Toni Haka-Risku

MVC-arkkitehtuurin toteutus web-sovelluskehyksissä

Tietotekniikan pro gradu -tutkielma

11. syyskuuta 2016

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Toni Haka-Risku

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

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: 35+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**). Työssä tutkitut sovelluskehykset ovat Django, Pyramid, Plone(TODO: jätetäänkö plone pois?) 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

Tähän voit kirjoittaa tutkielmasi esipuheen. Tutkielmissa on harvemmin esipuheita, mutta jos sen kirjoitat, pidä se lyhyenä (enintään sivu).

Esipuheen tulisi kertoa ennemminkin tutkielmaprosessista kuin tutkielman sisällöstä. Esimerkiksi jos tutkielman aiheen valintaan tai tekemiseen liittyy jokin erikoinen sattumus, voit siitä kertoa esipuheessa. Tapana esipuheessa on myös kiittää nimeltä mainiten tärkeimpiä tutkielman tekemisessä auttaneita ihmisiä – ainakin ohjaajia, puolisoa ja lapsia. (Yleensä perhe on auttanut vähintään tukemalla ja kannustamalla.)

Esipuhe kannattaa kirjoittaa minä-muodossa. Tavanomaista on myös allekirjoittaa se.

Jyväskylässä 11. syyskuuta 2016

Tutkielman tekijä

Termiluettelo

Sovellusarkkitehtuuri	Donald Knuthin 1977–1989 laatima eräajotyyppinenj ladonta-järjestelmä (**knuth86:_texbook**).
MVC	TEXin (**knuth86:_texbook**) päälle rakennettu rakenteisten kirjoitelmien ladontaan tarkoitettu järjestelmä (**lamport94:_latex**). Siitä on nykyään käytössä versio LATEX 2_ε.
Sovelluskehys	TEXin (**knuth86:_texbook**) päälle rakennettu rakenteisten kirjoitelmien ladontaan tarkoitettu järjestelmä (**lamport94:_latex**). Siitä on nykyään käytössä versio LATEX 2_ε.
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

Kuviot

Sisältö

1 Johdanto

1.1 Tausta

1.2 Tutkimuskysymys

MVC-arkkitehtuuri on saanut paljon huomiota web-sovelluskehyksien toteutuksissa ja useat web-sovelluskehykset ovat luokiteltu MVC-pohjaisiksi sovelluskehyksiksi (**mvcframeworks**). Se on kuitenkin alunperin tarkoitettu matalan tason käyttöliittymäsovellusten toteuttamiseen, jossa esimerkiksi hallitaan yksittäisiä näppäimistöltä 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 (**ibm_watson**). Myös Pyramid-sovelluskehyksen tekijät kyseenalaistavat MVC-arkkitehtuurin toteutuksen Pyramidissa ja uskovat MVC:n olevan sellaisenaan sopimaton web-ohjelmointiin, vaikka Pyramidin toteutus onkin hyvin lähellä alkuperäistä MVC:tä (**pyramid**). Django on vastaavasti toteutettu MVC:n pohjalta, mutta sen väitetään myös toteuttavan MVC hieman erilailla kuin MVC on alunperin tarkoitettu (**django_mvc**).

Tutkimuksen tarkoituksena on selvittää ohjelmistokehityksen ja ohjelmistoarkkitehtuurejen taustoja. Näiden pohjalta siirrytään tutkimaan millä tavalla MVC-arkkitehtuuri on toteutettu web-sovelluskehyksissä ja miten se eroaa alkuperäisestä MVC:n toteutuksesta. Lisäksi pyritään toteuttamaan MVC alkuperäisen toteutuksen mukaan käyttäen Flask-sovelluskehystä(**krasner**). Havaintojen pohjalta selvitetään voidaanko MVC tuoda sellaisenaan sovelluskehykseen 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 kommunikaation 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 sovelluskehyksien toteutuksessa ja selvitetään löytyykö sovelluskehyksien 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 rajaus

Aluksi tarkastellaan ohjelmikstokehitystä ja ohjelmistoarkkitehtuureja yleisesti pohjustamaan MVC-arkkitehtuuria. MVC:n toteutus käydään hyvin yksityiskohtaisesti läpi ja sen vertaaminen rajataan Python-websovelluskehyksiin. Tarkasteltavat web-sovelluskehykset rajataan Pyramid-, Django-, Plone- sekä Flask-sovelluskehyksiin. Pyramid, Django ja Plone 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 sovelluskehykseen. 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**), jotka pohjautuvat Trygve Reenskaugin esittelemään MVC:n määritelmään (**xerox-original**).

Tarkasteltavat sovelluskehykset käydään ensiksi yleisellä tasolla läpi, jonka jälkeen niitä tarkastellaan MVC:n näkökulmasta. Yleisellä tasolla tarkoitetaan sovelluskehyksen historian ja käyttötarkoituksen esittelemistä. Sovelluskehyksien muihin teknisiin ominaisuuksiin ei oteta kantaa. Flask-osiossa MVC-arkkitehtuuri toteutaan niinkuin se on Krasnerin julkaisussa määritelty. 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 sovelluskehyksien dokumentaatioiden etsintään.

Aluksi muodostetaan kokonaiskuva tuloksista, jolloin silmäillään 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 viittauksien 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 sovelluskehyksien 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**). Kirja on jaettu neljään pääosioon: *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. *Int-*

roduction 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**).

Toiseksi tärkeäksi lähteeksi valitaan— 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**). 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 (**burbeck**). 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**) (**.**) 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 (**xerox**). Erityisesti Reenskaugin ja Adele Goldbergin julkaisu, jossa kerrotaan jokaisen MVC komponentin tehtävä....

4.3 Web-sovelluskehykset

Web-sovelluskehyksistä löydetty kirjallisuus on hyvin suppea, eikä tämän varaan voida rakentaa kovinkaan perusteellista tieteellistä pohjaa. Tämän vuoksi tutkimuksessa joudutaan osaksi turvautumaan sovelluskehyksien omaan dokumentatioon täydentämään lähdemateriaalia. IEEE Xploren ja ACM Digital Libraryn avulla löytyy kolme julkaisua, joita käytetään tutkimuksen pohjana sovelluskehyksiä tarkastellessa. Ensimmäinen artikkeli on Okanovicin ja Mateljan kirjoittama artikkeli, jossa esitellään web-sovelluskehyksien suunnittelua (**ockanovic**). Se myös sivuuttaa lyhyesti MVC:tä. Toisena artikkelina käytetään ACM:stä tuloksena saatua Iwan Vosloon julkaisua, jossa käydään läpi yleisesti web-sovelluskehyksien rakennetta (**vosloo**). Kolmanneksi hyödynnetään Ignacion artikkelia, jossa esitellään ketteriä web-sovelluskehyksiä sekä millä tavalla näitä tulisi vertailla (**ignacio**). Lisäksi käytetään IEEE:stä Ahamedin julkaisua, joka esittelee yleisesti asioita joita tulisi ottaa huomioon sovelluskehystä valittaessa (**towards_framework**).

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 sovelluskehyksien lähdemateriaaliksi (**liza**).

5 Ohjelmistokehitys

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 ohjelmistoja. Monet tuotantolinjastot sekä rahoitusjärjestelmät teollisuudessa ovat täysin automatisoituja. Viihdeteollisuus kuten musiikki, pelit, elokuvat ja televisio ovat kaikki monimutkaisten ohjelmistojen 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 ohjelmia nykypäivänä. Liike-elämässä ihmiset kirjoittavat taulukkolaskentaohjelmia helpottaakseen työtään. Tutkijat ja insinöörit kirjoittavat ohjelmia prosessoimaan kerättyä dataa ja harrastelijat kirjoittavat ohjelmia viihdyttääkseen itseään. Suurin osa ohjelmistokehityksestä on kuitenkin toteutettu ammattilaisten toimesta tukemaan liiketoimintamalleja, kehittämään laitteita sekä rakentamaan erilaisia palveluita. Ammattilaisten totetuttamat ohjelmistot toteutetaan usein ryhmissä ja ne ovat suunnattu käytettäväksi muille kuin ohjelmoijille itselleen. Ohjelmistokehityksellä tarkoitetaan koko ohjelmiston elinkaaren mittaista tuotantoprosessia alkumäärittelyistä ylläpitoon asti.

5.1 Ohjelmisto

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 (**Sommenville**). Ohjelmistot eivät ole pelkästään tietokoneohjelmia vaan ne koostuvat ohjel-

man lisäksi dokumentaatioista sekä erilaisista konfiguraatioista, joilla määritellään ohjelma toimimaan tietyllä tavalla. Dokumentaatio sisältää usein tiedon ohjelmiston rakanteesta sekä käyttäjille suunnatun ohjeen miten sovellusta hallitaan ja käytetään (**Sommenville**) section 1.1. Testaus ja sen automaatio on myös suuren osana ohjelmistoja nykypäivänä <lähde tähän pohjaamaan väitettä>.

Sommerville jakaa ammatilaisten kehittämät ohjlemistot kahteen tuotekategoriaan: geneerise-set tuotteet ja räätälöidyt tuotteet. Geneerisillä tuotteilla

5.2 Laatu

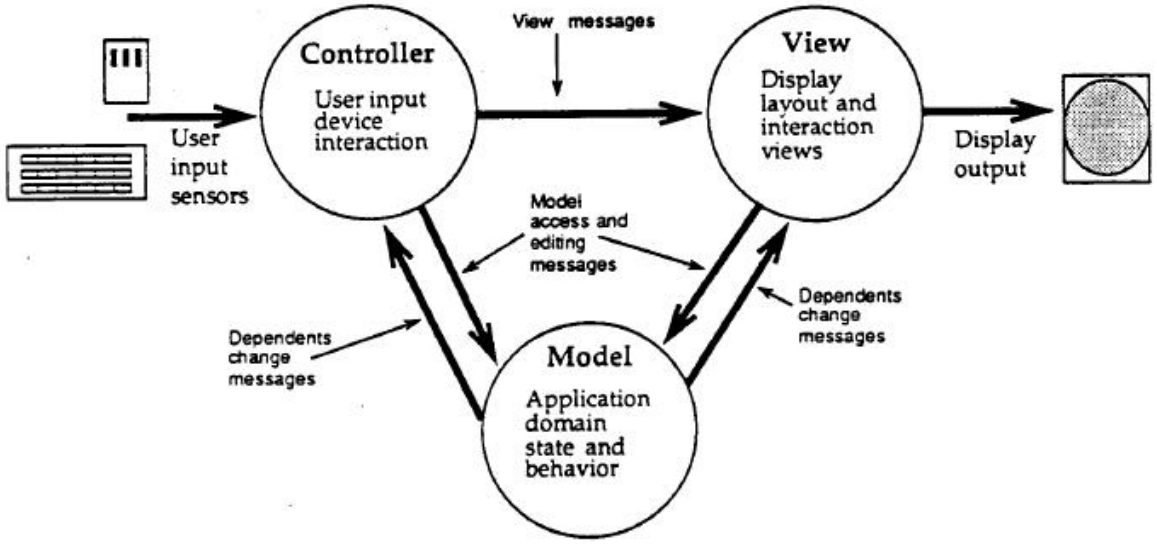
5.3 Ohjelmistojen tyypit

- lista tyypeist 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. "

6 Sovellusarkkitehtuurit

7 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 1. Model-View-Controller State and Message Sending **krasner_desc**

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

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**).

7.1 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 (**xerox**). Alkuperäinen Xerox PARC:n tuottama raportti MVC:stä oli Reenskaugin vuonna 1979 kirjoittama THING-MODEL-VIEW-EDITOR (**xerox-thing**). 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 esimerkisi valikon, joka vaihtuu sisällön muuttuessa (**xerox-thing**). 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) (**xerox-original**). Koska MVC:n historia ja suurin osa MVC:n alkuperäisistä julkaisuista pohjautuvat Smalltalk-ohjelmointikieleen, esitellään myös tässä tutkielmassa MVC:n totetusta Smalltalkilla. Tämä ei kuitenkaan rajoita tarkastelua, koska arkkitehtuurin idea pysyy täysin samana riippumatta ohjelmointikielestä.

7.2 Malli (Model)

Malli pitää yllä sovelluksen tilaa sekä vastaa sovelluksen tallentamasta datasta. Se voi olla esimerkiksi kokonaislukumuuttuja laskuri-sovelluksessa, merkkijono-olio tekstinkäsittely-ohjelmassa 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 tapauksesssa mallin ei tarvitse tietää ohjaimen ja näkymän olemassaolosta (**burbeck**).

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 (**burbeck**). 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**).

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 (**burbeck**). Näkymät käyttävät riippuuvuusmekanismia päivittääkseen itsensä mallin muutoksien perusteella. Esimerkiksi mallin muuttuessa lähetätetään *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 (**burbeck**).

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 valittin toimimaan kommunikaatiokanavana mallien ja näkymien välille, koska se aiheuttaa vähiten rajoituksia ja esteitä (**burbeck**).

7.2.1 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 riippu-

vuuksiin. Kummatkin sisältävät siis myös viitteen siihen malliin, johon ne on rekisteröity (**burbeck**). 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 (**burbeck**).

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 sovittamisen 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 esitetyssä 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 (**burbeck**). Monet näkymät käyttävät myös erilaisia transformaatioinstansseja, joilla hallitaan esimerkiksi näkymän skaalausta ruudulla. Tähän ei kuitenkaan perehdytä sen enempää, koska ne menevät tutkimuksen rajojen ulkopuolelle.

7.2.2 Ohjain (Controller)

Ohjaimen tehtävänä on ottaa vastaan syötteitä sekä koordinoida 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 syötelaitteiden sekä ohjaimen välillä. Sensori mallintaa syötelaitteiden 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 liitetyltä 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 hallinan. 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* -viesteja 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 *controlActivity* määrää ohjaimen toiminnan hallinnan aikana **krasner_desc**

7.2.3 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 yliluokasta.

```
1   Mouse MenuController subclass: #CounterControIler
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  isControlActlve
23        "Ota hallinta kun sinistä nappia ei paineta"
24        ↑super isControlActive & sensor blueButtonPressed not
```

Määrätään näkymä (*CounterView*), joka peritään *View* -yliluokasta. Määrätään 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
```

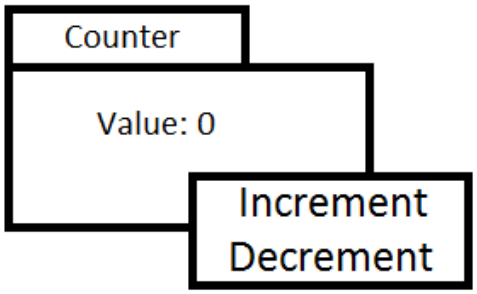
Luodaan myös metodi, joka palauttaa näkymään liitetyn ohjaimen.

```
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 2. Kuva CounterView -näkymästä **krasner_desc**

8 Sovelluskehykset

Soveluskehykset ovat suosittuja, koska ne tarjoavat uudelleenkäytettäviä ratkaisuja erilaisiin ongelmiin sovelluskehityksessä. Toimialueesta riippumatta sovelluskehyksiä tulisi käyttää hyväksi kirjottaessa 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(**towards_framework**). Tärkein ero sovelluskehyksen ja kirjaston välillä on se, että kirjaston ohjelmakoodi kutsustaan aina kehittäjän toimestaa. Sovelluskehyksessä taas kehittäjän ohjelmakoodia kutsutaan aina sovelluskehyksen toimesta (**pyramid_intr**).

Sovelluskehys ei myöskään generoi koodia. Se käyttää erilaisia komponentteja ja kirjastoja luodakseen infrastruktuurin, jonka päälle voidaan rakentaa sovelluksia sovelluskehyksen ehdoilla. Sovelluskehyksen 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 (**towards_framework**).

Web-sovelluskehykset ovat sovelluskehyksiä, jotka tarjoavat ratkaisuja helpottamaan websovellusten toteuttamista. Web-sovelluskehyksissä 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 (**vosloo**).

Yleisimmät teknologiat mitä web-sovelluskehykset tarjoavat ovat rajapinta tietokannalle, template-moottori sekä mahdollisuus käsitellä http-pyyntöjä ohjelmakoodissa. Tietokantarajapinnalla 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 (**Ghandeharizadeh**).

Seuraavassa esimerkissä esitellään miten Djangon 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 Djangon 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 (**jinja**).

Web-sovelluskehykset voidaan jakaa serveri-pohjaisiin ja selain-pohjaisiin sovelluskehyksiin. Serveri-pohjaisissa web-sovelluskehyksissä sovelluksen tilaa hallitaan serverin puolella. Tuolloin käyttöliittymää perustuu tilaan, mikä on sillä hetkellä serverin puolella. Selain-pohjaisissa sovelluskehyksissä sisältö muuttuu selaimen sisällä käyttäjän päässä (**vosloo**). Tässä tutkimuksessa käsiteltävät sovelluskehykset ovat Pythonilla toteutettuja serveri-pohjaisia sovelluskehyksiä. 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 voidaa laajentaa erilaisilla laajennoksilla.

8.1 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-sovelluskehykset. 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-sovelluskehyksen. 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-sovelluskehyksen kehittämiseen vaan tarjota kokoelma erilaisia teknologioita (**pylons_about**). Vuonna 2008 Pyramid tunnettiin nimellä repoze.bfg. Joulukuun alussa tapahtui ohjelmakoodin uudelleen nimeäminen ja ominaisuuksien lisääminen sekä poistaminen (**pyramid_about**).

Koska Pyramid pyrkii tarjoamaan vain välttämättömimmät työkalut web-sovelluksien kehitykseen, 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 määrätään 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 sovelluskehyksessä. Tämä ei kuitenkaan tarkoita sitä, että kehittäjä voisi täysin unohtaa tietoturvan merkityksen.

Static Assets - Staattisien resurssejen jakaminen niille tarkoitettuihin paikkoihin tiedostorakenteessa.

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.

8.2 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 muutamaamasta 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-sovelluskehyksen, jonka avulla he pystyivät vastaamaan journalistisen ympäristön haasteisiin. Kesällä 2005 he saivat kehitettyä sovelluskehyksen siten, että se oli käytössä suurimmassa osassa World Onlinen sivustoja. Tuolloin mukaan kehitykseen tuli Jacob Kaplan-Moss. Kehittäjät päättivät julkaista heinäkuussa 2005 sovelluskehyksen nimellä Django jazz-kitaristi Django Reinhardtin mukaan (**django_history**).

Django tarjoaa samat välttämättömät työkalut kuin Pyramidissa. Se tarjoaa myös ylläpitäjille suunnatun työkalun, josta voidaan hallita sovellusta käyttöliittymätasolta. Lisäksi se tarjoaa lomake-työkalut, käyttäjätasoisen autentikaation sekä tietokanta-abstrahoinnin. Tietokanta-abstraktiolla tarkoitetaan sisäänrakennettua virtuaalista ympäristöä tuomaan yhteys tietokantaan olio-ohjelmoinnin kautta (Object-relational mapping)(**djangobook**). 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.

8.3 Flask

Flask on mikro-sovelluskehys (micro-framework), jossa sovelluskehyksen lähdekoodi on pidetty mahdollisimman minimalistisena. Sen tarkoituksena ei ole tehdä valintoja kehittäjän puolesta millaisia teknologioita tämän tulisi käyttää. Ne teknologiat mitä Flask tuo muka-

naan, 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 sovelluskehyksistä minimaalisin eikä se toteuta minkään laista MVC:hen liittyvää pohjaa. Tämän vuoksi sitä käytetään hyväksi tutkiessa miten MVC tulisi toteuttaa web-sovelluskehykseen, jotta sen alkuperäiset määritelmät täyttyisivät (**flask**).

8.4 Plone

9 MVC:n toteutus web-sovelluskehyksissä

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 osioinnista. Osioinnin ei tule määrittää sovelluksen arkkitehtuuria.

(**ibm_watson**)

10 Yhteenveto

Liitteet