

Toni Haka-Risku

MVC-arkkitehtuurin toteutus web-sovelluskehyksissä

Tietotekniikan pro gradu -tutkielma

30. tammikuuta 2017

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Toni Haka-Risku

Yhteystiedot: Ag C416.1, `toni.haka-risku@student.jyu.fi`

Ohjaaja: Ohjaamaton työ

Työn nimi: MVC-arkkitehtuurin toteutus web-sovelluskehyksissä

Title in English: Usage of the gradu3 document class for \LaTeX theses

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Kaikki suuntautumisvaihtoehdot

Sivumäärä: 51+1

Tiivistelmä: Tässä tutkielmassa esitellään MVC-arkkitehtuurin toteutusta Python-pohjaisissa web-sovelluskehyksissä. Työssä selvitetään millä tavalla MVC on toteutettu sovelluskehyksissä ja vastaako se alkuperäistä MVC:n toteutusta, joka määritellään Krasnerin artikkelissa (Krasner 1988). Työssä tutkitut sovelluskehykset ovat Django, Pyramid ja Flask. Sovelluskehykset eivät toteuttaneet MVC:tä niinkuin se on määritelty. Flaskin avulla ei myöskään voida toteuttaa MVC:tä Krasnerin vaatimusten mukaisesti.

Avainsanat: \LaTeX , gradu3, pro gradu -tutkielmat, kandidaatintutkielmat, käyttöohje

Abstract: This document is a sample gradu3 thesis document class document. It also functions as a user manual and supplies guidelines for structuring a thesis document.

The English abstract of a thesis should usually say exactly the same things as the Finnish tiivistelmä.

Keywords: \LaTeX , gradu3, Master's Theses, Bachelor's Theses, user's guide

Esipuhe

Tutkielman aiheen valintaan vaikutti aiempi kokemus ohjelmoinnista niin vapaa-ajalla kuin työelämässäkin. Suurimmassa osassa ohjelmistoprojekteja, joissa olen työskennellyt, on ollut käytössä MVC-pohjainen sovelluskehys työkaluna. Django-sovelluskehysten sivuilla kuitenkin kyseenalaistetaan MVC:n toteutus sellaisenaan. Tästä sain idean kirjoittaa Pro Gradu tutkielman löytääkseni vastauksen kyseiseen ongelmaan. Jyväskylässä 30. tammikuuta 2017

Tutkielman tekijä

Termiluettelo

Sovellusarkkitehtuuri	kertoo millä tavalla sovellus on rakennettu
MVC	Model-View-Controller sovellusarkkitehtuuri
Sovelluskehys	on sovelluskehitykselle tarkoitettu pohja, joka kutsuu käyttäjän kirjoittamaa ohjelmakoodia ja tarjoaa ratkaisuja yleisimpiin toistuviin ongelmiin
WSGI	määrittää miten web-palvelin kommunikoi web-sovellusten kanssa ja millä tavalla web-sovellukset yhdistetään prosessoimaan pyyntöjä.
Template	Dynaaminen html-pohja, johon voidaan kirjoittaa ohjelmalogiikkaa
Piirikytkentä	Dynaaminen html-pohja, johon voidaan kirjoittaa ohjelmalogiikkaa
Pakettikytkentä	on tiedonsiirtomenetelmä, jossa yhteys data jaetaan paketeiksi. Osapuolien väliseen viestintään ei varata kaistaa, eikä osapuolien välille luoda katkeamatonta yhteyttä (Arlington 1981).
Piirikytkentä	on tiedonsiirtomenetelmä, jossa yhteys pysyy auki riippumatta siitä, onko osapuolien välillä liikennettä. Piirikytkentä on käytössä esim. puhelinverkoissa (Arlington 1981).

Kuviot

Kuvio 1. Bush 1981a	8
Kuvio 2. Model-View-Controller State and Message Sending krasner_desc	15
Kuvio 3. Kuva CounterView -näköymästä krasner_desc	26
Kuvio 4. Laskurisovelluksen toiminta	38
Kuvio 5. Pyramidin kommunikointi MVC-komponenttien kesken. Kuva on muokattu Krasnerin esittelemästä kommunikointimallista Krasner:desc	39

Sisältö

1	JOHDANTO	1
1.1	Tausta	1
1.2	Tutkimuskysymys	1
2	TUTKIMUKSEN RAKENNE	2
3	AIHEEN RAJAUS	3
4	KIRJALLISUUSKATSAUS	4
4.1	Sovelluskehitys ja arkkitehtuurit	4
4.2	MVC	5
4.3	Sovelluskehitys	6
4.4	Web-sovelluskehitykset	6
5	INTERNET	7
5.1	Historia	8
5.2	TCP	8
5.3	TCP:n toiminta	9
5.4	Internet Protokolla	11
5.5	HTTP	11
5.6	WEB-sokit	11
6	SOVELLUSKEHITYS	12
6.1	Sovelluskehityksen määritelmä	12
6.2	Sovellukset	13
7	WEB-SOVELLUKSET	14
7.1	Laatu	14
7.2	Ohjelmistojen tyypit	14
8	MVC	15
8.1	Historia	16
8.2	Malli (Model)	17
8.2.1	Näkymä (View)	18
8.2.2	Ohjain (Controller)	20
8.2.3	Esimerkkiohjelma	22
9	SOVELLUSKEHYKSET	27
9.1	Pyramid	29
9.2	Django	31
9.3	Flask	31
10	MVC:N VAATIMUKSET	33
11	MVC & PYRAMID	34

11.0.1Tiedostojen tarkastelu	34
11.0.2Sovelluksen toiminta	38
11.0.3Yhteenveto	39
12 YHTEENVETO.....	41
LÄHTEET	42
LIITTEET.....	45

1 Johdanto

1.1 Tausta

1.2 Tutkimuskysymys

MVC-arkkitehtuuri on saanut paljon huomiota web-sovelluskehityksen toteutuksissa ja useat web-sovelluskehitykset ovat luokiteltu MVC-pohjaisiksi sovelluskehityksiksi (Bestframeworks 2009). Se on kuitenkin alunperin tarkoitettu matalan tason käyttöliittymäsovellusten toteuttamiseen, jossa esimerkiksi hallitaan yksittäisiä näppäimistöä tulleita syötteitä eikä sitä ole suoraan tarkoitettu käytettäväksi web-sovellusten ohjelmointiin. Alkuperäisen MVC:n toteutuksen soveltuvuutta web-ohjelmointiin onkin epäilty. Esimerkiksi Leff soveltaa artikkelissaan MVC:n käyttämistä web-sovelluksissa, mutta samalla esittelee alkuperäisen MVC:n toteuttamisen ongelmana. Tämä johtuu web-sovelluksen jakautumisesta asiakkaan (client) ja palvelimen (server) välille (Avraham 2001). Myös Pyramid-sovelluskehityksen tekijät kyseenalaistavat MVC-arkkitehtuurin toteutuksen Pyramidissa ja uskovat MVC:n olevan sellaisenaan sopimaton web-ohjelmointiin, vaikka Pyramidin toteutus onkin hyvin lähellä alkuperäistä MVC:tä (Project 2010c). Django on vastaavasti toteutettu MVC:n pohjalta, mutta sen väitetään myös toteuttavan MVC hieman erilailla kuin MVC on alunperin tarkoitettu (Foundation n.d.).

Tutkimuksen tarkoituksena on selvittää ohjelmistokehityksen ja ohjelmistoarkkitehtuurien taustoja. Näiden pohjalta siirrytään tutkimaan millä tavalla MVC-arkkitehtuuri on toteutettu web-sovelluskehityksissä ja miten se eroaa alkuperäisestä MVC:n toteutuksesta. Lisäksi pyritään toteuttamaan MVC alkuperäisen toteutuksen mukaan käyttäen Flask-sovelluskehitystä (Krasner 1988). Havaintojen pohjalta selvitetään voidaanko MVC tuoda sellaisenaan sovelluskehityseen ja mitä mahdollisia ongelmia se tuo.

2 Tutkimuksen Rakenne

Tutkimus aloitetaan kirjallisuuskatsauksella, jossa tarkastellaan mitä aiempaa tutkimusta ohjelmistokehityksestä ja MVC:stä on tehty. Lisäksi käydään läpi mitä lähteitä löytyy Python-pohjaisista web-sovelluskehyksistä sekä ohjelmistoarkkitehtuureista. Tämän jälkeen tutkitaan MVC:n historiaa sekä millä tavalla MVC on tarkoitettu toteutettavaksi. Tässä vaiheessa käydään läpi jokaisen MVC-komponentin tarkoitus sekä niiden keskinäisen kommunikation rakentuminen. Lisäksi esitellään Dortmundin yliopistossa kirjoitettu esimerkkiohjelma Smalltalkilla siitä miten MVC:n toteutus tuodaan sovellukseen käytännössä.

MVC:n tarkastelun jälkeen esitellään tutkimuksessa käytetyt web-sovelluskehykset, joita käytetään apuna MVC:n tutkimisessa. Sovelluskehyksistä käydään läpi sen historia sekä yleisellä tasolla mihin käyttötarkoitukseen sovelluskehys on tarkoitettu. Tämän jälkeen verrataan MVC:n toteutusta erikseen jokaiseen sovelluskehykseen ja selvitetään millä tavalla niiden sovellusarkkitehtuuri mahdollisesti eroaa MVC:stä, Havaintojen perusteella pohditaan MVC:n mahdollisia ongelmia sovelluskehysien toteutuksessa ja selvitetään löytyykö sovelluskehysien arkkitehtuurista jotain yhtenäisiä piirteitä, mitkä ovat kytköksissä MVC:n toteutukseen. Saatujen tulosten pohjalta kirjoitetaan Flask-sovellus, joka toteuttaa MVC:n niinkuin se on alunperin tarkoitettu.

Tutkimuksen lopuksi koostetaan havainnoista yhteenveto, jossa pohditaan saatuja tuloksia ja selvitetään pystytäänkö niiden perusteella vastaamaan tutkimuskysymykseen.

3 Aiheen raja

Aluksi tarkastellaan ohjelmikstokehitystä ja ohjelmistoarkkitehtuureja yleisesti pohjustamaan MVC-arkkitehtuuria. MVC:n toteutus käydään hyvin yksityiskohtaisesti läpi ja sen vertaaminen rajataan Python-websovelluskehyskehyksiin. Tarkasteltavat web-sovelluskehyskehykset rajataan Pyramid-, Django- sekä Flask-sovelluskehyskehyksiin. Pyramid ja Django toteuttavat MVC:n kaltaisen sovellusarkkitehtuurin. Flask on sovelluskehys, joka tarjoaa vain välttämättömät kirjastot web-sovelluksen toteuttamiseen. Sitä käytetään tutkimuksessa työkaluna selvittämään miten MVC tulisi toteuttaa sovelluskehyskehykseen. MVC:stä on olemassa erilaisia versioita, joten sen määrittely tulee rajata tarkasti. Kun puhutaan MVC:stä tarkoitetaan tällä Krasnerin artikkelissa esiteltyjä määrittelyitä MVC:n toteutuksesta (Krasner 1988), jotka pohjautuvat Trygve Reenskaugin esittelemään MVC:n määritelmään (Parc 1979a).

Tarkasteltavat sovelluskehyskehykset käydään ensiksi yleisellä tasolla läpi, jonka jälkeen niitä tarkastellaan MVC:n näkökulmasta. Yleisellä tasolla tarkoitetaan sovelluskehyskehyksen historian ja käyttötarkoituksen esittelemistä. Sovelluskehyskehyksien muihin teknisiin ominaisuuksiin ei oteta kantaa. Flask-osiossa MVC-arkkitehtuuri toteutetaan niinkuin se on Krasnerin julkaisussa määritetty. Näiden havaintojen pohjalta pyritään vastamaan tutkimuskysymykseen.

4 Kirjallisuuskatsaus

Kirjallisuuskatsauksessa käydään läpi vaihe vaiheelta, miten lähdemateriaalia kerätään tutkimusta varten. Lähdemateriaalin haku toteutetaan hakukoneilla, jotka ovat tarkoitettu erityisesti tieteellisten artikkeleiden etsimiseen. Tässä tutkielmassa käytetyt hakukoneet ovat seuraavat: IEEE Xplore, ACM Digital Library, Google Scholar sekä joissakin tapauksissa Google:n yleinen hakukone. Yleistä hakukonetta on käytetty esimerkiksi sovelluskehityksien dokumentaatioiden etsintään.

Aluksi muodostetaan kokonaiskuva tuloksista, jolloin silmäilläään läpi saatuja artikkeleita. Tässä vaiheessa tarkoitus ei ole vielä valita mitään pohjaksi tutkielmalle, vaan kerätä informaatiota siitä millainen lähdemateriaali on tarjolla kokonaisuudessaan. Saaduista tuloksista poimitaan artikkeleita, jotka sopivat tutkimuksen aihepiiriin. Seuraavaksi artikkeleista valitaan tutkielmalle pohjakirjallisuus. Tässä vaiheessa artikkelit luetaan huolellisesti läpi ja varmistutaan siitä, että ne ovat tieteellisesti päteviä tutkielmaa varten. Erityisesti kiinnitetään huomiota viittausten määrän valittaessa tärkeimmät lähdemateriaalit. Tutkielmassa esiintyy myös satunnaisia viittauksia, joita ei ole kirjallisuuskatsauksessa mainittu. Tutkimuksen pääkirjallisuus kuitenkin käydään läpi kirjallisuuskatsauksessa. Haussa käytetään seuraavia hakutermejä: "Software engineering", "Software development", "MVC", "MVC Architecture", "frameworks", "web frameworks" ja "MVC- Architecture". Tarkasteltavat artikkelit rajataan kuitenkin niihin, jotka esittelevät suoraan MVC:tä itseään, tarjoavat lähdemateriaalin sovelluskehityksen esittelyyn tai auttavat pohjustamaan yleisesti ohjelmistokehitysten tarkastelua.

4.1 Sovelluskehitys ja arkkitehtuurit

Sovelluskehityksen tarkasteluun löytyy runsaasti lähdemateriaalia. Merkittäväksi lähteeksi valitaan Ian Sommervillen kirjoittama *Software Engineering 9th Edition* -kirja, jossa käydään läpi mitä kaikkea isojen ohjelmistojen kehitykseen kuuluu (Sommerville 2010). Kirja on jaettu neljään pääosiin: *Introduction to Software Engineering, Dependability and Security, Advanced Software Engineering* ja *Software Management*. Tutkimuksessa lähteenä käytetään *Introduction to Software Engineering* ja *Advanced Software Engineering* -osioita.

Introduction to Software engineering käy läpi ohjelmistojen suunnittelua ja toteutusta yleisesti. Tällaisia ovat esimerkiksi vaatimusmäärittelyt, prosessit, arkkitehtuuri ja testaus. *Advanced Software Engineering* esittelee ohjelmistojen kirjoittamista siten, että ne olisivat mahdollisimman uudelleen käytettäviä ja ylläpidettäviä. Tähän tarjotaan ratkaisuksi erilaisia sovellusarkkitehtuureja (esim. *Component-based architecture* ja *Service-oriented architecture* (Sommerville 2010)).

Toiseksi tärkeäksi lähteeksi valitaan ietf.org jota käytetään Internetin synnyn pohjalähteeksi, tcp ja muut protokollat?— TODO: tähän jatkoa

4.2 MVC

Google Scholarin tuloksista löytyy kolme artikkelia MVC:stä, jotka sopivat lähdemateriaaliksi tutkimukseen. Ensimmäinen artikkeleista on John Deaconin kirjoittama artikkeli, joka tarkastelee lyhyesti MVC:tä (Deacon 2009). Artikkeli on kuitenkin hyvin suppea, mutta selittää tiivistetysti MVC:n idean. Toinen artikkeli on Steve Burbeckin kirjoittama, joka käsittelee MVC:tä sellaisena kuin sitä käytettiin Smalltalkissa (Steve 1992). Burbeckin artikkeliin viitataan monissa MVC:tä käsittelevissä julkaisuissa, joten sen arvo tämän tutkielman pohjakirjallisuudessa on vahva. Viittausten määrä on katsottu hakemalla artikkelia Google Scholarin hakukoneessa. Seuraavaksi kartoitetaan pohjakirjallisuutta käyttäen ACM Digital Library sekä IEEE Xplore -hakukoneita. Kolmas artikkeli Glenn E. Krasnerin kirjoittama julkaisu, jossa esitellään MVC:n toteutusta erilaisissa Smalltalk-sovelluksissa. Julkaisusta löytyy useita versioita, joista tässä tutkielmassa käytetään molempia (Krasner 1988) (.) Tähän artikkeliin on myös viitattu runsaasti, joten se on Burbeckin julkaisun kanssa tärkeimpiä lähteitä MVC:n pohjakirjallisuudessa. Kirjoitushetkellä viittauksia Krasnerin artikkeliin on 2263. Monien MVC-arkkitehtuuria soveltavien artikkeleiden lähdeviitteistä löytyy viittauksia Burbeckin ja Krasnerin artikkeleihin. Tämän perusteella pystytään toteamaan kyseisten artikkeleiden olevan tieteellisesti päteviä ja tarjoavan kattavan lähdemateriaalin MVC:n pohjaksi. Burbeckin ja Krasnerin kirjoittamien artikkeleiden taustalta löytyy MVC-arkkitehtuurin alkuperäinen kehittäjä Trygve Reenskaug, jonka omia julkaisuja sekä kotisivujen MVC-osiota käytetään myös lähteenä tutkielmassa (Parc 1978). Erityisesti Reenskaugin ja Adele Goldbergin julkaisu, jossa kerrotaan jokaisen MVC komponentin tehtävä....

4.3 Sovelluskehitys

Sommervillen kirja isona osana lähteitä

4.4 Web-sovelluskehikset

Web-sovelluskehiksistä löydetty kirjallisuus on hyvin suppea, eikä tämän varaan voida rakentaa kovinkaan perusteellista tieteellistä pohjaa. Tämän vuoksi tutkimuksessa joudutaan osaksi turvautumaan sovelluskehysten omaan dokumentaatioon täydentämään lähdemateriaalia. IEEE Xplore ja ACM Digital Libraryn avulla löytyy kolme julkaisua, joita käytetään tutkimuksen pohjana sovelluskehiksiä tarkastellessa. Ensimmäinen artikkeli on Okanovicin ja Mateljan kirjoittama artikkeli, jossa esitellään web-sovelluskehysten suunnittelua (Okanovic 2011). Se myös sivuuttaa lyhyesti MVC:tä. Toisena artikkelina käytetään ACM:stä tuloksena saatua Iwan Vosloon julkaisua, jossa käydään läpi yleisesti web-sovelluskehysten rakennetta (Kourie 2008). Kolmanneksi hyödynnetään Ignacion artikkelia, jossa esitellään ketteriä web-sovelluskehiksiä sekä millä tavalla näitä tulisi vertailla (Fernández-Villamor 2008). Lisäksi käytetään IEEE:stä Ahamedin julkaisua, joka esittelee yleisesti asioita joita tulisi ottaa huomioon sovelluskehystä valittaessa (Sheikh I. Ahamed ja Pezewski 2008).

Google Scholarin hakutuloksista löytyi Liza Daly:n kirjoittama ja O'Reillyn julkaisema "Next Generation Web Frameworks in Python", joka sisältönsä puolesta sopii hyvin pohjaksi tutkimuksessa käsiteltävien sovelluskehysten lähdemateriaaliksi (Daly 2007).

5 Internet

Internet ja sen arkkitehtuuri on syntynyt hyvin pienistä saavutuksista. Internetin taustalla ei ole alunperin ollut minkäänlaista suurempaa suunnitelmaa. Tutkiessamme Internetin arkkitehtuuria tulee muistaa, että tekniset muutokset ovat jatkuvia tietotekniikan alalla. Monet arkkitehtuurilliset ratkaisut, jotka olivat suosiossa muutamia vuosia sitten ovat nyt jo vanhentuneet. Taas ratkaisut, jotka tänä päivänä nähdään hyvinä, voivat jo huomenna olla huonoja. Voidaan sanoa, että ainut asia joka Internetin suhteen pysyy on jatkuva muutos. Internet yhteisöjen monet jäsenet väittävät, että Internetillä ei ole arkkitehtuuria, mutta kuitenkin eräänlaisia perinteitä, johon koko Internet pohjautuu. Internetin tavoite on tarjota yhteydet, jossa työkaluna käytetään Internet Protokollaa. Monien yhteisön jäsenien mielestä tärkeää ei ole se millä tavalla yhteydet luodaan verkon sisällä, vaan se millä tavalla yhteyksien alkua loppupää käyttäytyvät. Tähän tärkeimpänä argumenttina pidetään sitä, että useat tarvittavat toiminnallisuudet voidaan toteuttaa vain IP:tä käyttävien systeemien puolella. Jokainen hyvin suunniteltu verkko sisältää mahdollisuuden jollakin tasolla epäonnistua viestin kuljettamisessa. Tällaisen ongelman kanssa tulisi toimia siten, että vastuu kommunikaation eheydestä annetaan ulkopuolisille sovelluksille. Nykyinen Internetin eksponentiaalinen kasvu näyttää kuitenkin sen, että yhteydet ovat paljon arvokkaampia kuin yksittäiset sovellukset. Tällaiset yhteydet vaativat yhteistyötä palveluntarjoajien ja televiestintäympäristöjen välillä. Carpentern 1996.

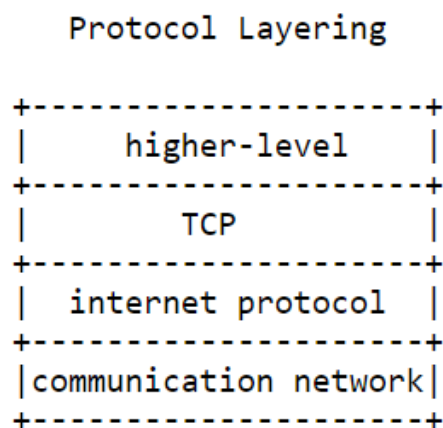
Yleisesti on haluttua, että olisi vain yksi protokolla, jonka kautta yhteydet voidaan muodostaa. Tämän protokollan päälle voidaan kuitenkin rakentaa monia muita protokollia, jotka hoitavat muita erilaisia tarpeita. Käytännössä on ainakin kaksi syytä miksi verkko-protokollien tasoja tarvitaan useita. Esimerkiksi eri IP:stä voidaan haluta vaihtaa toiseen versioon. Myös uudet vaatimukset voivat synnyttää uusia protokollia. Internet tason protokolla tulee olla täysin riippumaton laitteistosta. Tällöin Internet voidaan toteuttaa minkä tahansa digitaalisen teknologian päälle. Carpentern 1996.

5.1 Historia

Internetin historia on lähtöisiin tarpeesta luoda yhteyksiä tietokoneiden välille, joiden avulla voidaan luoda verkkoja. Verkkojen kautta erilaiset sovellukset voivat keskustella keskenään. Tämä on johtanut tarpeeseen luoda standardit, jotka määrittävät sen millä tavalla yhteyksiä tulisi luoda ja ylläpitää. TCP (Transmission Control Protocol) on standardi, joka määrittää yhteyden luonnin tietokoneiden välillä sekä niiden välisen kommunikoinnin. TCP:n avulla sovellukset pystyvät lähettämään dataa toisilleen verkon yli. TCP kehitettiin Yhdysvaltain asevoimien tutkimusorganisaatiossa DARPA:ssa (Defense Advanced Research Projects Agency) Bush 1981a.

5.2 TCP

TCP on tarkoitettu toimivan eri tasoihin jaetussa protokollien hierarkiassa, joka tukee verkko-sovelluksia. Sen tarkoitus on pystyä toimimaan monien alemman tason protokollien päällä ja olla ottamatta kantaa käytetäänkö yhteyden muodostuksessa esimerkiksi paketti- tai piirikytkentää. TCP tarjoaa luotettavan DARPA määrittää eri protokollien tasot seuraavasti:



Kuvio 1. Bush 1981a

TCP on rakennettu Internet Protokollan päälle, joka tarjoaa TCP:lle työkalut määrittää lähettäjän ja vastaanottajan osoitteet erilaisissa verkoissa. Internet Protokolla hoitaa myös mahdollisten TCP-segmenttejen pirstaloinnin ja uudelleen kokoamisen, jolloin TCP:n ei tarvitse näistä huolehtia. Se myös pitää mukanaan tiedon TCP-segmenttejen järjestyksestä, tieto-

turvamäärityksistä sekä lajittelusta. TCP tarjoaa rajapinnan luotettavalle yhteydelle useista verkoista koostuvassa ympäristössä Bush 1981a

TCP koostuu kahdesta rajapinnasta, joista toinen toimii käyttäjälle tai sovellukselle ja toinen alemman tason protokollille kuten Internet Protokolalle. Sovelluksille tarkoitettu rajapinta koostuu vastaavista kutsuista mitä käyttöjärjestelmä tarjoaa tiedostojen hallintaan. Esimerkiksi se sisältää kutsut yhteyden avaamiseen ja sulkemiseen sekä datan vastaanottoon ja lähteykseen yhteyden läpi. TCP pystyy myös asynkronisesti kommunikoimaan sovelluksien kanssa. Rajapinta alemmille protokollille on yleisesti määrittelemätöln, mutta se olettaa rajapinnassa olevan mekansimin, jossa kaksi tasoa pystyvät asynkronisesti siirtämään dataa toistensa välillä. Yleensä kyseinen määrittely oletetaan löytyvän alemman tason protokollalta. TCP on tarkoitettu toimivan geneerisessä ympäristössä, mutta yleisesti voidaan olettaa alemman tason protokollan olevan Internet Protokolla Bush 1981b.

5.3 TCP:n toiminta

TCP:n tärkein tehtävä on tarjota turvallinen ja luotettava yhteys kahden prosessin välille. Tarjotakseen tällaisen palvelun Internetin päälle,tulee tietyt osa-alueet toteutua. Alla TCP:n toteuttamat osa-alueet listattuna selityksineen:

Datan siirto	TCP pystyy lähettämään jatkuvasti paketteja kummankin käyttäjän suuntaan. Joskus käyttäjien tulee varmistaa, että kaikki lähetetty data on lähetetty vastaanottajalle. Tähän tarkoitukseen on kehitelty push -toiminto, joka käskää TCP:tä lähettämään datan vastaanottajalle Bush 1981b.
Luotettavuus	TCP:n tulee pystyä selviytymään tilanteesta, jossa data on jollain tavalla korruptoitunut tai hävinnyt. Tämä on ratkaistu lisäämällä jokaiseen pakettiin sekvenssinumero ja jokaisessa viestissä vaaditaan kuittausta vastaanottajalta. Jos kuittausta ei jostain syystä saada, lähetetään paketti uudelleen. Vastaanottajan puolella sekvenssinumeron avulla pystytään päättelemään oikea järjestys Bush 1981b.

Datavirran hallinta	TCP antaa vastaanottajalle mahdollisuuden rajata datamääriä. Tämä mahdollistetaan antamalla vastaanottajalle mahdollisuus lähettää kuittausviestin mukana ikkuna, joka määrittää rajat datalle Bush 1981b.
Kanavointi	Koska yhden isäntäkoneen on mahdollista pavella useita TCP-yhteyksiä rinnakkain, tulee TCP:n tarjota lista osoitteita ja portteja yksittäistä severiä kohden. Jokaiselle yhteydelle luodaan soketti (socket). Soketti-pari muodostaa uniikin yhteyden, missä yksittäistä sokettia voidaan myös käyttää moniin eri yhteyksiin rinnakkain. Hyvänä esimerkkinä yhden soketin jakamisesta on erilaiset logitusprosessit, joita käytetään jatkuvasti. Erilisiin portteihin liittäminen hoidetaan aina jokaisen isäntäkoneen toimesta Bush 1981b.
Yhteydet	Luotettavuuden ja kanavoinnin toteutus vaatii TCP:tä erillisen tilan alustamista ja ylläpitoa jokaiselle datavirralle. Tietoa soketeista, sekvenssinumeroista ja ikkunoiden kokoa kutsutaan yhteydeksi. Jokainen yhteys määritellään sokettiparin perusteella. Kun kaksi prosessia haluaa kommunikoida kekskenään, kummankin TCP:n tulee ensin luoda yhteys, jolloin kummankin tila alustetaan. Kun yhteys on valmis, se terminoidaan tai suljetaan resurssien vapauttamiseksi. Koska yhteydet luodaan epävakaan internet-yhteyden sekä epäluotettavien prosessejen välille, TCP käyttää aikapohjausta kättely-mekanismia yhteyden muodostamisessa. Tämän avulla vältetään virheelliset yhteydenmuodostukset Bush 1981b.
Priorisointi ja tietoturva	TCP:n käyttäjät voivat hallita priorisointia sekä tietoturvaa erilaisilla arvoilla. Tällainen on esimerkiksi TCP-oktetin sisältämät ensimmäiset kolme bittiä, jotka määrittävät oktetin tärkeysjärjestyksen Xiao 2000

<https://tools.ietf.org/html/rfc793>

5.4 Internet Protokolla

5.5 HTTP

5.6 WEB-soketit

6 Sovelluskehitys

Katsomalla ympärille nykyajan yhteiskunnassa, huomataan monien asioiden toimivan ohjelmien varassa. Puhelimet, elektroniset kellot, autot ja erilaiset yhteiskunnalliset palvelut sisältävät kaikki ohjelman, jolla toimitoja sekä dataa hallitaan. Nykyajan modernia yhteiskuntaa ei pystytä ylläpitämään ilman sovelluksia. Monet tuotantolinjastot sekä rahoitusjärjestelmät teollisuudessa ovat täysin automatisoituja. Viihdeteollisuus kuten musiikki, pelit, elokuvat ja televisio ovat kaikki monimutkaisten sovellusten varassa. Ihmisten tiedot ovat monissa valtion hallintajärjestelmissä sekä niin julkisissa kuin yksityisissäkin potilastietojärjestelmissä. Nykypäivänä katsottaessa ympärille löydämme lähes poikkeuksetta jonkin asian, jonka taustalla on ohjelma. Suuri määrä ihmisiä kirjoittaa sovelluksia nykypäivänä. Liike-elämässä ihmiset kirjoittavat taulukkolaskentaohjelmia helpottaakseen työtään. Tutkijat ja insinöörit kirjoittavat ohjelmia prosessoimaan kerättyä dataa ja harrastelijat kirjoittavat ohjelmia viihdyttääkseen itseään. Suurin osa sovelluskehityksestä on kuitenkin toteutettu ammattilaisten toimesta tukemaan liiketoimintamalleja, kehittämään laitteita sekä rakentamaan erilaisia palveluita. Ammattilaisten toteuttamat ohjelmistot toteutetaan usein ryhmissä ja ne ovat suunnattu käytettäväksi muille kuin ohjelmoijille itselleen (Sommerville 2010).

6.1 Sovelluskehityksen määritelmä

Sovelluskehityksellä tarkoitetaan koko ohjelmiston elinkaaren mittaista tuotantoprosessia alkumäärittelyistä ylläpitoon asti. Se on ammattihaara, joka on erikoistunut tutkimaan ja tuottamaan sovelluksia. Sovelluskehittäjät käyttävät erilaisia teorioita, käytänteitä ja työkaluja ratkaisemaan sovelluskehitykseen liittyviä ongelmia. Sovelluskehitys ei kuitenkaan keskity ainoastaan teknisiin ongelmiin vaan myös projektin hallintaan ja työkaluihin, jotka tukevat sovellusten kehittämistä Sommerville 1.1.1. Sovelluskehityksessä tarkoituksena on saada aikaan tuloksia

6.2 Sovellukset

Ohjelmistot ovat abstrakteja ja aineettomia. Ne eivät noudata mitään fysiikan lakeja eivätkä ne koostu mistään materiasta tai tuotantoprosessista. Tämä tekee ohjelmistokehityksestä helppoa ja vapaata, koska sille ei ole asetettu mitään luonnollisia esteitä. Rajojen puuttuminen johtaa myös siihen, että ne voivat olla usein erittäin monimutkaisia, vaikeita ymmärtää ja kallista muuttaa. Ohjelmistojen tyypit vaihtelevat sulautetuista järjestelmistä maailmanlaajuisiin tietojärjestelmiin. Tietojärjestelmän kehittäminen organisaatiolle eroaa täysin esimerkiksi ohjainlaitteen ohjelmoinnista. Näitä kahta taas ei voi verrata esimerkiksi grafiikkapohjaiseen pelikehitykseen. Kuitenkin kaikki nämä tarvitsevat ohjelmistokehitystä (Sommerville). Ohjelmistot eivät ole pelkästään tietokoneohjelmia vaan ne koostuvat ohjelman lisäksi dokumentaatioista sekä erilaisista konfiguraatioista, joilla määritellään ohjelma toimimaan tietyllä tavalla. Dokumentaatio sisältää usein tiedon ohjelmiston rakenteesta sekä käyttäjille suunnatun ohjeen miten sovellusta hallitaan ja käytetään (Sommerville 2010) section 1.1. Testaus ja sen automaatio on myös suuren osana ohjelmistoja nykypäivänä <lähde tähän pohjaamaan väitettä>.

Sommerville (1,1) jakaa ammatilaisten kehittämät ohjelmistotuotteet kahteen kategoriaan: geneeriset tuotteet ja räätälöidyt tuotteet. Geneeristen tuotteiden määritelmät ovat sitä kehittävä organisaation hallinnassa. Päättäväältä ominaisuuksista on siis kehittävällä osapuolella. Räätälöidyissä tuotteissa tilaaja määrittelee ohjelmiston tarkoituksen ja sen ominaisuudet. Näitä kahta tyyppiä voidaan kuitenkin yhdistää, mikä on varsin yleistä ohjelmistoille. Pohjana voi olla geneerinen tuote, johon toteutetaan erilaisia tasoja, jotka ovat määrittely tilaajan toimesta (Sommerville 2010)

7 Web-sovellukset

World Wide Web:n kehityksellä on ollut tärkeä rooli sovelluksien kehityksessä. Alunperin Web oli yliopistojen käytössä datan tallennusta varten ja sillä oli hyvin vähän vaikutusta sovelluksiin. Webin ympärille kirjoitetut sovellukset olivat käynnissä paikallisissa tietokoneissa ja niihin oli vain pääsy organisaation sisällä. 2000-luvun alussa Web alkoi kehittyä laajemmin. Selaimiin lisätyt ominaisuudet mahdollistivat Web-pohjaisten sovellusten käytön siten, että erillisen sovellukselle tarkoitetun käyttöliittymän sijasta voitiin käyttää selainta. Tämä johti monien erilaisten sovelluksien kehitykseen, jotka tarjosivat innovatiivisia palveluita käytettäväksi selaimen kautta. Webin kehityksen myötä selaimet kehittivät siten, että ne pystyivät ajamaan myös pieniä sovelluksia paikallisesti. Tämä muutti organisaatioiden sekä liiketoiminnan tapoja käyttää sovelluksia. Sen sijaan, että sovellukset olisi asennettu käyttäjien tietokoneille, ne voitiin asentaa suoraan yhdelle web-palvelimelle. Tämä laski myös kustannuksia tehdä muutoksia sovelluksiin. Suurin osa on liiketoimien harjoittajista on siirtynyt käyttämään web-pohjaisia sovelluksia siellä missä se on mahdollista **Software eningeering and the Web**; Sommerville 2010.

Seuraava vaihe web-sovellusten kehityksessä olivat web-palvelut. Web-palvelut ovat sovelluskomponentteja, jotka tarjoavat hyödyllisiä palveluja käytettäväksi muille sovelluksille. Siinä missä web-sovellus tarjoaa käyttöliittymän sovellukselle, web-palvelu tarjoaa rajapinnan käyttöliittymäsovellukselle käytettäväksi. Web-palvelu on yksittäinen osa sovellusta ja sovelluksella voi olla monia web-palveluja käytettävänä.

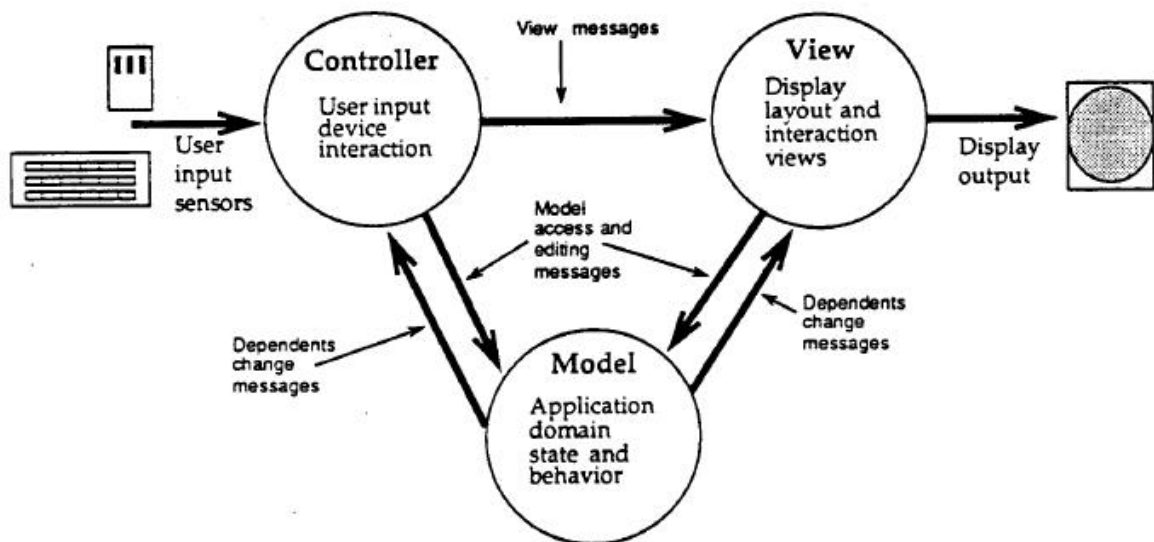
7.1 Laatu

7.2 Ohjelmistojen tyypit

- lista typeist selityksineen - viite barry boehm "For another thing, unlike the engineering of electrons, materials, or chemicals, the basic software elements we engineer tend to change significantly from one decade to the next. "

8 MVC

MVC-arkkitehtuurin perusajatus on erottaa käyttöliittymä sovelluslogiikasta ja näin tehdä sovelluksesta helposti ylläpidettävä kolmen eri komponentin avulla: Malli (Model), Näkymä (View) ja Ohjain (Controller). Jokainen komponentti on erikoistunut sovelluksessa johonkin tiettyyn tehtävään. Mallin tehtävänä on hallita sovelluksen tilaa ja vastata sen käsittelemästä datasta ohjaimelle ja näkymälle. Näkymän tehtävänä on taas näyttää sovelluksen käyttöliittymä ja sitä kautta mallin dataa. Ohjaimen tarkoitus on ottaa vastaan syötteitä käyttäjältä käskien mallia ja näkymää muuttumaan tarvittaessa.



Kuvio 2. Model-View-Controller State and Message Sending **krasner_desc**

Jokaisella komponentilla on oma rajattu tehtävänsä ja ohjelmakoodi tulee jakaa näiden komponenttien kesken. Jotta MVC:tä pystyttäisiin käyttämään tehokkaasti, tulee ymmärtää komponenttien työnjako sekä se kuinka komponentit kommunikoivat keskenään (Steve 1992).

Luodessamme MVC-arkkitehtuurin toteuttavia komponentteja, tulee ne periä jostakin abstraktista pohjaluokasta (Model, View tai Controller), joka määrittelee kyseisen komponentin käyttäytymisen MVC:ssä (**krasner_desc**). Tässä kappaleessa käydään jokaisen komponentin toteutus erikseen läpi käyttäen ohjelmointikielenä Smalltalkia. Lähteenä käytetään Krasnerin julkaisua (**krasner_desc**).

Yleisesti MVC-komponenttien toimintaa kuvaavassa esimerkissä käyttäjältä tulee jokin syöte, jonka sillä hetkellä aktiivinen ohjain ottaa vastaan. Syötteen perusteella ohjain lähettää mallille viestin. Malli puolestaan tekee sille määrättyjä operaatioita muuttaen tilaansa ja lähettää edelleen viestin muutoksestaan kaikille siihen liitetyille riippuvuuksille (näkymät ja ohjaimet). Näkymät voivat tämän jälkeen kysyä mallilta sen nykyistä tilaa ja päivittää itsensä, jos siihen on tarvetta. Ohjaimet voivat myös muuttaa tilaansa riippuen mallin tilasta (**krasner_desc**).

Suurin merkitys MVC:llä on luoda silta ihmismielen hahmottamalle mallille ja tietokoneessa esiintyvälle mallille. Oikein toteutettuna MVC:n avulla luodaan illuusio siitä, että käyttäjä kommunikoi suoraan mallin kanssa. Todellisuudessa kuitenkin ohjain ja näkymä muodostavat yhdessä rajapinnan sille, miltä malli näyttää ulospäin ja miten sitä käsitellään. Ohjain huolehtii syötteiden vastaanottamisesta ja käsittelemisestä. Näkymä taas huolehtii mallin graafisesta puolesta (**reenskaug_tools**).

8.1 Historia

MVC:n esitteli Norjalainen Trygve Reenskaug ollessaan mukana Xerox PARC -tutkimushankkeessa. Ensimmäinen julkaisu MVC:stä kirjoitettiin vuonna 1978 samassa tutkimuskeskuksessa. Tuolloin julkaisussa esiteltiin kolmen komponentin sijasta neljä komponenttia: Malli (Model), Näkymä(View), Ohjain(Controller) sekä Muokkaaja(Editor). Muokkaaja on väliaikainen komponentti, jonka näkymä luo itsensä ja syötelaitteiden välille. Muokkaaja-komponentista kuitenkin luovuttiin käsitteenä ja se sisällytettiin näkymään ja ohjaimeen (Parc 1978). Alkuperäinen Xerox PARC:n tuottama raportti MVC:stä oli Reenskaugin vuonna 1979 kirjoittama THING-MODEL-VIEW-EDITOR (Parc 1979b). Raportti esitteli MVC:n komponentteja käyttäen hyväksi esimerkkejä Reenskaugin omasta suunnittelutyöstä. Thing-komponentilla mallinnettiin jotakin isompaa kokonaisuutta, joka hallitsee pienempiä kokonaisuuksia. Sitä voidaan ajatella eräänlaisena suurena mallina, joka on jaettu useisiin pienempiin malleihin. Editor-komponentti luo rajapinnan käyttäjän ja yhden tai useamman näkymän välille. Se tarjoaa käyttäjälle sopivan komento-rajapinnan kuten esimerkiksi valikon, joka vaihtuu sisällön muuttuessa (Parc 1979b). Reenskaug hylkäsi kuitenkin Editor- ja Thing-komponentin ja päätyi Adele Goldbergin avustuksella termeihin Models-Views-Controllers julkaisten saman

vuoden lopulla raportin, jossa määritellään lyhyesti jokaisen komponentin tehtävä (MODELS-VIEWS-CONTROLLERS) (Parc 1979a). Koska MVC:n historia ja suurin osa MVC:n alkuperäisistä julkaisuista pohjautuvat Smalltalk-ohjelmointikieleen, esitellään myös tässä tutkielmassa MVC:n toteutusta Smalltalkilla. Tämä ei kuitenkaan rajoita tarkastelua, koska arkkitehtuurin idea pysyy täysin samana riippumatta ohjelmointikielestä.

8.2 Malli (Model)

Malli pitää yllä sovelluksen tilaa sekä vastaa sovelluksen tallentamasta datasta. Se voi olla esimerkiksi kokonaislukumuuttuja laskuri-sovelluksessa, merkkijono-olio tekstinkäsittelyohjelmassa tai mikä tahansa monimutkainen olio (**krasner_desc**). Kaikkein yksinkertaisimmassa tapauksessa mallin ei tarvitse kommunikoida ollenkaan ohjaimen ja näkymän kanssa, vaan toimia passiivisena säiliönä datalle. Tällaisesta tilanteesta on hyvä esimerkki yksinkertainen tekstieditori, jossa teksti nähdään juuri sellaisena kuin se olisi paperilla. Tässä tapauksessa mallin ei tarvitse ottaa vastuuta kommunikoinnista näkymälle, koska muutokset tekstiin tapahtuvat käyttäjän pyynnöstä. Tällöin ohjain ottaa vastaan käyttäjän syötteet ja voi esimerkiksi ilmoittaa näkymälle muutoksesta, jolloin näkymä päivittää mallin. Ohjain voi myös päivittää mallin ja ilmoittaa tästä näkymälle, jolloin näkymä voi pyytää mallin sen hetkistä tilaa. Kummassakaan tapauksessa mallin ei tarvitse tietää ohjaimen ja näkymän olemassaolosta (Steve 1992).

Malli ei kuitenkaan aina voi olla täysin passiivinen. Se voi myös muuttua ilman, että se tarvitsee ohjaimen tai näkymän käskyä. Otetaan esimerkiksi malli, joka muuttaa tilaansa satunnaisin väliajoina. Koska malli muuttaa itseään, täytyy sillä olla jokin yhteys näkymään, jotta se voi antaa tiedon muutoksestaan (Steve 1992). Datan kapseloinnin ja ohjelmakoodin uudelleen käytön kannalta ei ole kuitenkaan järkevää, että malli on suoraan yhteydessä näkymään ja ohjaimeen. Ohjaimen ja näkymän tulee siis olla riippuvaisia mallista, mutta ei toisinpäin. Näin mahdollistetaan myös se, että mallilla voi olla useita näkymiä ja ohjaimia (**krasner_desc**).

Yleensä mallin tila muuttuu ohjaimista tulleiden käskyjen kautta. Tämän muutoksen tulisi heijastua kaikkiin näkymiin, jotka ovat sidottuja malliin. Tällaisia tilanteita varten kehitettiin

riippuvuudet (*dependents*). Riippuvuuksilla tarkoitetaan listaa niistä ohjaimista ja näkymistä, jotka ovat sidottuja malliin. Mallilla tulee siis olla lista riippuvuuksista ohjaimiin ja näkymiin sekä myös kyky lisätä ja poistaa niitä. Malli ei siis tiedä mitään yksittäisistä riippuvuuksista, mutta pystyy kuitenkin lähettämään itsestään muutosviestejä (*change messages*) listassa oleville ohjaimille ja näkymille. Mallin tuottamat muutosviestit voivat olla minkä tyyppisiä tahansa, joten ohjaimet ja näkymät reagoivat niihin omalla määritellyllä tavallaan (Krasner 1988, s.2-3).

Mallille määritellään pääluokka *Model* ja tälle viitemuuttuja *dependents*, joka viittaa yhteen riippuvaan komponenttiin tai listaan riippuvista komponenteista. Kaikki uudet mallit tulee periä niiden pääluokasta, jotta saavutetaan sama toiminnallisuus kaikkiin mallikomponentteihin. Komponenttien tieto mallin muutoksista tukeutuu täysin mallin riippuvuusmekanismiin. Kun jokin komponentti luodaan, se rekisteröi itsensä malliin riippuvuudeksi ja samalla tavalla se myös poistaa itsensä (Steve 1992). Näkymät käyttävät riippuvuusmekanismia päivittääkseen itsensä mallin muutoksien perusteella. Esimerkiksi mallin muuttuessa lähetetään *changed*, jonka pohjalta jokainen riippuvuus saa *update* -viestin. Viestillä voi olla myös erilaisia parametrejä, joiden perusteella viestiä pystytään tarkentamaan. Esimerkiksi mallin, johon on liitetty useita näkymiä, ei välttämättä tarvitse lähettää kaikille näkymille viestiä muutoksestaan. Se voi välittää viestin mukana parametrina tiedon muutoksesta, jonka perusteella jokainen vastaanottaja voi päättää miten toimia (Steve 1992).

Alkuperäinen *update* -metodi on peritty *Object* -luokasta, eikä se tuolloin tee vielä yhtään mitään. Useimmilla näkymillä se on kuitenkin toteutettu näyttämään näkymä uudestaan kutsuttaessa. Tämä *changed/update* -mekanismi valittiin toimimaan kommunikaatiokanavana mallien ja näkymien välille, koska se aiheuttaa vähiten rajoituksia ja esteitä (Steve 1992).

8.2.1 Näkymä (View)

Näkymän tehtävänä on huolehtia graafisesta puolesta MVC:ssä. Näkymä pyytää yleensä mallilta datan ja tämän pohjalta näyttää käyttäjälle käyttöliittymän sovellukseen. Toisinkuin malli, jota pystytään rajoittamattomasti yhdistelemään moniin näkymiin ja ohjaimiin, jokainen näkymä on liitetty yhteen ohjaimeen. Näkymä siis sisältää viitteen ohjaimeen ja ohjain

sisältää viitteen näkymään. Kuten ohjain, näkymä on myös rekisteröity mallin riippuvuuksiin. Kummatkin sisältävät siis myös viitteen siihen malliin, johon ne on rekisteröity (Steve 1992). Jokaisella näkymällä on tasan yksi malli ja yksi ohjain (**krasner_desc**).

Näkymä vastaa myös MVC-komponenttien sisäisestä kommunikaatiosta MVC-kolmikon luontivaiheessa. Näkymä rekisteröi itsensä riippuvuudeksi malliin, asettaa viitemuuttujansa viittamaan ohjaimeen ja välittää itsestään viestin ohjaimelle. Viestin avulla ohjain rekisteröi näkymän omaan viitemuuttujaansa. Näkymällä on myös vastuu poistaa viitteet sekä rekisteröinnit (Steve 1992).

Näkymä ei sisällä ainoastaan komponentteja datan näyttämiseen ruudulla, vaan se voi sisältää myös useita alanäkymiä (*subviews*) ja ylänäkymiä (*superviews*). Tästä muodostuu hierarkia, jossa ylänäkymä hoitaa aina jonkun suuremman kokonaisuuden, kuten esimerkiksi näytön pääikkunan. Alanäkymä taas huolehtii jostain pienemmästä yksityiskohdasta pääikkunassa. Näkymillä on myös viite erilliseen transformaatioluokkaan, joka hoitaa kuvan soveltamisen ja yhdistämisen alanäkymien ja ylänäkymien välillä. Jokaisella näkymällä tulee siis olla toteutus, jolla hoidetaan alanäkymien poistaminen sekä lisääminen. Samalla tulee määritellä ominaisuus, jolla sisäiset transformaatiot tuodaan transformaatioluokalle. Tämä helpottaa näkymän ja sen alanäkymien yhdistämistä (**krasner_desc**). Burbeck havainnollistaa Smalltalkilla kirjoitetulla esimerkillä kuinka MVC-kolmikko luodaan. Esitetyssä esimerkissä on yksinkertaistettu versio MVC-kolmikon luonnista siten, että mukana on myös ylä- ja alanäkymien toteutus.

```
1  openListBrowserOn: aCollection label: labelString initialSelection: sel
2    "Create and schedule a Method List browser for
3    the methods in aCollection."
4    | topView aBrowser |
5    aBrowser ← MethodListBrowser new on: aCollection.
6    topView ← BrowserView new.
7    topView model: aBrowser; controller: StandardSystemController new;
8          label: labelString asString; minimumSize: 300@100.
9    topView addSubview:
10   (SelectionInListView on: aBrowser printItems: false oneItem: false
```

```

11    aspect: #methodName change: #methodName: list: #methodList
12    menu: #methodMenu initialSelection: #methodName)
13    in: (0@0 extent: 1.0@0.25) borderWidth: 1.
14    topView addSubview:
15      (CodeView on: aBrowser aspect: #text change: #acceptText:from:
16        menu: #textMenu initialSelection: sel)
17    in: (0@0.25 extent: 1@0.75) borderWidth: 1.
18    topView controller open

```

Seuraavaksi käydään rivi kerrallaan läpi mitä yllä esitettyssä ohjelmakoodissa tapahtuu. Mallin luonnin jälkeen [5] luodaan viite uudelle *BrowserView* -luokan instanssille [6]. *BrowserView* on peritty *StandardSystemView* -luokasta. Seuraavaksi määritellään malli ja ohjain sekä muuttujat näkymän otsikolle ja koolle [7]. Jos ohjainta ei määritellä erikseen, käytetään näkymän *defaultController* metodia. Riveillä [7-11] luodaan alanäkymä *SelectionInListView* ja riveillä [12-15] luodaan toinen alanäkymä *CodeView*. Lopuksi [16] avataan ohjain, joka käynnistää ikkunoiden piirtämisprosessin.

Näkymät saattavat tarvita myös oman protokollan itsensä näyttämiseen. Kun malli ilmoittaa muutoksestaan, *update* -metodi näkymässä kutsuu *display*, joka puolestaan kutsuu *displayBorder*, *displayView* ja *displaySubviews*. Jos näkymä tarvitsee erityistä käyttäytymistä itsensä näyttämiseen, se toteutetaan edellämainituissa metodeissa. Muuten käytetään pääluokasta perittyjä ominaisuuksia (Steve 1992). Monet näkymät käyttävät myös erilaisia transformaatio-instansseja, joilla hallitaan esimerkiksi näkymän skaalausta ruudulla. Tähän ei kuitenkaan perehdytä sen enempää, koska ne menevät tutkimuksen rajojen ulkopuolelle.

8.2.2 Ohjain (Controller)

Ohjaimen tehtävänä on ottaa vastaan syötteitä sekä koordinoita malleja ja näkymiä saatujen syötteiden perusteella. Sen tulee myös kommunikoida muiden ohjaimien kanssa. Teknisesti ohjaimessa on kolme viitemuuttujaa: malli, näkymä ja sensori (sensor). Sensorin tehtävänä on toimia rajapintana syötelaiteiden sekä ohjaimen välillä. Sensori mallintaa syötelaiteiden käyttäytymistä ja muuttaa ne ohjaimen ymmärtämään muotoon.

Ohjaimien tulee käyttäytyä siten, että vain yksi ohjain ottaa vastaan syötteitä kerrallaan. Esimerkiksi näkymät pystyvät esittämään informaatiota rinnakkain monen näkymän kautta, mutta käyttäjän toimintoja tulkitsee aina vain yksi ohjain. Ohjain on siis määritelty käyttäytymään siten, että se osaa tietyn signaalin perusteella päättää tuleeko sen aktivoida itsensä vai ei. Ohjain sisältää toiminnallisuuden jonka perusteella se pystyy päättämään tuleeko hallinta pitää itsellä vai luovuttaa eteenpäin (**krasner_desc**). Ohjainten ylimmällä tasolla on *ControlManager*, joka kysyy jokaiselta päänäkymään liitetystä ohjaimelta erikseen, haluaako tämä ottaa hallinnan. Jos ohjaimen näkymä sisältää kursorin, vastaa ohjain kutsuun myönteisesti, jolloin kyseinen ohjain saa hallinnan. Hallitsevan ohjaimen näkymä kysyy seuraavaksi mahdollisten alanäkymien ohjaimilta samalla tavalla haluaako jokin ohjaimista hallinnan itselleen. Jos myönteisesti vastaava ohjain löytyy, ottaa se uuden hallinnan. Tätä prosessia jatkamalla löydetään matalimman tason näkymä ja sen ohjain ottaa lopullisen hallinnan. Ohjain pitää hallinnan itsellään niin kauan kunnes kursoria liikutetaan näkymän rajoista ulos. Ainoastaan se jonka kohdalla kursori on, vastaa kutsuun ja tuolloin ottaa hallinnan. Näkymillä on oikeus kysyä alanäkymiensä ohjaimia. Ohjaimien tehtävänä on kysyä omalta näkymältään onko kursori niiden päällä.

Krasner määrittelee seuraavat metodit, joiden avulla ohjaimet viestivät (**krasner_desc**):

isControlWanted - Tuleeko ohjaimen ottaa hallinta.

isControlActive - Onko ohjain aktiivinen.

controlToNextLevel - Luovutetaan hallinta seuraavalle ohjaimelle.

viewHasCursor - Onko ohjaimen näkymässä hiiren kursori.

controlInitialize - Kun ohjain on saanut hallinnan, alustetaan se.

controlLoop - Lähettää *controlActivity* -viestejä niin kauan, kuin ohjaimella on hallinta.

controlTerminate - Lopettaa ohjaimen hallinnan.

Kun ohjain saa hallinnan itselleen, kutsuu se *startUp* -metodia, joka puolestaan kutsuu seuraavia metodeja: *controlInitialize*, *controlLoop* ja *controlTerminate*. Metodit voidaan ylikirjoittaa, jolloin saavutetaan jokin haluttu ominaisuus kyseisessä vaiheessa. Esimerkiksi *controlInitialize* ja *controlTerminate* määräävät mitä tehdään, kun ohjain saa hallinnan tai luovuttaa sen eteenpäin. Ohjaimen hallinnan aikana kutsutaan *controlLoop* -metodia, joka taas kutsuu *controlActivity* -metodia niin kauan kuin ohjaimella on hallinta. Metodi *controlActi-*

vity määrää ohjaimen toiminnan hallinnan aikana **krasner_desc**

8.2.3 Esimerkkiohjelma

Seuraavaksi esitellään Dortmundin yliopistossa kirjoitettu yksinkertainen esimerkkiohjelma Smalltalkilla siitä miten MVC:n toteutus tuodaan sovellukseen käytännössä. Ohjelmakoodi löytyy myös Krasnerin artikkelista **krasner_desc** Ohjelmassa toteutetaan yksinkertainen laskuri-ohjelma, joka käyttää MVC-arkkitehtuuria toteutuksessaan. Ohjelmassa esitellään mallina *Counter* -luokka ja näkymänä *CounterView* -luokka. *Counter* perii mallin ominaisuudet ja toimii ohjelmassa yksinkertaisen kokonaisluku-muuttujan ylläpitäjänä. *CounterView* perii näkymän ominaisuudet ja esittää mallin arvon ruudulla. Ohjaimena toimii *CounterController* -luokka, joka perii ohjaimen käyttäytymisen. Ohjain tarjoaa sovellukselle painikkeet, joista voidaan vähentää tai lisätä laskurin arvoa.

Määritellään ensiksi *Counter* -luokka, joka peritään *Model* -luokasta.

```
1 Model subclass: #Counter
2   instanceVariableNames: 'value'
3   classVariableNames: ''
4   poolDictionaries: ''
5   category: 'Demo—Counter'
```

Seuraavaksi määritellään *Counter*-luokalle metodeita, jotka määrävät laskuriarvon alustamisen sekä muokkaamisen.

```
1 Counter methods For: 'Initialize—release'
2 Initialize
3   "Aseta alkuarvoksi 0"
4   self value: 0
5 Counter methodsFor: 'accessing'
6 value
7   "Palauta mallin arvo"
8   ↑value
9 value: aNumber
```

```

10  "Aseta mallin arvo"
11  value <– aNumber.
12  self changed "to update displayed value"
13  Counter methodsFor: 'operations'
14  decrement
15  "Vähennä mallin arvoa yhdellä."
16  self value: value – 1
17  Increment
18  "Lisää mallin arvoa yhdellä."
19  self value: value + 1

```

Lisätään luokkaan metodi, jolla itse luokasta saadaan muodostettua instanssi.

```

1  Counter class methodsFor: 'instance_ creation'
2  new
3  "Palauta uusi instanssi luokasta"
4  ↑super new initialize

```

Seuraavaksi määritellään ohjain (*CounterController*), joka peritään *Controller* luokasta. Luodaan myös ohjaimelle metodit, joiden avulla ohjataan mallia sekä näkymää. Metodeissa toteutetaan valikko, joka tarjoaa mahdollisuuden joko vähentää tai lisätä laskurin arvoa. Kaikki *CounterController* -luokassa käytetyt määrittelemättömät muuttujat peritään ylläluokasta.

```

1  Mouse MenuController subclass: #CounterController
2  instanceVariableNames: ' _ '
3  classVariableNames: ' _ '
4  poolDictionaries: ' _ '
5  category: 'Demo–Counter'
6  CounterController methodsFor: 'initialize–release'
7  initialize
8  "Alusta valikko, jossa on mahdollisuus vähentää tai
9   lisätä mallin arvoa"
10  super initialize.

```

```

11 Self yellowButtonMenu: (PopupMenu labels:
12     'Increment\Decrement' withCRs)
13     yellowButtonMessages: #(increment decrement)
14 CounterController methodsFor: 'menu_messages'
15 decrement
16     "Vähennä mallin arvoa yhdellä."
17     self model decrement
18 increment
19     "Lisää mallin arvoa yhdellä"
20     self model increment
21 CounterController methodsFor: 'control_defaults'
22 isActive
23     "Ota hallinta kun sinistä nappia ei paineta"
24     ↑super isActive & sensor blueButtonPressed not

```

Määritetään näkymä (*CounterView*), joka peritään *View* -yliluokasta. Määritetään myös näkymälle metodit, joiden avulla näytetään mallin tila ruudulla.

```

1 View subclass: #CounterView
2     instanceVariableNames: ''
3     classVariableNames: ''
4     poolDictionaries: ''
5     category: 'Demo-Counter'
6
7 CounterView methodsFor: 'displaying'
8 displayView
9     "Näytä mallin arvo näkymässä"
10    | box pos displayText |
11    box ← self insetDisplayBox.
12    "Asettele teksti näkymään. Asettelu ei
13    ole tutkielman kannalta oleellista."
14    pos ← box origin + (4 @ (box extent y / 3)).

```

```

15     displayText ← ('value:', self model value printString)
16         asDisplayText.
17     displayText displayAt: pos

```

Määritellään *update* -metodi, jotta näkymä pystyy päivittämään itsensä. Metodia kutsutaan yleensä mallin tilan muuttuessa.

```

1 CounterView methodsFor: 'updating'
2 update: aParameter
3     "Yksinkertaisesti päivitä näyttö uudestaan"
4     self display

```

Luodaan myös metodi, joka palauttaa näkymään liitetyn ohjaimen.

```

1 CounterView methodsFor: 'controller_access'
2 defaultControllerClass
3     "Palauta näkymään rekisteröity ohjain"
4     ↑CounterController

```

Lopuksi tarvitaan metodi, joka luo uuden näkymän sekä rekisteröi mallin ja ohjaimen itseensä. Näkymä näyttää ruudulta samalta kuin kuvassa 2.

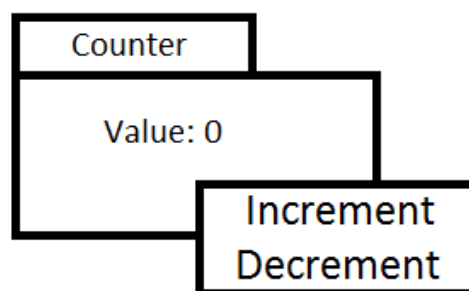
```

1 CounterView class methodsFor: 'instance_creation'
2 open
3     "Avaa näkymän uudelle laskurisovellukselle. Tässä
4     metodissa nähdään kuinka näkymä huolehtii mallin
5     rekisteröinnistä sekä nähdään kuinka näkymiä voi
6     olla useita sisäkkäin."
7     | aCounterView topView |
8     "Luo laskurinäkymälle uusi näkymä, joka näyttää
9     laskurin arvon"
10    aCounterView ← CounterView new
11    "Asetetaan malliksi Counter –luokan instanssi"
12    model: Counter new.

```



```
13   aCounterView borderWidth: 2.  
14   aCounterView insideColor: Form white.  
15   "Asetetaan ylimmäksi näkymäksi StandardSystemView  
16   —luokan instanssi, joka vastaa perinteistä  
17   ikkunointimallia"  
18   topView ← StandardSystemView new  
19       label: 'Counter'.  
20   topView minimumSize: 80@40.  
21   "Lisätään edellä luotu laskurinäkymä ylinäkymän  
22   alanäkymäksi"  
23   topView addSubview: aCounterView.  
24   "Käynnistetään ohjain"  
25   topView controller open
```



Kuvio 3. Kuva CounterView -näkymästä **krasner_desc**

9 Sovelluskehukset

Sovelluskehukset ovat suosittuja, koska ne tarjoavat uudelleenkäytettäviä ratkaisuja erilaisiin ongelmiin sovelluskehityksessä. Toimialueesta riippumatta sovelluskehiksi tulisi käyttää hyväksi kirjoittaessa monimutkaisia sovelluksia. Sovelluskehys tuo sovellukseen tason, jossa sovelluksen osat on abstrahoitu erilaisilla luokilla sekä rajapinnoilla, joita voidaan käyttää uudelleen sovelluksen eri osissa. Sovelluskehys ei ole vain kokoelma rajapintoja ja kirjastoja (Sheikh I. Ahamed ja Pezewski 2008). Tärkein ero sovelluskehiksen ja kirjaston välillä on se, että kirjaston ohjelmakoodi kutsutaan aina kehittäjän toimesta. Sovelluskehityksessä taas kehittäjän ohjelmakoodia kutsutaan aina sovelluskehiksen toimesta (**pyramid_intr**).

Sovelluskehys ei myöskään generoi koodia. Se käyttää erilaisia komponentteja ja kirjastoja luodakseen infrastruktuurin, jonka päälle voidaan rakentaa sovelluksia sovelluskehiksen ehdoilla. Sovelluskehiksen käyttäminen myös rajoittaa sovelluksen rakennetta ja pakottaa sovelluksen toteuttamaan asioita tietyin ehdoin. Rajoitusten ansiosta sovelluskehittäjä voi keskittyä toimialueeseen liittyviin ongelmiin välittämättä koko sovelluksen yksityiskohtaisesta toteutuksesta (Sheikh I. Ahamed ja Pezewski 2008).

Web-sovelluskehukset ovat sovelluskehiksiä, jotka tarjoavat ratkaisuja helpottamaan web-sovellusten toteuttamista. Web-sovelluskehityksissä käyttöliittymä näytetään käyttäjille selaimen välityksellä. Sovellus ajetaan joko serverillä tai suoraan käyttäjän selaimessa. Sovellus määrittää käyttöliittymän sivujen järjestyksen, sisällön sekä mahdollisten toimintojen esittämisen käyttäjälle, jonka kautta käyttäjä voi vaikuttaa serverillä sijaitsevaan sovellukseen (Kourie 2008).

Yleisimmät teknologiat mitä web-sovelluskehukset tarjoavat ovat rajapinta tietokannalle, template-moottori sekä mahdollisuus käsitellä http-pyyntöjä ohjelmakoodissa. Tietokantarakapinnalla tuodaan sovelluskehitykseen taso, jonka avulla helpotetaan kommunikointia tietokannan kanssa. Yleisimmin käytetty ohjelmointitekniikka tähän on ORM (Object-Relational-Mapping), jolla muunnetaan dataa tietokannan ja ohjelmakoodin välillä. Ohjelmoijalle tämä näkyy ns. virtuaalisena olio-tietokantana, jonka avulla voidaan lukea sekä muokata tietokantaa kutsuilla ohjelmakoodista. Tällöin suoria kyselyitä tietokantaan ei tarvita (Shahram

Ghandeharizadeh 2014). Seuraavassa esimerkissä esitellään miten Django ORM:ia käytetään.

```
from django.db import models
```

```
class Blog(models.Model):  
    name = models.CharField(max_length=100)  
    tagline = models.TextField()
```

```
from blog.models import Blog
```

```
b = Blog(name='Beatles_Blog', tagline='All_the_latest_Beatles_news.')
```

```
b.save()
```

Esimerkissä luodaan luokka, jolla määritellään taulu SQL-tietokantaan. Model-luokasta periminen mahdollistaa luokasta luotujen instanssien tallentumisen tauluun. Lisäksi luokalle voidaan määrittää attribuutteja, jotka käyttävät Django tarjoamia kenttiä. Kentät vastaavat tietokannassa olevia data-tyyppejä. Lopullinen tallentuminen tapahtuu kutsumalla instanssin save() -metodia.

```
beatles_blog = Blog.objects.get(name="Beatles_Blog")
```

Yllä oleva rivi hoitaa kyselyn nimen perusteella blog -nimisestä taulusta, luo sen kenttien perusteella instanssin Blog -luokasta ja tallentaa sen muuttujan arvoksi.

Template-moottori on teknologia HTML-sivujen tuottamiseen, jolla generoidaan dynaamisia HTML-sivuja yhdistämällä ohjelmalogiikkaa sekä HTML-kieltä. Alla on esitelty Jinja2-kieli, jota käytetään template-kielenä Flask:ssa.

```
1 <title>{ % block title % } { % endblock % } </title>
2 <ul>
3   { % for user in users % }
4     <li><a href="{{ user.url }}">{{ user.username }}</a></li>
5   { % endfor % }
6 </ul>
```

Yllä esitellyssä esimerkissä luodaan HTML-sivu, jossa tulostetaan otsikko sekä lista url-osoitteita sekä käyttäjänimiä jokaista käyttäjää kohden (Ronacher 2008).

Web-sovelluskehikset voidaan jakaa serveri-pohjaisiin ja selain-pohjaisiin sovelluskehikseen. Serveri-pohjaisissa web-sovelluskehiksissä sovelluksen tilaa hallitaan serverin puolella. Tuolloin käyttöliittymää perustuu tilaan, mikä on sillä hetkellä serverin puolella. Selain-pohjaisissa sovelluskehiksissä sisältö muuttuu selaimen sisällä käyttäjän päässä (Kourie 2008). Tässä tutkimuksessa käsiteltävät sovelluskehikset ovat Pythonilla toteutettuja serveri-pohjaisia sovelluskehiksiä. Käsiteltävät kehykset ovat Django, Pyramid ja Flask. Django on käsiteltävistä sovelluskehyksistä kaikkein monoliittisin ja tarjoaa eniten ominaisuuksia valmiiksi asennettuna. Pyramidissa taas on vähemmän ominaisuuksia suoraan asennettuna, jonka kautta se pyrkii antamaan käyttäjälle enemmän valinnanvaraa erilaisten teknologioiden valitsemiseen. Flaski taas on mikro-sovelluskehys, joka tarjoaa kaikkein vähiten työkaluja web-kehitykseen näistä kolmesta. Sovelluskehysten koko kertoo myös niiden toteutuksesta: Django (1.7.4) 36 megabittiä, Pyramid (1.5.3) 5.6 megabittiä ja Flask (0.10.1) 1.2 megatavua. Sovelluskehysten koot tarkastettiin komentoriviltä käyttäen linuxin `du -työkalua`. Kaikkia kolmea sovelluskehystä voidaan laajentaa erilaisilla laajennoksilla.

9.1 Pyramid

Pyramid on Python-pohjainen web-sovelluskehys, jonka tehtävänä on helpottaa web-kehitystä tarjoamalla kehittäjälle valmiita työkaluja avuksi kehitykseen. Pyramid on suunniteltu siten, että kehittäjän ei tarvitse tietää suuria määriä erilaisia malleja ja tekniikoita pystyäkseen tuottamaan web-sovelluksia. Se ei myöskään pakota käyttämään kehityksessä mitään erityistä tekniikkaa, vaan pyrkii olemaan mahdollisimman yksinkertainen ja helposti laajennettavissa

erilaisiin käyttötarkoituksiin. Laajentamisella tarkoitetaan erilaisten lisäosien liittämistä Pyramidiin. Yksinkertaisuuden ja mimimaalisuuden ansiosta se on myös nopeampi kuin monet muut Python-pohjaiset web-sovelluskehikset. Tämä johtuu Pyramidin poikkeuksellisen pienestä kutsupinosta ajamisen aikana (**pyramid_intr**).

Pyramid sai alkunsa Pylons -projektista syyskuussa vuonna 2005, jolloin jo yli 30 Python-sovelluskehystä kilpaili käyttäjistä. Ben Bangert ja James Gardner alkoivat yhdessä kehittää sovelluskehystä, josta tuli lopulta Pylons. Alunperin Pylons oli muokattu Myghty Python Templating Framework:n pohjalta tarjotakseen MVC-pohjaisen web-sovelluskehiksen. Myghty -sovelluskehystä ajettiin mod_pythonin päällä, mutta Pylonsin pyrkimys oli käyttää WSGI:tä hyödyntämään joustavaa komponenttipohjaista lähestymistapaa web-sovelluksissa (**pyramid_history**). Pylons projektin tarkoituksena ei ole keskittyä yhden yksittäisen web-sovelluskehiksen kehittämiseen vaan tarjota kokoelma erilaisia teknologioita (Project 2010a). Vuonna 2008 Pyramid tunnettiin nimellä repoze.bfg. Joulukuun alussa tapahtui ohjelmakoodin uudelleen nimeäminen ja ominaisuuksien lisääminen sekä poistaminen (Project 2010b).

Koska Pyramid pyrkii tarjoamaan vain välttämättömimmät työkalut web-sovelluksien kehittämiseen, sen kehittäjät ovat päätyneet web-kehityksessä neljään yleisimpään ongelmaan ja tarjoavat niihin ratkaisun Pyramidissa:

URL Mapping - URL:ien liittäminen ohjelmakoodiin.

Template - Tuodaan sovelluksen näkymä selaimelle käyttäen template-kieltä, jolla määritetään näkymän rakenne. Kieli on usein HTML:än mukana tuotuja loogisia ilmaisuja, joiden perusteella template-engine rakentaa HTML-sivun selaimelle. Templatejen avulla pystytään erottamaan käyttöliittymä sovelluslogiikasta tehokkaasti.

Security - Perinteiset tietoturvaongelmat tulee olla ratkaistuna valmiiksi jo sovelluskehiksessä. Tämä ei kuitenkaan tarkoita sitä, että kehittäjä voisi täysin unohtaa tietoturvan merkityksen.

Static Assets - Staattisten resurssien jakaminen niille tarkoitettuihin paikkoihin tiedostorekenteessa.

Yllä määriteltyjen neljän ongelman lisäksi Pyramid tarjoaa lisäosien kautta monia erilaisia työkaluja, joiden avulla pystytään laajentamaan sen ominaisuuksia (**pyramid_intr**). Tutkiel-

man aiheen rajauksen vuoksi ei kuitenkaan käydä läpi yksityiskohtaisemmin Pyramidin toteutusta ja siihen liitettävissä olevia lisäosia, vaan keskitytään tarkastelemaan MVC:n toteutusta Pyramidissa.

9.2 Django

Django on web-sovelluskehys, joka sai alkunsa kehitysryhmässä Kansaksen osavaltiossa Yhdysvalloissa 2003, kun web-kehittäjät Adrian Holovaty ja Simon Willison alkoivat käyttää Pythonia web-kehityksessä. The World Online -ryhmä (WO), joka oli vastuussa muutamasta paikallisesta uutissivustosta, menestyivät ympäristössä, jossa oli tiukat aikarajat. Journalistit vaativat ominaisuuksien ja kokonaisten sovelluksien valmistumista muutamassa päivässä ja joskus jopa tunneissa. Holovaty ja Willison kehittivät web-sovelluskehiksen, jonka avulla he pystyivät vastaamaan journalistisen ympäristön haasteisiin. Kesällä 2005 he saivat kehitettyä sovelluskehiksen siten, että se oli käytössä suurimmassa osassa World Onlinen sivustoja. Tuolloin mukaan kehitykseen tuli Jacob Kaplan-Moss. Kehittäjät päättivät julkaista heinäkuussa 2005 sovelluskehiksen nimellä Django jazz-kitaristi Django Reinhardtin mukaan (Adrian Holovaty 2009a).

Django tarjoaa samat välttämättömät työkalut kuin Pyramidissa. Se tarjoaa myös ylläpitäjille suunnatun työkalun, josta voidaan hallita sovellusta käyttöliittymätasolta. Lisäksi se tarjoaa lomake-työkalut, käyttäjätasoisien autentikaation sekä tietokanta-abstrahoinnin. Tietokanta-abstraktiolla tarkoitetaan sisäänrakennettua virtuaalista ympäristöä tuomaan yhteys tietokantaan olio-ohjelmoinnin kautta (Object-relational mapping)(Adrian Holovaty 2009b). Siinä missä Pyramid pyrkii tarjoamaan kehittäjille valinnanvaraa erilaisten komponenttejen suhteen, Django tarjoaa kokonaisvaltaisen ratkaisun sisältäen kaikki tarvittavat työkalut web-sovellusten rakentamiseen. Djangoon on tarjolla myös paljon erilaisia paketteja täydentämään sitä.

9.3 Flask

Flask on mikro-sovelluskehys (micro-framework), jossa sovelluskehiksen lähdekoodi on pidetty mahdollisimman minimalistisena. Sen tarkoituksena ei ole tehdä valintoja kehittäjän

puolesta millaisia teknologioita tämän tulisi käyttää. Ne teknologiat mitä Flask tuo mukanaan, voidaan korvata toisella vastaavalla teknologialla. Esimerkiksi flask tuo mukanaan Jinja2 template-kielen, joka voidaan korvata millä tahansa muulla template-kielellä.

Flaski ei tuo mitään lomake-työkaluja, tietokanta-abstrahointeja tai mitään missä ulkoinen kirjasto ei pystyisi kyseistä teknologiaa toteuttamaan. Flask tukee laajennoksia, joilla sitä voidaan laajentaa erilaisilla teknologioilla. Tällaisia ovat esimerkiksi erilaiset lomakekirjastot ja tietokanta-abstrahoinnit. Flask on tämän tutkimuksen sovelluskehysistä minimaalisin eikä se toteuta minkään laista MVC:hen liittyvää pohjaa. Tämän vuoksi sitä käytetään hyväksi tutkiessa miten MVC tulisi toteuttaa web-sovelluskehukseen, jotta sen alkupe-
räiset määritelmät täyttyisivät (Ronacher 2013).

10 MVC:n vaatimukset

Vaikka MVC:tä ei ole tarkoitettu alunperin web-sovelluksiin, voivat ne hyötyä MVC:n arkkitehtuurista. Suurin ongelma MVC:n käyttämisessä web-sovelluskehyksissä on palvelimen (server) ja asiakkaan (client) välinen ositus. Näkymä näytetään aina asiakkaan selaimessa sen omalla päätelaitteella HTML-kielellä. Malli ja Ohjain taas voivat olla ositettu teoriassa miten vain asiakkaan ja palvelimen välillä. Web-sovelluksissa kehittäjä pakotetaan osioimaan sovellus. MVC:n tulee olla riippumaton osiinnista. Osioinnin ei tule määrittää sovelluksen arkkitehtuuria Avraham 2001.

Pyramidin sovellusarkkitehtuuri on toteutettu käyttäen pohjana MVC-arkkitehtuuria. Tämä ei kuitenkaan tarkoita sitä, että Pyramid toteuttaisi MVC:n teknisesti sellaisena kuin esimerkiksi Krasner määrittelee **Krasner:desc** Tärkeimpänä vaatimuksena MVC:n toteutukselle on se, että näkymä ja ohjain luovat rajapinnan, jonka kautta käyttäjä keskustelee mallin kanssa. Malli ei saa olla suorassa yhteydessä käyttäjään **Reenskaug:tools** Mallin tulee olla riippumaton näkymästä ja ohjaimesta. Sen tulee myös hallita sovelluksen tilaa sekä pystyä antamaan informaatiota sovelluksen tilasta **Burbeck** Näkymän on keskusteltava mallin kanssa sekä hoitaa mallilta saadun datan graafinen näyttäminen Krasner 1979, s.1. Ohjain puolestaan ottaa vastaan syötteitä ja lähettää viestejä tämän perusteella näkymälle ja mallille **Burbeck** Lisäksi Mallin tulee pystyä kommunikoimaan muutoksistaan ohjaimelle ja näkymälle. Näiden rajoituksien perusteella pystytään vastaamaan tutkielmassa siihen, toteuttaako Pyramid MVC-arkkitehtuurin vai ei.

11 MVC & Pyramid

Pyramidin kehittäjien dokumenteissa esitellään Pyramid MVC-kehiksenä, mutta samalla myös kyseenalaistetaan tämä väite. Erityisesti mallin ja ohjaimen määritelmä puuttuu **Pyramid:intr**. Tässä osiossa käsitellään Pyramidia MVC:n näkökulmasta. Tarkastelua varten toteutetaan Pyramidilla vastaava laskuri-sovellus kuin Krasnerin julkaisussa käyttäen Pyramidin tarjoamia ominaisuuksia **Krasner:desc**. Sovellus rajataan käyttämään SQL-tietokantaa mallin datan tallennukseen sekä *URL dispatch* -tekniikkaa **urldispatch; sql**. Sovelluksen pohjalta tarkastellaan MVC:n kannalta kolmea oleellista tiedostoa: *views.py*, *models.py* ja *template.pt*. Näin tutkielman tarkastelu pystytään rajaamaan mahdollisimman pienelle alueelle, jolloin tutkielma on helpompi keskittää tarkastelemaan yksittäistä tekniikkaa.

Tässä osiossa tarkastellaan Pyramid-sovelluksen tiedostoja sekä niiden sisältöjä MVC-komponentteina. Erityisesti keskitytään ohjaimen ja näkymän toteukseen. Samalla tutkitaan voidaanko Pyramid havaintojen perusteella luokitella MVC:n toteuttavaksi sovelluskehikseksi. Koska Pyramidissa MVC:n määrittely on hyvin epävakaa pohjalla, tulee tehdä selvästi mitä ominaisuuksia komponenteilta vaaditaan. Esimerkiksi *views.py* ja *template.pt* -tiedostot ovat tarkoitettu toimimaan yhdessä näkymänä, jolloin ohjaimen toteuttamat tehtävät sisällytettäisiin näkymään. MVC:tä tutkiessa täytyy sovelluksen komponentit kuitenkin jakaa kolmeen osaan. Tiedosto *views.py* sisältää paljon ohjaimelle yhteisiä piirteitä, joten se erottuu selvästi *template.pt* -tiedostosta. Tutkielmassa oletetaan, että *models.py* sisältää mallin, *views.py* ohjaimen ja *template.pt* näkymän.

11.0.1 Tiedostojen tarkastelu

```
1 # Tiedosto: models.py
2 class Counter(Base):
3     __tablename__ = 'counter'
4
5     # Asetetaan mallille attribuutit, jotka
6     # vastaavat mallin tilasta.
7     id = Column(Integer, primary_key=True)
```

```

8     name = Column(Unicode(255), unique=True)
9     value = Column(Integer)
10
11     # Määritellään luokalle konstruktori,
12     # joka saa parametreiksi nimen ja alkuarvon.
13     def __init__(self, name, value):
14         self.name = name
15         self.value = value
16
17     def increment(self):
18         self.value += 1
19
20     def decrement(self):
21         self.value -= 1
22
23 # Luodaan instanssi mallista ja rekisteröidään
24 # se sovellukseen.
25 def populate():
26     session = DBSession()
27     model = Counter(name=u'counter', value=0)
28     session.add(model)

```

Models.py -tiedosto sisältää malliin liittyvän ohjelmakoodin. Jokaiselle mallille luodaan aina oma luokkansa, jossa määritellään mallin ominaisuudet. Malli rekisteröidään *populate* -funktion kautta, jota kutsutaan Pyramidin toimesta. Ohjelmakoodista nähdään, että malli ei luo minkäänlaista riippuvuutta näkymään tai ohjaimeen. Se myös pitää huolen datan käsittelystä. Malli on siis Pyramidissa itsenäinen komponentti, joka huolehtii sovelluksen tilasta. Tämän perusteella malli toteutuu Pyramidissa MVC-arkkitehtuurin mukaisesti. *Views.py* -tiedostossa määritellään näkymän ohjelmakoodi. Pyramidissa on konkreettisesti määritelty ohjelmakooditasolla vain malli ja näkymä. Jokaista näkymää kohden on oma funktio, joka ottaa vastaan *request*-olion. Request-oliossa tuodaan sovellukselle kaikki tieto käyttäjäs-

tä ja sovelluksen viesteistä. Tämän perusteella tulkitaan request-oliossa tuotu data käyttäjän syötteiksi. Vaikka *views.py* nimetään Pyramidissa näkymäksi, on se toteutukseltaan hyvin lähellä ohjainta. Tästä syystä tarkastellaan funktion toteutusta mahdollisena ohjaimena. Tästä eteenpäin puhuttaessa ohjaimesta Pyramidissa, tarkoitetaan sillä *views.py* -tiedoston sisältä-mää funktiota.

```
1  # Tiedosto: views.py
2  @view_config(route_name='counter_view',
3               renderer='templates/counter.pt')
4  def counter_view(request):
5      dbsession = DBSession()
6
7      # Rekisteröidään malli.
8      counter = dbsession.query(Counter).filter(
9          Counter.name==u'counter').first()
10     try:
11         request.params['minus'].
12         counter.decrement()
13     except KeyError:
14         pass
15
16     try:
17         request.params['plus']
18         counter.increment()
19     except KeyError:
20         pass
21
22     # Palautetaan laskurin arvo, joka tulkitaan
23     # ja näytetään template.pt -tiedostossa
24     return {'value': counter.value}
```

Määritellään funktiolle URL-osoite sekä liitetään siihen template-tiedosto (2). Tämän jäl-

keen rekisteröidään malli mukaan funktioon (8). Koska tarkastelemme funktiota ohjaimena, voimme tulkita templatien näkymäksi. Tällöin funktioon rekisteröidään malli sekä näkymä, jolloin rekisteröinnin puolesta se toteuttaa ohjaimelle tarkoitetut ominaisuudet MVC:ssä. Funktioon lisätään myös toiminto laskurin vähentämiselle. Mallin *value* -arvoa muutetaan, kun request-oliosta löytyy tietty parametri. Funktio palauttaa paluuarvona mallin arvon, joka tuodaan käsiteltäväksi templateen. Funktio siis ottaa vastaan syötteitä ja niiden perusteella lähettää viestejä mallille sekä näkymälle. Tämän perusteella todetaan, että se täyttää rajauksessa määrättyt ohjaimen ominaisuudet.

Templatessa yhdistetään HTML-merkkäuskieli ja sovelluksen ohjelmakoodi. Tästä generoidaan HTML-sivu, joka näytetään selaimelle. Koska malli sekä ohjain on jo määritelty, täytyy selvittää täyttääkö template näkymälle määritellyt ominaisuudet.

```
1  <body>
2    <h1>${ value }</h1>
3    <form action=" . / " method=" get ">
4      <button type="submit" name="plus" value="plus">
5          Increment </button>
6      <button type="submit" name="minus" value="minus">
7          Decrement </button>
8    </form>
9  </body>
```

Templatessa luodaan lomake kahdelle painikkeelle, joista kumpikin lähettää lomakkeen eteenpäin *counter_view* -funktiolle. Lomakkeen tiedot tulevat funktiolle request-oliossa, joka sisältää tässä tapauksessa *plus*- tai *minus*-parametrin riippuen siitä kumpaa painiketta on painettu. Lomakkeen tiedot lähetetään samaan osoitteeseen (3), mistä sitä on alunperin kutsutukin. Eron tuo kuitenkin request-oliossa tuodut parametrit. Otsikossa (2) tuodaan näkyviin laskurin sen hetkinen arvo, joka saadaan tietoon ohjaimelta.

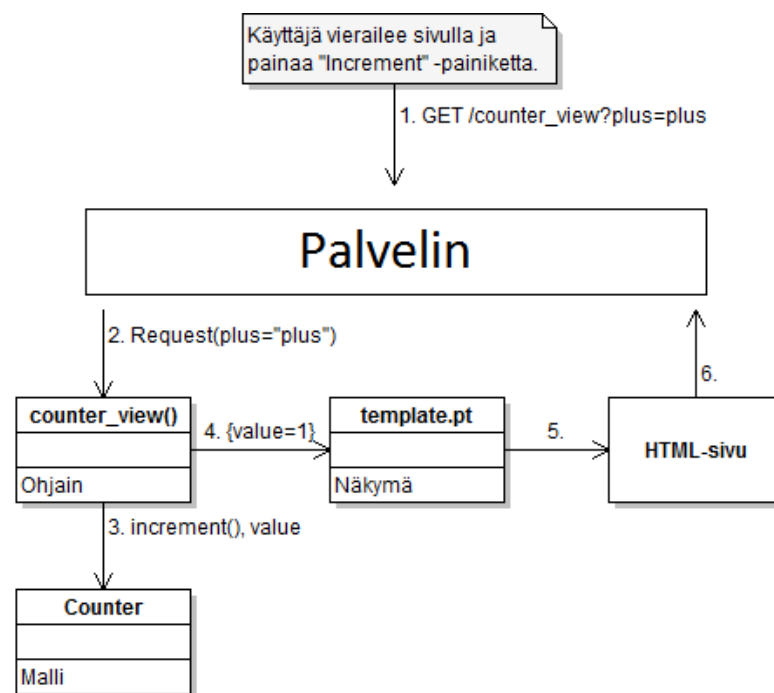
Templatessa hoidetaan sovelluksen graafinen puoli, joten se vastaa ominaisuuksiltaan näkymää.

Ongelmaksi muodostuu kuitenkin näkymän ja mallin välinen kommunikointi. Näkymä ei ole

yhteydessä malliin suoraan, vaan tarvitsee ohjaimen kautta tiedon mallin tilasta. Näkymä ei siis sellaisenaan toteuta sille asetettuja ominaisuuksia.

11.0.2 Sovelluksen toiminta

Alla olevassa kuvassa esitellään visuaalisesti miten laskurisovellus muodostaa HTML-sivun, kun käyttäjä painaa sivulla *Increment* -painiketta. Kuvan vaiheet toteutetaan numerojärjestyksessä alkaen ensimmäisestä.

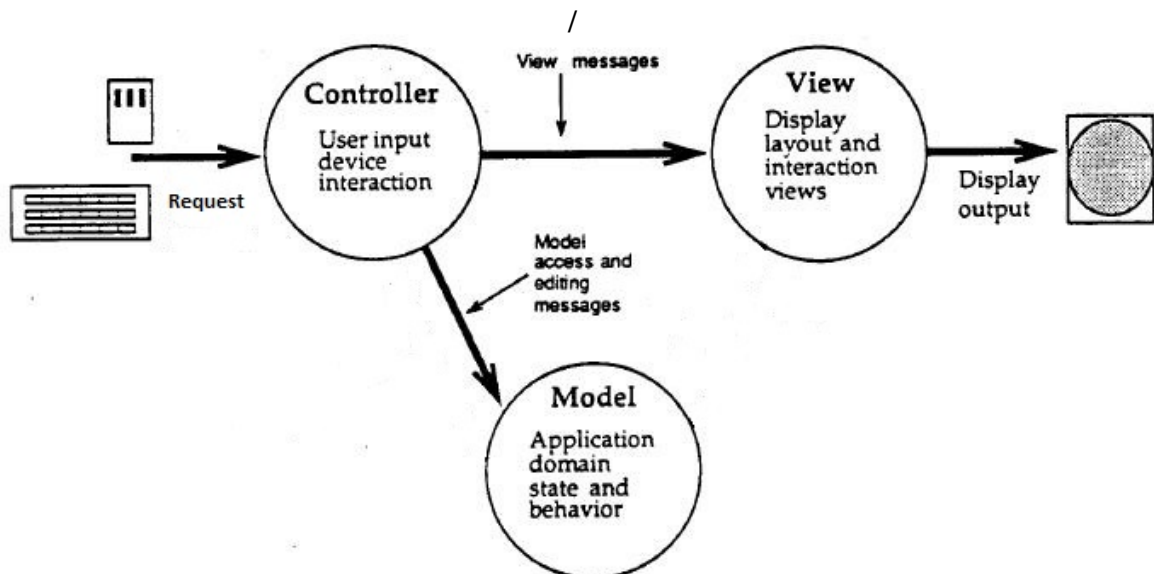


Kuvio 4. Laskurisovelluksen toiminta

- 1 Palvelimelta pyydetään HTTP-protokollan mukaisesti sivua *plus* -parametrilla.
- 2 Palvelin pyytää sovellukselta sivua. Parametri tuodaan sovellukselle *request*-oliiossa.
- 3 Ohjain käskee mallia muuttamaan tilaansa ja pyytämään samalla tiedon muutoksen jälkeisestä arvosta.
- 4 Ohjain palauttaa mallin arvon, joka käsitellään templatessa.
- 5 Templatessa generoidaan HTML-sivu, joka tuodaan palvelimelle vastauksena.

11.0.3 Yhteenveto

Malli sekä ohjain toteutuvat Pyramidissa MVC:n mukaisesti yksittäisinä komponentteina, mutta kommunikaatio näiden välillä ei mene MVC:n mukaisesti. Malli on itsenäinen komponentti, jolla ei ole riippuvuutta näkymään tai ohjaimeen. Se myös huolehtii sovelluksen käsittelemästä datasta ja vastaa tarvittaviin pyyntöihin. Se ei kuitenkaan pysty kertomaan ohjaimelle ja näkymälle omista muutoksistaan, koska kaikki muutokset tulevat näkymälle asti vastauksena vasta HTTP-pyyntöön mukana. Ohjain taas huolehtii request-oliossa tulevista syötteistä ja vaikuttaa malliin sekä näkymään. Näkymä toteuttaa sovelluksen graafisen näyttämisen selaimelle, mutta ei toteuta sille määrättyjä sääntöjä. MVC:ssä näkymän tarkoitus on kommunikoida suoraan mallin kanssa. Tämä ei kuitenkaan onnistu Pyramidissa, jossa näkymä on yhteydessä vain ohjaimeen. Pyramidin MVC-toteutusta tarkastellessa tulee ottaa huomioon yksittäisten komponenttien toteutus sekä niiden välinen yhteistyö. Yksittäiset komponentit toteutuvat Pyramidissa MVC-arkkitehtuurin mukaisesti, mutta niiden välinen yhteistyö ei toteudu.



Kuvio 5. Pyramidin kommunikointi MVC-komponenttien kesken. Kuva on muokattu Krasnerin esittelemästä kommunikointimallista **Krasner:desc**

Yllä esitetyssä kommunikaatiomallissa ohjain saa vastaan request-olion, jossa tuodaan kaikki tarvittava tieto käyttäjältä. Tämän perusteella ohjain ohjaa mallia sekä muuttujaa. Samalla se pyytää mallilta tietoa sovelluksen tilasta ja välittää tiedon näkymälle. Näkymä taas välittää

ohjaimen tuoman datan käyttäjälle graafisena.

Pyramidin MVC-komponenttien kommunikointi voidaan esittää käyttäen pohjana muokaten Krasnerin kommunikaatiomallia **Krasner:desc** Alla esitellyssä kuvassa havainnollistetaan, kuinka mallin ja näkymän kommunikaatio puuttuu täysin ja kaikki data tuodaan ohjaimen kautta. Lisäksi Malli ei pysty kommunikoimaan muutoksista suoraan ohjaimelle ja näkymälle vaan tarvitsee aina HTTP-pyyynnön.

Alunperin Pyramidin dokumentaatioissa kyseenalaistettiin mallin sekä ohjaimen toteutus **Pyramid:intr** Tutkimuksen pohjalta voidaan kuitenkin todeta, että ongelmaksi ei muodostu yksittäisien komponenttien toteutus, vaan komponenttien välinen kommunikointi. Erityisesti näkymän ja mallin yhteistyö jää kokonaan puuttumaan, jolloin saadaan ristiriita MVC:n alkuperäisen määritelmän kanssa Krasner 1979, s. 1. Tämä johtuu siitä, että Pyramidissa *views.py* -tiedoston sisältämä ohjelmakoodi on nimensä mukaisesti tarkoitettu näkymäksi ja *template.pt* -tiedosto katsotaan osaksi samaa komponenttia. Tutkimuksen tuloksien perusteella voidaan kuitenkin todeta, että *views.py* -tiedoston näkymä-funktio toteuttaa kaikki ohjaimelle määritellyt ominaisuudet. Template puolestaan hoitaa sovelluksen graafisen puolen, joten sen ominaisuudet ovat mahdollisimman lähellä näkymää. Template ei kuitenkaan riitä toteuttamaan näkymän ominaisuuksia, koska se on täysin riippuvainen ohjaimesta. Lisäksi Malli ei pysty kommunikoimaan suoraan näkymälle ja ohjaimelle ilman, että ohjain/näkymä pyytää Mallilta nykyistä tilaa.

(Avraham 2001)

12 Yhteenveto

Lähteet

Adrian Holovaty, Jacob Kaplan-Moss. 2009a. *Django's History*. <http://www.djangobook.com/en/2.0/chapter01.html#django-s-history>.

———. 2009b. *The Django Book*. <http://www.djangobook.com/en/2.0/>.

Arlington, Wilson Boulevard. 1981. “TRANSMISSION CONTROL PROTOCOL”. <https://www.ietf.org/rfc/rfc3439.txt>.

Avraham, Avraham Leff James T. Rayfield. 2001. “Web-Application Development Using the Model/View/Controller Design Pattern: 1.2, Web.Applications and the MVC Design”. <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=950428>.

Bestframeworks. 2009. *List of Python MVC-frameworks by bestwebframeworks.com*. <http://www.bestwebframeworks.com/compare-web-frameworks/python/>.

Bush, R. 1981a. “TRANSMISSION CONTROL PROTOCOL”. <https://tools.ietf.org/html/rfc793#section-1.1>.

———. 1981b. “TRANSMISSION CONTROL PROTOCOL”. <https://tools.ietf.org/html/rfc793#section-1.4>.

Carpentern, B. 1996. “Architectural Principles of the Internet”. <https://www.ietf.org/rfc/rfc1958.txt>.

Daly, Liza. 2007. *Next-Generation Web Frameworks in Python*. O'Reilly Media.

Deacon, John. 2009. “Computer Systems Development, Consulting and Training Model-View-Controller (MVC) Architecture”.

Fernández-Villamor, José Ignacio. 2008. “A comparison model for agile web frameworks”. <http://dl.acm.org/citation.cfm?id=1621087.1621101&coll=DL&dl=ACM&CFID=629380688&CFTOKEN=89109865>.

Foundation, Django Software. n.d. *FAQ: General*. <https://docs.djangoproject.com/en/1.10/faq/general/n>.

Kourie, Iwan Vosloo & Derrick G. 2008. “Server-Centric Web Frameworks: An Overview”.
<http://dl.acm.org/citation.cfm?id=1348246.1348247&coll=DL&dl=ACM&CFID=509430230&CFTOKEN=54230385>.

Krasner, Glenn E. Krasner & Stephen T. Pope. 1979. “Models-Views-Controllers”. <http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf>.

———. 1988. “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80”. <http://www.ics.uci.edu/~redmiles/ics227-SQ04/papers/KrasnerPope88.pdf>.

Okanovic, T. Mateljan, V. Okanovic'. 2011. “Designing a New Web Application Framework”.
<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5967262>.

Parc, Xerox. 1978. “MVC Xerox Parc”. <http://heim.ifi.uio.no/trygver/themes/mvc/mvc-index.html>.

———. 1979a. “Xerox Parc THE Original MVC reports”. http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf.

———. 1979b. “Xerox Parc THING-MODEL-VIEW-EDITOR”. <http://heim.ifi.uio.no/trygver/1979/mvc-1/1979-05-MVC.pdf>.

Project, Pylons. 2010a. *About Pylons*. <http://www.pylonsproject.org/about/pylons>.

———. 2010b. *About Pyramid*. <http://www.pylonsproject.org/projects/pyramid/about>.

———. 2010c. *Pyramid introduction*. <http://www.kemeneur.com/clients/pylons/docs/pyramid/narr/introduction.html>.

Ronacher, Armin. 2008. *Jinja2*. <http://jinja.pocoo.org/docs/dev/>.

———. 2013. *Flask*. <http://flask.pocoo.org/docs>.

Shahram Ghandeharizadeh, Ankit Mutha. 2014. "An Evaluation of the Hibernate Object-Relational Mapping for Processing Interactive Social Networking Actions". <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=950428>.

Sheikh I. Ahamed, Alex Pezewski, ja Al Pezewski. 2008. "Towards Framework Selection Criteria and Suitability for an Application Framework". <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1286492>.

Sommerville, Ian. 2010. *Software Engineering 9th edition*. Pearson.

Steve, Burbeck. 1992. "Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)". <http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>.

Xiao, X. 2000. "TCP Processing of the IPv4 Precedence Field". <https://tools.ietf.org/html/rfc793#section-1.1>.

Liitteet