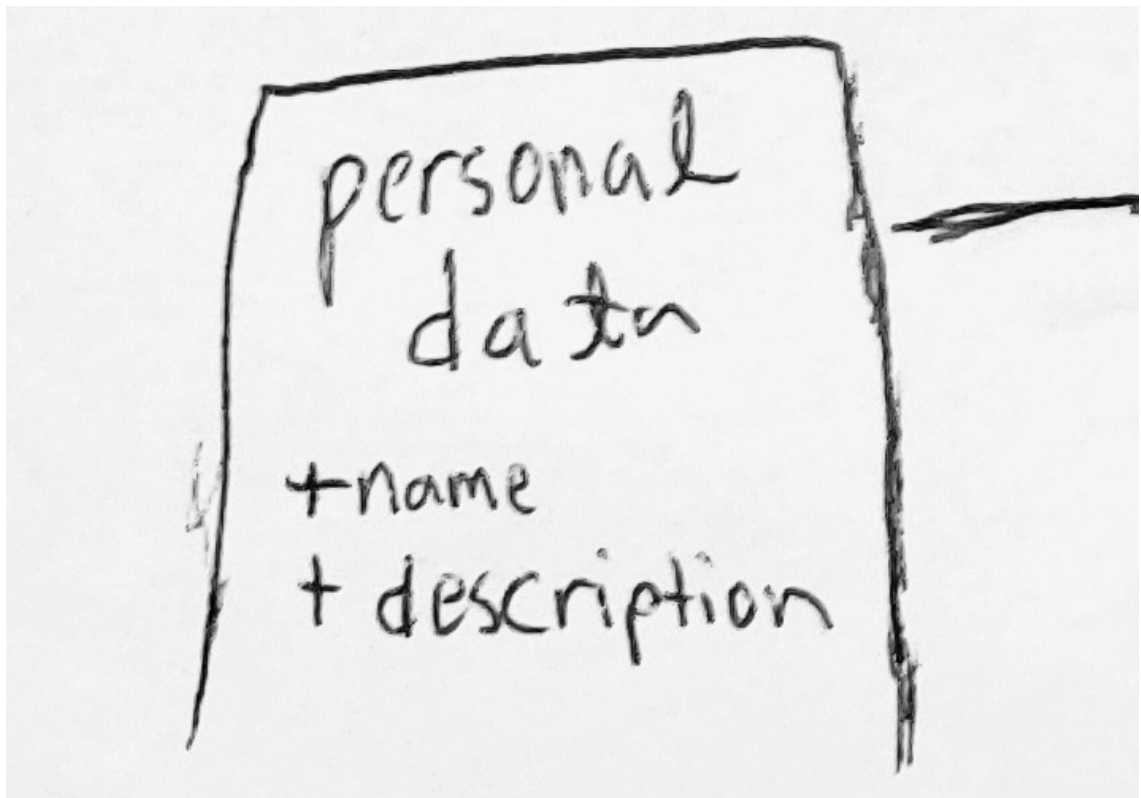
roolituksessa ei ollut selkeää rajaa, vaan osa komponenteista on molemmissa osissa kiinni.

MVC-arkkitehtuurissa kaikilla osilla on täsmällinen rooli (ks. luku 2). Vilin, Sepen ja Juhan mallinuksissa esimerkiksi controller-osaan lisättiin model-osan asioita. MVC:n osien roolien sekoittuminen ei mielestäni johdu haastateltavien pätevyyden puutteesta, vaan siitä, että nykyaikaisten applikaatioiden monimutkainen rakenne ja vaatimukset ylittävät MVC-metodin alkuperäisesti asetetut rajat. Lisäksi mallin lähestymistavassa ja soveltamisessa heijastuvat kehittäjien henkilökohtaisten ajattelutapojen erot, erilaiset vahvuudet ja työnkuva.

## 5.2 UML-mallinnuskieltä käytetään soveltavasti

Toinen aineistosta noussut teema on se, että UML-mallinnuskieltä käytetään soveltavasti osana arkkitehtuurin rakentamista. On selvää, että UML-kielen pieniä sääntöjä ei haastattelutilanteessa voi käyttää ajan puutteen vuoksi. Tätä varten haastattelun kolmas osa käsitteli UML:n todellista käyttöä. Haastattelihoita pyydettiin näyttämään rakennetun kuvanjakopalvelun arkkitehtuurista niitä kohtia, joissa on käyttänyt UML-kieltä, tai haluaisi käyttää. Sepen ensimmäinen vastaus kysyttäessä UML:n käytöstä kuvaa hyvin toista teemaa: *“Todellisuus on hienompi kuin mun yksinkertaistus tästä applikaatiosta.”*

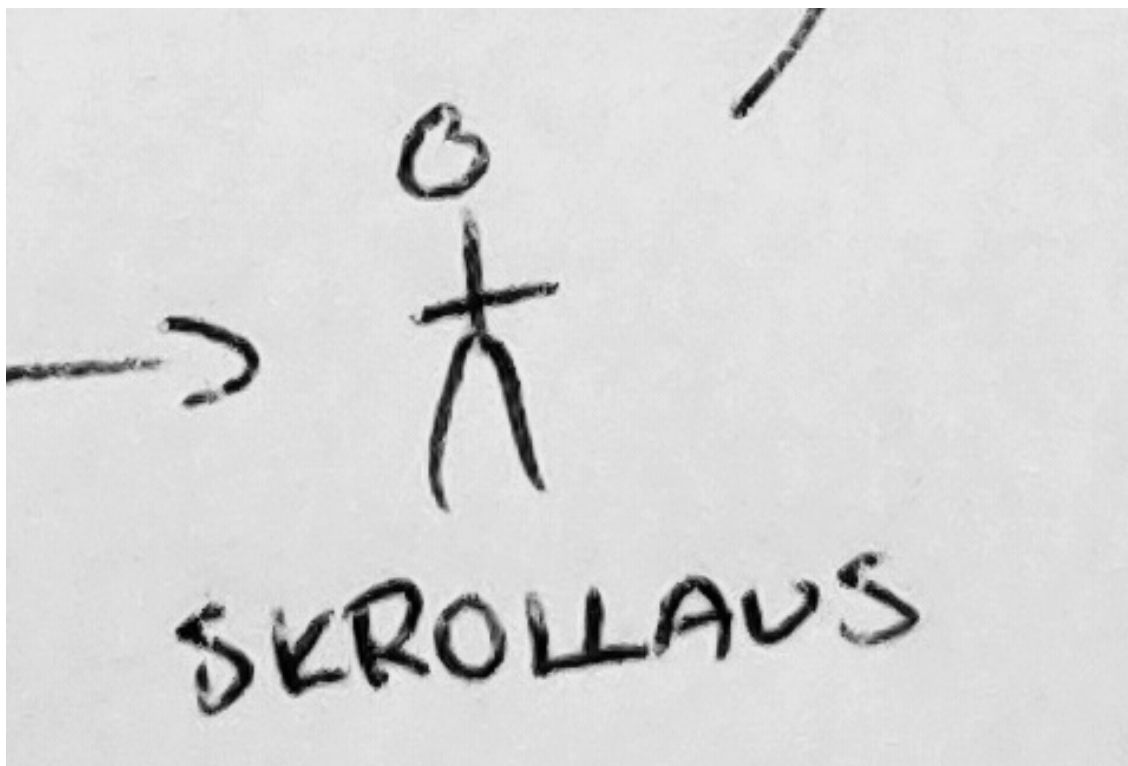
Vili käytti UML-kielen standardeja rakentaessaan arkkitehtuuria kuvanjakopalvelulle. Esimerkiksi hänen piirtämänsä luokkakaavio (ks. kuvio 18) käyttää luvussa 2 esitettyjä ohjeita. Samasta kuvioista voidaan myös huomata, että luokkakaaviossa ei ole pohjaa. Vili kertoo, että jättää laatikoista pohjan piirtämättä, jotta attribuuttien ja metodien lisääminen on rakentamisen jokaisessa vaiheessa helpompaa ja nopeampaa.



Kuvio 18. Vilin piirtämä luokkakaavio

Sepe piirsi kaikki käyttäjän toimintoja kuvaavat asiat arkkitehtuurissa tikku-ukoilla (ks. kuvio 19). Tikku-ukon avulla on Sepen mukaan helpompi huomata käyttäjän keskeiset toiminnot arkkitehtuurista heti ensimmäisellä silmäyksellä. Kuten luvussa 1 kerrotaan, arkkitehtuurin yksi tavoitteista on mahdollistaa tehokas työskentely suunnittelijoille sovelluksen kaikissa vaiheissa. Sepen piirtämässä arkkitehtuurissa (ks. kuvio 13) nousee esille luovia ja kekseliäitä tapoja käyttää UML-standardia.





Kuvio 19. Sepen piirtämä tikku-ukko

Jaakko kertoi haastattelun aikana, että hän käyttää UML-kielen kaltaista metodia rakentaessaan sovelluksia, mutta kutsuu tätä omaksi laatikkometodikseen. Juha taas sanoi, että ”UML on mulle sitä, että laitetaan kaikki classeihin ja piiretään sitten seinälle.” Jaakon ja Juhan asennetta UML-kieltä kohtaan voidaan kuvata rehellisenä ja terävänä tosiasiana UML:n nykyisestä tilasta: mallinnuskieltä käytetään soveltavasti yli sille asetettujen rajojen. Vaikka UML-mallinnuskieli on alan standardi, sen käyttö on enimmäkseen luovaa ja sovellettua. Haastateltavien näkökulmasta UML-mallinnuskielen sovellettu käyttö johtaa tehokkaampaan arkkitehtuurin rakentamiseen.

### 5.3 MVC-arkkitehtuuria ei nähdä kestäväenä metodina

Kolmas esille noussut teema on se, että MVC-arkkitehtuuria ei nähdä kestäväenä metodina.

Jaakon ja Juhan haastatteluissa nousi esille MVC-arkkitehtuurin vanhanaikaisuus. Juha kertookin MVC:n käytöstä haastattelun toisen osan kysymyksen jälkeen seuraavasti: *”En tykkää MVC(-metodista) hirveästi. Vanhanaikainen tapa tehdä asioita.”* Silti Juha korosti MVC:n olevan kärjessä kaikista metodeista joita hän käyttää, koska se on tutuin työkalu.

Sepe kertoi osana haastattelun toista osaa, että MVC-metodin käyttö on haastavaa nykyaikaisten applikaatioiden suunnitteluvaiheessa. Tämä näkyy myös kuviossa 15 kun Sepe on yhdistänyt controller-osan ja model-osan toiminnot keskenään.

Jaakko nosti haastattelun aikana esiin arkkitehtuurien tuen loppumisen. Tällä hän tarkoittaa sitä, että jos arkkitehtuurin joku osa rakennetaan kirjaston tai muun palikan mukaan, on mahdollista, että sitä ei enää käytetä muutaman vuoden päästä. Jaakko kertoo asiasta näin:

Pitää valita arkkitehtuuri niin, että sitä voi käyttää mahdollisimman pitkään osana sovelluksen kehitystä. Olen käyttänyt aikanaan joitain arkkitehtuureja, jotka on sittemmin ovat kuollut pois. Esimerkiksi Yahoo UI Frameworkit ja arkkitehtuurit menee ja tulee.

MVC-arkkitehtuurimalli itsessään ei voi vanhentua. Mutta tällä metodilla luodut arkkitehtuurin osat voivat. Arkkitehtuurit ovat ikuisia, tai kuten Juha asian ilmaisi: *”Isossa firmassa missä mä oon töissä kaikkein eniten aikaa menee siihen, että mietitään miten se data saadaan databaseen. Sitä kun ei voi muuttaa jälkikäteen.”*

Haastateltavat olivat yhtä mieltä siitä, että MVC-arkkitehtuurin käyttö nykyaikaisten applikaatioiden suunnittelussa ei välttämättä ole huomispäivän trendi. Vaikka asenne metodia kohtaan onkin tämä, Jaakko, Juha ja Sepe käyttävät sujuvasti MVC:tä kuvanjakopalvelun arkkitehtuurin rakentamiseen.

#### 5.4 Kehittäjän rooli organisaatiossa vaikuttaa MVC-arkkitehtuurin käyttöön

Neljäs aineistosta esiin nouseva teema on se, että kehittäjän rooli organisaatiossa vaikuttaa MVC-arkkitehtuurin käyttöön. Haastattelujen perusteella arkkitehtuurin suunnittelujärjestykseen vaikuttaa se, toimitko front-end- vai back-end-ohjelmoijana organisaatiossa.

Vili pohti osana haastattelua sitä, että hän on huomannut front-end ohjelmoijien aloittavan MVC-arkkitehtuurin rakentamisen view-osasta ensin, eli sovelluksen visuaalisesta puolesta. Vili itse aloitti arkkitehtuurin rakentamisen model-osasta, koska se pitää sisälleen sovelluksen toimintalogiikan, ja on täten helpommin hahmotettavissa. Jos rakentamisen aloittaisi view-osasta, sovelluksen konkreettisten toimintojen mallintaminen olisi helpompaa.

Krasnerin ja Popen (1988, 26-27) mukaan arkkitehtuurin rakentaminen kannattaa aloittaa model-osasta, koska se sisältää sovelluksen keskeiset toiminnot.

Jaakko aloitti arkkitehtuurin suunnittelemisen rakentamalla yhden tavan siirtää kuvia tietokannasta käyttäjän sovellukseen (ks. kuvio 12). Kuviossa view-osa on sininen PWA-laatikko, ja se on REST-rajapinnan takana vastaanottamassa kuvia. Jaakko työskentelee järjestelmäasiantuntijana, eli back-end puolella. Hän aloitti arkkitehtuurin rakentamisen model-osasta.

Sepe aloitti arkkitehtuurin rakentamisen view-osasta (ks. kuvio 15), ja piirsi tähän osaan osan sovelluksen toiminnallisuuksista. Hänelle sovelluksen rakentaminen oli luontevampaa aloittaa view-osasta, koska toiminnallisuuksien hahmottaminen visuaalisen view-osan kautta on helpompaa (ks. kuvio 13). Sepe työskentelee ohjelmistokehittäjänä ja käyttää react-pohjaista front-end kirjastoa.

Haastattelujen perusteella MVC:n suunnittelujärjestyksessä on suora korrelaatio siihen, työskenteleekö front-end- vai back-end-ohjelmoijana.

## 6 Johtopäätökset

Opinnäytetyössä tutkin sovellusarkkitehtuurien käyttöä nykypäivänä. Pyrin työssäni vastaamaan tutkimuskysymykseen: ”Millä tavoin sovellusarkkitehtuuristandardeja käytetään nykyaikaisten sovellusten suunnittelussa?” Tutkimuksen perusteella voidaan todeta, että standardinomainen metodien käyttö ei ole nykypäivää.

Nykyaikainen metodien ja standardien käyttö on luovaa ja sovellettua. Sepe kertoo rakentaessaan arkkitehtuuria todellisen standardien käytön olevan kaukana hänen luonnoksestaan (ks. kuvio 13). Silti Sepen rakentama arkkitehtuuri on selkeä kokonaisuus ja sillä päästiin samaan lopputulokseen kuin luvussa 4.1 esitettyyn malliin eli ymmärrettävään arkkitehtuuriin. Aineistosta esiin nousseet teemat on kerätty yhteen taulukossa 2.

Taulukko 2. Yhteenveto aineistosta esiin nousseista teemoista

Teema	Yhteenveto
MVC-arkkitehtuurin osien rooli nähdään epäselvänä	MVC:n osien roolien sekoittuminen johtuu siitä, että nykyaikaisten applikaatioiden monimutkainen rakenne ja vaatimukset ylittävät MVC-metodin alkuperäisesti asetetut rajat.

UML-mallinnuskieltä käytetään soveltavasti	Haastateltavien näkökulmasta UML-mallinnuskielen sovellettu käyttö johtaa tehokkaampaan arkkitehtuurin rakentamiseen.
MVC-arkkitehtuuria ei nähdä kestäväenä metodina	Haastateltavat olivat yhtä mieltä siitä, että MVC-arkkitehtuurin käyttö nykyaikaisten applikaatioiden suunnittelussa ei välttämättä ole huomispäivän trendi.
Kehittäjän rooli organisaatiossa vaikuttaa MVC-arkkitehtuurin käyttöön	Haastattelujen perusteella MVC:n suunnittelujärjestyksessä on suora korrelaatio siihen, työskenteleekö front-end- vai back-end-ohjelmoijana.

Esitin johdannossa kaksi hypoteesia perustaen ne jo julkaistuun tutkimukseen ja alan trendeihin. Luvussa 5 esitetyistä teemoista voi vetää sen johtopäätöksen, että ensimmäinen hypoteesi MVC-arkkitehtuurin käytöstä alan standardina pitää vain osittain paikkansa. MVC-arkkitehtuuria ei nähdä kestäväenä metodina, mutta sitä käytetään silti soveltavasti nykyaikaisten sovellusten suunnitteluun.

Toinen hypoteesi UML:n käytöstä vahvistui tutkimuksen jälkeen. Haastattelujen aikana nousi esiin luovia ja ketteriä tapoja ilmaista arkkitehtuurin osia ja muotoja. Jaakko kertoi käyttävänsä keksimäänsä laatikkometodia puhuttaessa mallinnuskielistä. Vaikka UML on alan standardi, sitä käytetään vapaasti osana arkkitehtuurin rakentamista. Voidaan todeta, että käytetty aika suhteessa tulokseen näkyi haasteena UML:n todelliselle käytölle.

Standardit ovat vahva perusta sovelluksen suunnittelulle, mutta se ei tarkoita sitä, että niitä kannattaa käyttää orjallisesti. Kuten Baltes ja Diehl (2013) sanovat: suurin osa arkkitehtuuriluonnoksista on olemassa vain viikon verran. Monet tämän päivän sovelluskehityshankkeet edellyttävät nopeita ja ketteriä metodeja, ja tutkimukseni mukaan standardit vaikuttavat liian kankealta tähän.

Osana haastatteluja rakennettiin yksinkertainen arkkitehtuuri kuvanjakopalvelulle. Arkkitehtuuri rakennettiin nopeasti ja kevyesti. Voidaan todeta, että standardien soveltava käyttö osana rakennettua arkkitehtuuria johtuu ainakin osittain suunniteltujen applikaatioiden pienestä koosta. Arkkitehtuurista on eniten hyötyä niissä organisaatioissa, joissa hallinnoidaan isoja sovellus- ja järjestelmäkokonaisuuksia (O'Regan 2014, 45). Sovellukset, joiden hallinta ja kehittäminen on jaettu useamman eri osaston tai ryhmän välillä, tarvitsevat edelleen arkkitehtuuria ja standardien vahvaa käyttöä toiminnan hahmottamiseen ohjelmistokehityksen eri vaiheiden aikana.

Arkkitehtuuria pidetään usein haasteena, joka syö sovellukselle budjetoitua työaikaa. Tämän tutkimuksen perusteella voidaan todeta, että nykyaikaiset applikaatiot ja sovellukset rakennetaan käyttäen ketteriä ja sovellettuja omiin tarpeisiin sopivia metodeja. Arkkitehtuurimetodien ja standardien luovaa ja sovellettua käyttöä on silti koordinoitava: liika vapaus johtaa väärinkäsityksiin ja kokonaisuuden hajoamiseen (Babar & Abrahamson 2008, 242-243). Alusta asti mietitty, suunniteltu ja toteutettu arkkitehtuuri on hyödyksi suunnittelijoille ja ohjelmoijille sen kaikissa vaiheissa.

Tutkimuksen pohjalta voidaan ehdottaa uudenlaista tapaa käyttää standardeja. Sovelluskehityksen vaiheita voitaisiin tehostetaan huomattavasti antamalla suunnittelijoille vapauksia käyttää omaa luovuuttaan osana arkkitehtuurin rakentamista. Esimerkiksi kevyiden sovellusten kohdalla standardien totisesta käytöstä voitaisiin siirtyä standardien sovellettuun käyttöön siihen käytettävän ajan ja osaamistason puitteissa.

## 7 Lopuksi

Kaikki arkkitehtuuri on suunnittelua mutta kaikki suunnittelu ei ole arkkitehtuuria. Kaiken kaikkiaan arkkitehtuuri sisältää merkittäviä tekijöitä, jotka vaikuttavat sovelluksen toimintaan ja loppupeleissä myös kustannuksiin. (Booch 2006, Buschmannin, Henneyn & Schmidtin 2007, 214 mukaan.) Sovelluksen suunnitteluun on olemassa useita standardeja, mutta käytämmekö niitä enää huomenna?

Opinnäytetyön tavoitteena ei ole muuttaa nykyisiä arkkitehtuurimetodeja eikä argumentoida standardien käyttöä vastaan. Sen sijaan tämän työn pohjalta saatua tietoa nykyaikaisista arkkitehtuureista voi hyödyntää tulevien sovellusten suunnitteluprosessin ja sovellettujen arkkitehtuurimetodien kehittämisessä, esimerkiksi avaamalla organisaation sisäisiä keskusteluja metodien luovasta ja sovelletusta käytöstä.

Tekemieni haastattelujen pohjalta on mielenkiintoista nähdä, miten MVC:n ja UML:n käy seuraavan vuoden ja vuosikymmenen aikana. Yksi mahdollinen jatkotutkimuksen suunta voisi olla haastatella laajemmalla skaalalla ohjelmistoalan ammattilaisia ja kategorisoida tuloksia organisaation kokoon nähden.

Jos voisin tehdä tutkimuksessa jotain toisin, lisäisin haastatteluiden lukumäärää. Tutkimustulokset ovat häilyviä neljän haastattelun takia. Työssä ei esimerkiksi selvinnyt tarpeeksi konkreettisia tapoja käyttää UML-kieltä soveltavasti, jotta siitä olisi voinut vetää

konkreettisia johtopäätöksiä. Lisäksi monipuolisuuden kannalta olisi ollut hyvä tutkia muidenkin kuin MVC-metodin käyttöä.

Hyvin suunniteltu ja dokumentoitu sovellusarkkitehtuuri antaa mahdollisuuden aikaiseen kommunikointiin haasteista ja ongelmista (Bass ym. 2003, 37). Arkkitehtuurin hyötyä ei voi kiistää. Omien suunnittelumetodien luova liittäminen ohjelmistoalan standardeihin antaa mahdollisuuden arkkitehtuurin onnistumiselle sen kaikissa vaiheissa. Uskon että tämä johtaa kestävämpään arkkitehtuuriin ja ennen kaikkea julkaistuihin sovelluksiin.

Tämä tutkimus on avannut kiinnostavia ovia sovelluskehityksen vaiheiden kokonaisvaltaiseen hahmottamiseen ja ymmärtämiseen. Toivon, että keskustelu käytetyistä arkkitehtuurimetodeista ja ohjelmistoalan standardeista jatkuu yhtä mielekkäänä myös tulevaisuudessa.

## Lähteet

Abrahamsson, Pekka; Salo, Outi; Ronkainen, Jussi; Warsta, Juhani 2002. Agile Software Development Methods: Review and Analysis. Espoo: VTT.

Babar, Muhammad; Abrahamsson, Pekka 2008. Architecture-Centric Methods and Agile Approaches. Berlin: Springer.

Baltes, Sebastian; Diehl Stephan 2013. Sketches and Diagrams in Practice — Supplementary Material. Luettavissa osoitteessa <<http://doi.org/10.5281/zenodo.818277>> (luettu 01.11.2018).

Bass, Len; Clements, Paul; Kazman, Rick 2003. Software architecture in practice. Addison-Wesley.

Brecht, Tim 2006. Evaluating network processing efficiency with processor partitioning and asynchronous I/Os. HP. Luettavissa osoitteessa <<http://www.hpl.hp.com/techreports/2005/HPL-2005-211R1.pdf>> (luettu 07.11.2018).

Buschmann, Frank; Henney, Kevlin; Schmidt, Douglas 2007. Pattern-Oriented Software Architecture, On Patterns and Pattern Languages.

Fowler, Martin 2004. A Brief Guide to the Standard Object Modeling Language Third Edition. Addison-Wesley.

Google Trends n.d. Layered architecture, Event-driven architecture, Microkernel architecture, Microservices architecture, Space-based architecture by time, location and popularity on Google Trends. Luettavissa osoitteessa <<https://trends.google.com/trends/explore?date=2005-01-01%202015-01-01&q=Layered%20architecture,Event-driven%20architecture,Microkernel%20architecture,Microservices%20architecture,Space-based%20architecture>> (luettu 01.11.2018). Google.

Hirsjärvi, Sirkka; Hurme, Helena 2000. Tutkimushaastattelu: Teemahaastattelun teoria ja käytäntö. Helsinki: Yliopistopaino.

Hirsjärvi, Sirkka; Remes, Pirkko & Sajavaara, Paula 2009. Tutki ja Kirjoita, 15 uudistettu painos. Tammi.

Kananen, Jorma 2008. Kvali - Kvalitaatiivisen tutkimuksen teoria ja käytänteet. Jyväskylän Yliopistopaino.

Krasner, Glenn; Pope, Stephen 1988. A Cookbook for Using View-Controller User the ModelInterface Paradigm in Smalltalk-80. Luettavissa osoitteessa <<https://www.lri.fr/~mbl/ENS/FONDIHM/2013/papers/Krasner-JOOP88.pdf>> (luettu 30.10.2018).

Maciaszek, Leszek 2007. Requirements Analysis and System Design. Pearson.

Moilanen, Pentti; Rähä, Pekka 2010. Merkitysrakenteiden tulkinta, osana julkaisua Aaltola, Juhani; Valli, Raine 2010, Ikkunoita tutkimusmetodeihin: 2, Näkökulmia aloittavalle tutkijalle tutkimuksen teoreettisiin lähtökohtiin ja analyysimenetelmiin, 3. uud. ja täyd. p. PS-Kustannus.

OMG 2005. About the unified modeling language specification version 2.0. Luettavissa osoitteessa <<https://www.omg.org/spec/UML/2.0/>> (luettu 30.10.2018).

OMG 2017. About the unified modeling language specification version 2.5.1. Luettavissa osoitteessa <<https://www.omg.org/spec/UML/2.5.1/>> (luettu 30.10.2018).

O'Regan, Gerard 2014. Introduction to Software Quality. Berlin: Springer.

Pender, Tom 2003. UML Bible. John Wiley & Sons.

Perry, Dewayne; Wolf, Alexander 1992. Software engineering notes. New York: ACM.

Reenskaug, Trygve 2007. The original MVC reports. Luettavissa osoitteessa <[http://folk.uio.no/trygver/2007/MVC\\_Originals.pdf](http://folk.uio.no/trygver/2007/MVC_Originals.pdf)> (luettu 30.10.2018).

Richards, Mark 2015. Software Architecture Patterns. California: O'Reilly Media.

Robie, Jonathan 1998, Document Object Model (DOM) Level 1 Specification: What is the Document Object Model?. Luettavissa osoitteessa <<https://www.w3.org/TR/DOM-Level-1/introduction.html>> (luettu 07.11.2018).

Ruparelia, Nayan 2010. ACM SIGSOFT Software Engineering Notes. New York: ACM.

Sommerville, Martin; Scott, Kendall 1999. Software engineering 9th edition. Pearson.

Tuomi, Juoni; Sarajärvi, Anneli 2002. Laadullinen tutkimus ja sisällönanalyysi. Helsinki: Tammi.

Unified Modeling Language. 2005. OMG. Luettavissa osoitteessa <<https://www.omg.org/spec/UML/2.0/Superstructure/PDF>> (luettu 30.10.2018)