Toni Juhani Haka-Risku

MVC-arkkitehtuurin toteutus Pyramid web-sovelluskehyksessä

Tietotekniikan
kandidaatintutkielma
7. maaliskuuta 2012

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Toni Juhani Haka-Risku
Yhteystiedot: tojuhaka@gmail.com
Työn nimi: MVC-arkkitehtuurin toteutus Pyramid web-sovelluskehyksessä
Title in English: Pyramid-framework and MVC-architecture
Työ: Tietotekniikan kandidaatintutkielma
Sivumäärä: 28
Tiivistelmä: Millä tavalla MVC-arkkitehtuuri toteutuu Pyramid web-sovelluskehyksessä?
English abstract: How does the Pyramid-framework use the MVC-architecture?
Avainsanat: Pyramid, MVC, Framework, Python
Keywords: Pyramid, MVC, Sovelluskehys, Python

Esipuhe

Tutkimuksen aiheen valintaan vaikutti työelämässä saatu kokemus web-sovellusten kehittämisestä Python-ohjelmointikielellä. Erityisesti kokemusta on kertynyt Plone-sisällönhallintajärjestelmästä, josta myös Zope- sekä Grok-sovelluskehys on tullut tutuksi. Kokemuksen kautta on myös herännyt mielenkiinto muita tarjolla olevia vaihtoehtoja kohtaan, joista tällä hetkellä uusin on Pyramid. Koska suurin osa Python-pohjaisista web-sovelluskehyksistä käyttää MVC-arkkitehtuuria, on mielenkiinto herännyt myös MVC-arkkitehtuuriin. Tämä johti kandidaatintutkielman aiheeseen, jossa tarkastellaan MVC-arkkitehtuurin suhdetta Pyramid web-sovelluskehykseen.

1 Sanasto

Sovelluskehys tarjoaa sovelluksen kehitykseen mukaan tason, joka tarjoaa erilaisia kirjastoja yleisimpien ohjelmointiongelmian ratkaisemiseksi. Sovelluskehys on eräänlainen työympäristö, joka helpottaa sovelluksen toteuttamista.

ZCA *Zope Component Architecture* on Python-sovelluskehys, joka tarjoaa välineitä komponentti-pohjaisen sovelluksen toteutukseen [16].

Grok on Pythonilla toteutettu web-sovelluskehys, joka on rakennettu käyttäen pohjana ZTK:ta (Zope Tool) [17].

Zope on Python-ohjelmointikielellä toteutettu avoimen lähdekoodin web-sovelluspalvelin [18].

URL *Uniform Resource Locator* on merkkijono, joka viittaa johonkin tiettyyn resurssiin Internetissä [19].

HTML *HyperText Markup Language* on merkkauskieli, jolla luodaan dokumentteja, jotka ovat helposti siirrettäviä erilaisten alustojen välillä [20].

Sisältö

2 Johdanto

Toteutuuko MVC-arkkitehtuuri Pyramid-sovelluskehyksessä?

2.1 Tausta

Rakentaessamme interaktiivisia sovelluksia, modulaarisilla komponenteilla on paljon etuja. Suunnittelemalla komponentit mahdollisimman erillään toisistaan helpotetaan sovelluksen ohjalmakoodin hallintaa. Samalla pystytään muuttamaan ohjelmakoodia ilman, että tarvitsee tuntea koko sovelluksen toteutusta [4, s.2]. MVC-arkkitehtuuri on ohjelmistokehityksessä käytetty rakenne, joka määrää ohjelmakoodin jaottelun kolmeen osaan: Malli (model), Näkymä (view) ja Ohjain (controller). MVC on käytössä monissa sovelluksissa ja erityisesti se on saanut huomiota web-sovelluskehyksien toteutuksissa. Sovelluskehyksen tehtävänä on tuoda sovelluksen kehitykseen mukaan taso, joka tarjoaa erilaisia kirjastoja ratkaisemaan yleisimpiä ongelmia. Monet Python-pohjaiset sovelluskehykset kuitenkin luokitellaan MVC-sovelluskehyksiksi [13], mutta ne eivät todellisuudessa toteuta täysin MVC-arkkitehtuuria sellaisena kuin se alunperin olisi tarkoitettu. Näistä yleisimpiä sovelluskehyksiä ovat Python-pohjaiset ja erityisesti ZCA:n (Zope Component Architecture) omaavat kehykset, kuten esimerkiksi Pyramid ja Grok. Syy sille, miksi kyseiset sovelluskehykset eivät toteuta MVC:tä täysin, on konkreettisen ohjaimen puute. Herää kysymys siitä ovatko kyseiset sovelluskehykset MVC-kehyksiä vai eivät? Tämän tutkielman tarkoituksena on vastata kysymykseen Pyramidin näkökulmasta.

2.2 Aiheen rajaus

Tutkielmassa käsiteltävä aihe on rajattu tarkastelemaan Pyramid web-sovelluskehystä sekä MVC-arkkitehtuuria. Pyramidia tarkastellaan MVC:n näkökulmasta, joten muihin Pyramidissa käytetyihin tekniikoihin ei oteta kantaa. Pyramidiin on saatavana myös suuret määrät erilaisia lisäosia, joilla saadaan sovelluskehys muokattua tiettyyn tarkoitukseen. Lisäksi lisäosat voivat tuoda pyramidiin uusia tekniikoita. Tästä syystä on lähes mahdotonta tutkia MVC:n toteutusta kaikkien tekniikoiden näkökulmasta, joten tutkielmassa käsitellään ainoastaan Pyramidin alkuperäistä toteutusta ilman lisäosia.

Tutkielmassa selvitetään MVC:n sekä Pyramidin taustat ennen varsinaista tutkimuskysymyksen käsittelyä. MVC-arkkitehtuurin esimerkkitoteutukset ohjelmoin-

titasolla sidotaan Smalltalk -ohjelmointikieleen. Syy kielen valintaan löytyy MVC-arkkitehtuurin historiasta sekä lähekirjallisuuden tarjoamasta materiaalista. Esitellyt ratkaisut pätevät kuitenkin mihin tahansa muuhun ohjelmointikieleen, joten kielen valinnalla ei tässä tilanteessa ole merkitystä. MVC:n tarkastelu Pyramidissa toteutetaan Krasnerin julkaiseman artikkelin pohjalta, jossa käydään kattavasti läpi MVC:n käyttöä ohjelmakooditasolla [4].

Tutkimuksessa esitetyissä esimerkeissä oletetaan lukijalla olevan perus ohjelmointitaidot hallussa. Joitakin tutkielmassa esiintyviä termejä ei selitetä erikseen, vaan ne löytyvät tutkielman sanasto-osiosta.

3 Kirjallisuuskatsaus

Kirjallisuuskatsauksessa käydään läpi vaihe vaiheelta, miten lähdemateriaalia kerätään tutkielmaa 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 perinteinen hakukone.

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 tutkielman aihepiiriin.

Seuraavaksi artikkeleista valitaan tutkielmalle pohjakirjallisuus. Tässä vaiheessa artikkelit luetaan huolellisesti läpi ja varmistutaan siitä, että ne ovat tieteellisesti päteviä tutkielmaa varten. Tutkielmassa esiintyy myös satunnaisia viittauksia, joita ei ole kirjallisuuskatsauksessa mainittu. Tutkielman pääkirjallisuus kuitenkin käydään läpi kirjallisuuskatsauksessa.

Haussa käytetään seuraavia hakutermejä: "MVC", "MVC Architecture"ja "MVC-Architecture". Erityisesti artikkeleita löytyy MVC-arkkitehtuurin soveltamisesta erilaisissa tekniikoissa. Tarkasteltavat artikkelit rajataan kuitenkin niihin, jotka esittelevät suoraan MVC:tä itseään.

Google Scholarin tuloksista löytyy kaksi artikkelia, jotka sopivat lähdemateriaaliksi tutkielmaan. Ensimmäinen artikkeleista on John Deaconin kirjoittama artikkeli, joka tarkastelee lyhyesti MVC:tä [1]. 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 [2]. Burbeckin artikkeliin viitataan monissa MVC:tä käsittelevissä julkaisuissa, joten sen arvo tämän tutkielman pohjakirjallisuudessa on vahva.

Seuraavaksi kartoitetaan pohjakirjallisuutta käyttäen ACM Digital Library sekä IEEE XPlore -hakukoneita. Tuloksista löytyy Glenn E. Krasnerin kirjoittama julkaisu, jossa esitellään MVC:n toteutusta erilaisissa Smalltalk-sovelluksissa. Julkaisusta löytyy useita versioita, joista tässä tutkielmassa käytetään kumpaakin [3] [4]. Suurimmaksi osaksi viitataan kuitenkin uudempaan julkaisuun. Tuloksien joukosta löytyy myös paljon MVC:tä soveltavia tutkimuksia, jotka eivät suoraan tarkastele MVC:tä sellaisenaan. 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 kehittäjä Trygve Reenskaug, jonka omia julkaisuja sekä kotisivujen MVC-osiota käytetään myös lähteenä tutkielmassa [5].

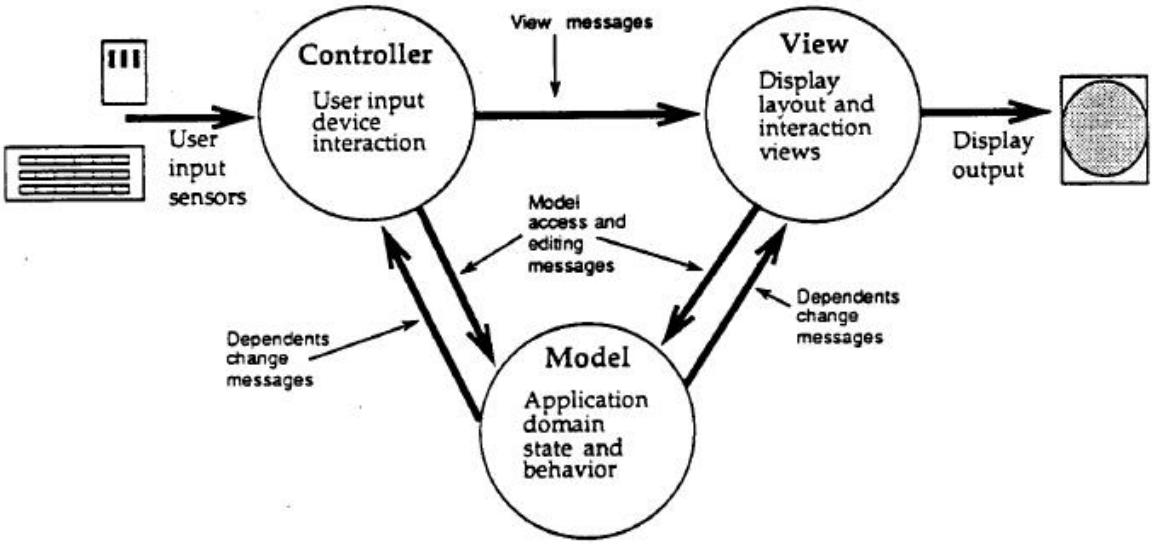
4 Tutkimuksen rakenne

Tutkielma aloitetaan käymällä kirjallisuuskatsauksena läpi MVC-arkkitehtuurin käsite sekä sen toteutus ohjelmointitasolla. Kirjallisuutena käytetään pääasiassa Krasnerin ja Burbeckin julkaisua [4, 2]. Tämän jälkeen käydään läpi Pyramid sovelluskehyksen peruselementit sekä asetetaan kriteerit, joiden pohjalta MVC:tä tutkitaan Pyramidissa. Kriiteereillä tarkoitetaan ennalta määrättyjä ominaisuuksia, joita Pyramidin tulee toteuttaa MVC:n näkökulmasta. tutkielmassa toteutetaan yksinkertainen laskurisovellus käyttäen pohjana Krasnerin julkaisussa toteutettua vastaavaa sovellusta [4]. Lopuksi tehdään johtopäätökset tutkielman tuloksista.

5 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 sovel-

luksen tilaa ja vastata sen käsittelemästä datasta ohjaimelle ja näkymälle. Näkymän tehtävänä on taas toteuttaa sovellukselle käyttöliittymä. Ohjaimen tarkoitus on ottaa vastaan syötteitä käyttäjältä käskien mallia ja näkymää muuttumaan tarvittaessa.



Kuva 1: Model-View-Controller State and Message Sending [4, s. 5]

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 [2].

Luodessamme MVC-arkkitehtuurin toteuttavia komponentteja, tulee ne periä jostakin abstraktista pohjaluokasta (Model, View tai Controller), joka määrittelee kyseisen komponentin käyttäytymisen MVC:ssä [4, s. 5]. Tässä kappaleessa käydään jokaisen komponentin toteutus erikseen läpi käyttäen ohjelmointikielenä Smalltalkia. Lähteenä käytetään Krasnerin julkaisua [4].

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 kiinnitettyille 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 [4, s. 4].

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 [9, s. 11-12].

5.1 Historia

MVC:n esitteli 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 [5]. Alkuperäinen Xerox PARC:n tuottama raportti MVC:stä oli Reenskaugin vuonna 1979 kirjoittama *THING-MODEL-VIEW-EDITOR* [6]. Raportti esitteli MVC:n komponentteja käyttäen hyväksi esimerkkejä Reenskaugin omasta suunnittelutyöstä. Reenskaug päätyi kuitenkin termeihin *Model-View-Controller* ja julkaisi saman vuoden lopulla raportin, jossa määritellään lyhyesti jokaisen komponentin tehtävä (*MODELS-VIEWS-CONTROLLERS*) [7].

Koska MVC:n historia ja suurin osa MVC:n alkuperäisistä julkaisuista pohjautuvat Smalltalk -ohjelmointikieleen, joudutaan väistämättä sitomaan MVC:n tarkastelu Smalltalk:iin. Tämä ei kuitenkaan rajoita tarkastelua, koska arkkitehtuurin idea pysyy täysin samana riippumatta ohjelmointikielestä.

5.2 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 [4, s. 3]. 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 [2].

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 [2]. 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 [4, s. 4].

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 [3, 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 [2].

5.3 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 [2]. Jokaisella näkymällä on tasan

yksi malli ja yksi ohjain [4, s. 7].

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 [2].

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 yleisesti määrätty ominaisuus, jolla hoidetaan alanäkymien poistaminen sekä lisääminen. Samalla tulee määritellä ominaisuus, jolla sisäiset transformaatiot tuodaan transformaatioluokalle. Tämä helpottaa näkymän ja sen alanäkymien yhdistämistä [4, s. 8].

5.4 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. Teknisesti ohjaimen käyttäytymisen määrittelee seuraavat metodit, joiden avulla ohjaimet viestivät [4, s. 9]:

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 [4, s. 9].

5.5 Esimerkkiohjelma

Seuraavaksi esitellään Dortmundin yliopistossa kirjoitettu yksinkertainen esimerkkiohjelma Smalltalkilla siitä miten MVC:n toteutus tuodaan sovellukseen käytännössä. Ohjelmakoodi löytyy myös Krasnerin artikkelista [4, s. 20]. 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 numero-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. Vastaava laskurisovellus toteutetaan Pyramidilla tässä tutkielmassa.

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

```
Model subclass: #Counter
   instanceVariableNames: 'value'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Demo-Counter'
```

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

```
Counter methods For: 'Initialize-release'
Initialize
    "Aseta alkuarvoksi 0"
    self value: 0
Counter methodsFor: 'accessing'
value
    "Palauta mallin arvo"
    ↑value
value: aNumber
    "Aseta mallin arvo"
    value <- aNumber.
    self changed "to update displayed value"
Counter methodsFor: 'operations'
decrement
    "Vähennä mallin arvoa yhdellä."
    self value: value -1
Increment
    "Lisää mallin arvoa yhdellä."
    self value: value + 1
```

Lisätään luokkaan metodi, jolla itse luokasta saadaan muodostettua instanssi.

```
Counter class methodsFor: 'instance␣creation'
new
    "Palauta uusi instanssi luokasta"
    ↑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.

```
Mouse MenuController subclass: #CounterControIler
    instanceVariableNames: '␣'
    classVariableNames: '␣'
```

```
    poolDictionaries: '␣'
    category: 'Demo-Counter'
CounterController methodsFor: 'initialize-release'
initialize
    "Alusta valikko, jossa on mahdollisuus vähentää tai
       lisätä mallin arvoa"
    super initialize.
    Self yellowButtonMenu: (PopUpMenu labels:
                            'Increment\Decrement' withCRs)
    yellowButtonMessages: #(increment decrement)
CounterController methodsFor: 'menu␣messages'
decrement
    "Vähennä mallin arvoa yhdellä."
    self model decrement
increment
    "Lisää mallin arvoa yhdellä"
    self model increment
CounterController methodsFor: 'control␣defaults'
isControlActlve
    "Ota hallinta kun sinistä nappia ei paineta"
    ↑super isControlActive & sensor blueButtonPressed not
```

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

```
View subclass: #Counterview
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Demo-Counter'

CounterView methodsFor: 'displaying'
displayView
    "Näytä mallin arvo näkymässä"
    | box pos displayText |
    box ← self insetDisplayBox.
    "Asettele teksti näkymään. Asettelu ei
```

```
    ole tutkielman kannalta oleellista."
    pos ← box origin + (4 @ (box extent y / 3)).
    displayText ← ('value:', self model value printString)
                asDisplayText.
    displayText displayAt: pos
```

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

```
CounterView methodsFor: 'updating'
update: aParameter
  "Yksinkertaisesti päivitä näyttö uudestaan"
  self display
```

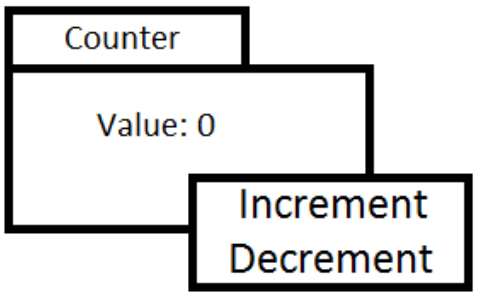
Luodaan myös metodi, joka palauttaa näkymään liitetyn ohjaimen.

```
CounterView methodsFor: 'controller␣access'
defaultControllerClass
    "Palauta näkymään rekisteröity ohjain"
    ↑CounterController
```

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

```
CounterView class methodsFor: 'instance␣creation'
open
    "Avaa näkymän uudelle laskurisovellukselle. Tässä
    metodissa nähdään kuinka näkymä huolehtii mallin
    rekisteröinnistä sekä nähdään kuinka näkymiä voi
    olla useita sisäkkäin."
    | aCounterView topView |
    "Luo laskurinäkymälle uusi näkymä, joka näyttää
    laskurin arvon"
    aCounterView ← CounterView new
    "Asetetaan malliksi Counter -luokan instanssi"
    model: Counter new.
    aCounterView borderWidth: 2.
    aCounterView insideColor: Form white.
    "Asetetaan ylimmäksi näkymäksi StandardSystemView
    -luokan instanssi, joka vastaa perinteistä
```

```
ikkunointimallia"
    topView ← StandardSystemView new
        label: 'Counter'.
    topView minimumSize: 80@40.
    "Lisätään edellä luotu laskurinäkymä ylinäkymän
    alanäkymäksi"
    topView addSubView: aCounterView.
    "Käynnistetään ohjain"
    topView controller open
```



Kuva 2: Kuva CounterView -näkymästä [4]

6 Pyramid

Pyramid on Python-pohjainen web-sovelluskehys, jonka tehtävänä on helpottaa web-kehitystä tarjoamalla kehittäjälle valmiita työkaluja avuksi kehitykseen.

6.1 Tausta

Sovelluskehyksen tehtävänä on tuoda sovelluksen kehitykseen mukaan taso, joka tarjoaa erilaisia kirjastoja ratkaisemaan yleisimpiä ongelmia, joita tulee vastaan sovelluksen kehityksen aikana. Näin vältetään jo ratkaistujen perusoperaatioiden toistoa ja pystytään keskittymään suoraan sovelluksen toteuttamiseen. Web-sovelluskehykset ovat erityisesti suunnattuja web-sovellusten ja -palvelujen toteuttamiseen. Tärkein ero sovelluskehyksen ja kirjaston välillä on se, että kirjaston ohjelmakoodi

kutsustaan aina kehittäjän toimestaa. Sovelluskehyksessä taas kehittäjän ohjelmakoodia kutsutaan aina sovelluskehyksen toimesta [10].

6.2 Sisältö

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 [10].

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 HTML-kuvauskieltä, jolla määrätään näkymän rakenne. Tämän avulla pystytään erottamaan käyttöliittymä sovelluslogiikasta tehokkaasti.

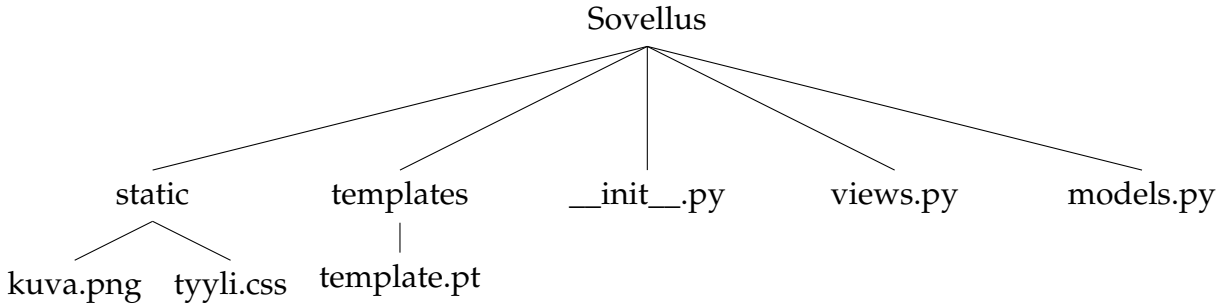
Security - Perinteiset tietoturvaongelmat tulee olla ratkaistuna valmiiksi jo sovelluskehyksessä. Tämä ei kuitenkaan tarkoita sitä, että kehittäjä voisi täysin unohtaa tietoturvan merkityksen.

Static Assets - Staattisien resurssejen jakaminen niille tarkoitettuihin paikkoihin tiedostorakenteessa.

Yllä määriteltyjen neljän pääongelman lisäksi Pyramid tarjoaa lisäosien kautta monia erilaisia työkaluja, joiden avulla pystytään laajentamaan sen ominaisuuksia [10]. Tutkielman aiheen rajauksen vuoksi ei kuitenkaan käydä läpi yksityiskohtaisemmin Pyramidin toteutusta ja siihen liitettävissä olevia lisäosia, vaan keskitytään tarkastelemaan MVC:n toteutusta Pyramidissa.

6.3 Tiedostorakenne

Kuten monissa web-sovelluskehyksissä, jaetaan sovelluslogiikka erillisiin tiedostoihin. Näin jokaisella ohjelmakoodilla on oma paikkansa ja sovelluksen hallinta kehityksen aikana helpottuu. Samalla sovelluskehys löytää tarvittavat asiat oikeista paikoista. Pyramidissa ei kuitenkaan pakoteta ohjelmakoodia mihinkään tietynnimiseen tiedostoon. Esimerkiksi mallin ohjelmakoodin ei tarvitse olla *models.py* -tiedostossa, vaan sen voi määritellä myös muualla. Tässä tutkimuksessa käytämme kuitenki tiedostorakennetta, joka generoidaan *Scaffold* -työkalulla [14]. Scaffoldilla voidaan luoda valmiita tiedostorakenteita, jotka sisältävät kaikki tarvittavat osat sovelluksen rakentamiseen. Seuraavassa kuvassa esitellään yksinkertaisen web-sovelluksen rakenne Pyramidissa:



Ilman tiedostopäätettä olevat lehdet ovat kansioita, jotka sisältävät muita tiedostoja. Tiedostot *models.py*, *views.py* ja *__init__.py* ovat Python-moduuleita, jotka sisältävät sovelluksen ohjelmakoodin. Yksinkertaistettuna *__init__ .py*:ssä alustetaan sovellus. Pyramidissa näkymien ohjelmakoodi on yleensä määritelty *views.py* -tiedostossa ja mallit *models.py* -tiedostossa. Tämä on suositeltavaa, mutta ei välttämätöntä. Mallejen ja näkymien ohjelmakoodit voidaan määritellä myös muissa tiedostossa, jos näin halutaan. Kansion *static* sisällä ovat kaikki staattiset resurssit, joita käytetään sovelluksessa. Staattisia resursseja voivat olla esimerkiksi kuva-, ääni- tai tyylitiedostot. Kansion *templates* alta löytyvät kaikki *template*-tiedostot, jotka määräävät sovelluksen näkymän selaimessa. Templatet yhdistävät selaimille tarkoitetun HTML-kuvauskielen sekä ohjelmakoodin yhtenäiseksi dokumentiksi, joka tuodaan selaimelle näytettäväksi. Tässä tutkielmassa mielenkiinto on kohdistettu *views.py*, *models.py* ja *template.pt* -tiedostoihin, koska ne sisältävät tarvittavat toteutukset MVC:n tarkasteluun. Seuraavaksi esitellään lyhyt selvitys MVC:n kannalta oleellisimpien tiedostojen sisällöstä:

views.py Nimensä mukaisesti sisältää näkymien ohjelmakoodin, jotka liitetään *templa-*

te-tiedostoihin.

template.pt Tuodaan sovelluksen näkymä selaimelle käyttäen HTML-kuvauskieltä, jolla määrätään näkymän rakenne. Tämän avulla pystytään erottamaan käyttöliittymä sovelluslogiikasta tehokkaasti.

models.py Sisältää mallejen ohjelmakoodit.

7 MVC & Pyramid

Pyramidin kehittäjien dokumenteissa esitellään Pyramid MVC-kehyksenä, mutta samalla myös kyseenalaistetaan tämä väite. Erityisesti mallin ja ohjaimen määritelmä puuttuu [10]. Tässä osiossa käsitellään Pyramidia MVC:n näkökulmasta.

7.1 Näkökulman rajaaminen

Tutkielmassa keskitytään Pyramidin alkuperäiseen toteutukseen, joten lisäosien tarjoamiin erilaisiin tekniikoihin ja toteutuksiin oteta kantaa. Tämä on tutkielman kannalta tärkeää, koska kaikkien mahdollisten lisäosien tuomien tekniikoiden tutkiminen olisi lähes mahdotonta. Näin tutkielma pysyy myös kandidaatin tutkielman rajoissa.

Pyramidin sovellusarkkitehtuuri on toteutettu käyttäen pohjana MVC-arkkitehtuuria. Tämä ei kuitenkaan tarkoita sitä, että Pyramid toteuttaisi MVC:n teknisesti sellaisena kuin esimerkiksi Krasner määrittelee [4]. Teknisyydellä tarkoitetaan tässä kontekstissa tarkkoja ohjelmointiteknisiä ratkaisuja, jotka ovat sidottuja tiettyyn kieleen tai työkaluun. Web-ohjelmointi tuo itsessään jo rajoituksia MVC-komponenttien kommunikaatiolle Pyramidissa. Esimerkiksi pyyntö tulee suorittaa aina, jos palvelimelta halutaan tuoda dataa selaimelle [11]. Tämän perusteella malli ei voi viestittää muutoksestaan näkymälle ja ohjaimelle ilman niiden kautta tulevaa pyyntöä toisinkuin esimerkiksi Krasnerin määritelmässä [3]. Ohjelmointityökalujen määrän sekä vauhdikkaan kehityksen myötä tämä voidaan kuitenkin mahdollisesti kiertää tulevaisuudessa. Tästä syystä MVC-toteutusta ei ole tutkielman kannalta järkevää tutkia liian teknisestä näkökulmasta, vaan toteutusta tulee tarkastella korkeammalla tasolla.

Tässä osiossa oletetaan MVC:n olevan abstrakti malli ohjelmakoodin jakamiselle helposti hallittaviin osiin. Se kuinka tämä teknisesti toteutetaan ei ole tutkielman

kannalta oleellista. Koska MVC:n toteutusta Pyramidissa tarkastellaan abstraktilla tasolla, määritetään selkeät ominaisuudet, jotka MVC-komponenttien tulee toteuttaa. Jokaisen komponentin on toteutettava sille määrätyt ominaisuudet, jotta voidaan todeta sen kuuluvan osaksi MVC-arkkitehtuuria. Ominaisuudet määritellään käyttäen pohjana Burbeckin ja Krasnerin julkaisuja [2, 4].

Tärkeimpänä vaatimuksena on se, että näkymä ja ohjain luovat rajapinnan, jonka kautta käyttäjä keskustelee mallin kanssa. Malli ei saa olla suorassa yhteydessä käyttäjään [9, s. 10]. Mallin tulee olla riippumaton näkymästä ja ohjaimesta. Sen tulee myös hallita sovelluksen tilaa sekä pystyä antamaan informaatiota sovelluksen tilasta [2]. Näkymän on keskusteltava mallin kanssa sekä hoitaa mallilta saadun datan graafinen näyttäminen [8, s.1]. Ohjain puolestaan ottaa vastaan syötteitä ja lähettää viestejä tämän perusteella näkymälle ja mallille [2]. Näiden rajoituksien perusteella pystytään vastaamaan tutkielmassa siihen, toteuttaako Pyramid MVC-arkkitehtuurin vai ei.

Tarkastelua varten toteutetaan Pyramidilla vastaava laskuri-sovellus kuin Krasnerin julkaisussa käyttäen Pyramidin tarjoamia ominaisuuksia [4]. Sovellus rajataan käyttämään SQL-tietokantaa mallin datan tallennukseen sekä *URL dispatch* -tekniikkaa [12, 15]. Sovelluksen pohjalta tarkastellaan MVC:n kannalta kolmea oleellisinta tiedostoa: *views.py*, *models.py* ja *template.pt*. Näin tutkielman tarkastelu pystytään rajaamaan mahdollisimman pienelle alueelle, jolloin tutkielma on helpompi keskittää tarkastelemaan yksittäistä tekniikkaa. Samalla tutkielma pysyy myös kandidaatintutkielman rajoisssa. Kaikkea sovelluksen sisältämää ohjelmakoodia ei käydä läpi, vaan tiedostoista otetaan esiin vain MVC:n kannalta oleellisin ohjelmakoodi.

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 sovelluskehykseksi. Koska Pyramidissa MVC:n määrittely on hyvin epävakaalla 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.

7.2 Tiedostojen tarkastelu

```
1  # Tiedosto: models.py
2  class Counter(Base):
3      __tablename__ = 'counter'
4
5      # Asetetaan mallille atribuutit, 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 kom-

ponentti, 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 käyttäjästä 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 puhutaessa ohjaimesta Pyramidissa, tarkoitetaan sillä *views.py* -tiedoston sisältämää 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 -tiedostssa
24     return {'value': counter.value}
```

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

män jälkeen rekisteröidään malli mukaan funktioon (8). Koska tarkastelemme funktiota ohjaimena, voimme tulkita templaten 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 määrätyt ohjaimen ominaisuudet.

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

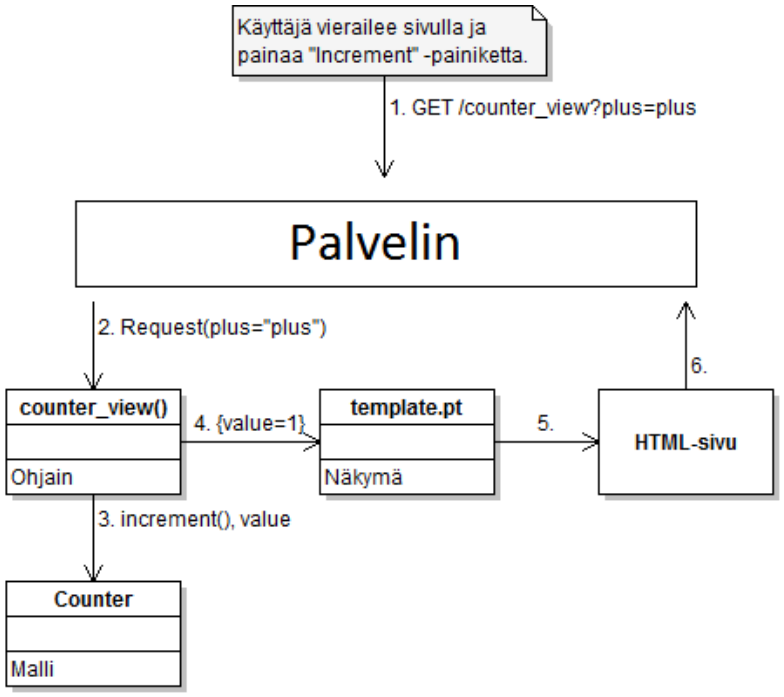
```
1  <body>
2    <h1>${value}</h1>
3    <form action="./" method="get">
4      <button type="submit" name="plus" value="plus">
5                Increment </button>
6      <button type="submit" name="minus" value="minus">
7                Decrement </button>
8    </form>
9  </body>
```

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 kutsuttukin. 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.

7.3 Sovelluksen toiminta

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



Kuva 3: Laskurisovelluksen toiminta

1 Palvelimelta pyydetään HTTP-protokollan mukaisesti sivua *plus* -parametrilla.
2 Palvelin pyytää sovellukselta sivua. Parametri tuodaan sovellukselle *request*-oliossa.
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.

7.4 Yhteenveto

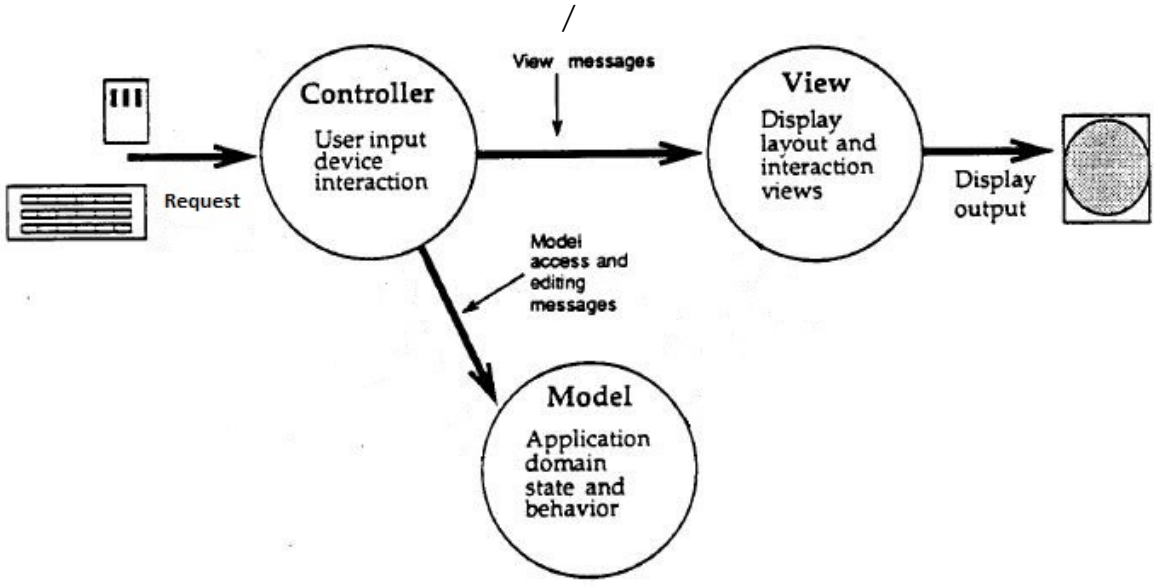
Malli sekä ohjain toteutuvat Pyramidissa MVC:n mukaisesti. Malli on itsenäinen komponentti, jolla ei ole riippuvuutta näkymään tai ohjaimeen. Se myös huolehtii sovelluksen käsittelemästä datasta ja vastaa tarvittaviin pyyntöihin. Ohjain taas

huolehtii request-oliossa tulevista syötteistä ja vaikuttaa malliin sekä näkymään. Kumpikin edellä mainittu komponentti toteuttaa MVC:ssä sille tarkoitetun roolin. 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.

8 Tulokset

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 toteutetaan eri tavalla. Erityisesti näkymäksi erotettu template tuo ongelmia, koska se ei suoraan yhteydessä mallin kanssa.

Pyramidin MVC-komponenttien kommunikointi voidaan esittää käyttäen pohjana muokaten Krasnerin kommunikaatiomallia [4]. Alla esitellyssä kuvassa havainnollisestetaan, kuinka mallin ja näkymän kommunikaatio puuttuu täysin ja kaikki data tuodaan ohjaimen kautta.



Kuva 4: Pyramidin kommunikointi MVC-komponenttien kesken. Kuva on muokattu Krasnerin esittelemästä kommunikointimallista [4]

Yllä esitetyssä kommunikaatiomallissa ohjain saa vastaan request-olion, jossa

tuodaan kaikki tarvittava tieto käyttäjästä. 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.

9 Johtopäätökset

Web-sovellusten tekninen toteutus tuo suoraan rajoitteita MVC-arkkitehtuurin toteutukselle. Erityisesti ongelmaksi muodostuu mallin viestiminen muutoksesta, koska kaikki data haetaan selaimen pyynnön kautta. Samalla MVC:n alkuperäisessä julkaisussa määrätään, että näkymä ei saa olla missään tekemisissä käyttäjän syötteiden kanssa [8, s. 1]. Esimerkiksi hiiren valinnat sekä näppäimistön syötteet tulisivat aina ohjaimen kautta. Tämä ei kuitenkaan ole Pyramidissa mahdollista, koska monissa tapauksissa syötteet tulevat selaimen kautta templaten generoiman HTML-sivun avulla. Syötteet tulee siis rajata request-olion sisältämään dataan. Tästä syystä MVC:tä ei ole järkevää tarkastella liian teknisestä näkökulmasta, vaan rajata tarkastelu korkeammalle tasolle.

Alunperin Pyramidin dokumentaatiossa kyseenalaistettiin mallin sekä ohjaimen toteutus [10]. 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 [8, 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.

Vaikka MVC-arkkitehtuurin toteutusta tutkitaan hyvin abstraktilla tasolla, ei Pyramid toteuta MVC:tä. Pyramidissa näkymä ja ohjain on yhdistetty yhteen komponenttiin, joka kommunikoi mallin kanssa. Näitä komponentteja ei pystytä kuitenkaan erottamaan toisistaan siten, että MVC-arkkitehtuuri toteutuisi. Tutkielman pohjalta voimme määritellä Pyramidin käyttävän MVC:n pohjalta toteutettua muunnosta, jossa kommunikaatiomallia on muutettu ja ohjain on sisällytetty näkymään.

Lähteet

[1] John Deacon Computer Systems Development, Consulting & Training *Model-View-Controller (MVC) Architecture*, August 1995, revised August 2000, April 2005 and May 2009

[2] Burbeck Steve, 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`

[3] Glenn E. Krasner & Stephen T. Pope, 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`

[4] 1988 Glenn E. Krasner & Stephen T. Pope, ParcPlace Systems Inc, *"A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System"* `http://www.create.ucsb.edu/~stp/PostScript/mvc.pdf`

[5] XEROX PARC 1978-79 *MVC XEROX PARC 1978-79* http://heim.ifi.uio.no/ trygver/themes/mvc/mvc-index.html

[6] XEROX PARC 1979 *THING-MODEL-VIEW-EDITOR* http://heim.ifi.uio.no/ trygver/1979/mvc-1/1979-05-MVC.pdf

[7] XEROX PARC 1979 *The Original MVC reports* `http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf`

[8] Trygve Reenskaug 1979 *Models-Views-Controllers* `http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf`

[9] *The Model-View-Controller (MVC) Its Past and Present* `http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf`

[10] *Pyramid Introduction* `http://www.kemeneur.com/clients/pylons/docs/pyramid/narr/introduction.html`

[11] *Research on Web Instant Messaging Using REST Web Service* http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=05607397

[12] *Pyramid Documentation, URL Dispatch* `http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/urldispatch.html`

[13] bestwebframeworks 2009 *List of Python MVC-frameworks by bestweb-frameworks.com* http://www.bestwebframeworks.com/compare-web-frameworks/python/

[14] Pyramid Documentation *Creating Pyramid Scaffolds* http://docs.pylonsproject.org/projects/pyramid/en/latest/narr/scaffolding.html

[15] W3Schools.com, 1999-2012 Refsnes Data *Introduction to SQL* `http://www.w3schools.com/sql/sql_intro.asp`

[16] 2007-2009 Baiju M *A Comprehensive Guide to Zope Component Architecture* `http://www.muthukadan.net/docs/zca.pdf`

[17] 2007-2008, The Grok Community *Grok: About* `http://grok.zope.org/about/`

[18] 2012 Zope Foundation *Zope Website* `http://www.zope.org`

[19] T. Berners-Lee CERN L. Masinter Xerox Corporation M. McCahill, University of Minnesota December 1994 *Uniform Resource Locators (URL)* `http://www.ietf.org/rfc/rfc1738.txt`

[20] Dave Raggett, W3C Recommendation 14-Jan-1997 *HTML 3.2 Reference Specification* `http://www.w3.org/TR/REC-html32`