TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JONI ERONEN

GRAAFISTEN KÄYTTÖLIITTYMIEN OHJELMISTOARKKITEHTUURIT

Kandidaatintyö

Tarkastaja: Tiina Schafeitel-Tähtinen
Jätetty tarkastettavaksi 09.12.2017

TIIVISTELMÄ

Joni Eronen: Graafisten käyttöliittymien ohjelmistoarkkitehtuurit
Tampereen teknillinen yliopisto
Kandidaatintyö, 25 sivua, 0 liitesivua
Joulukuu 2017
Tietotekniikan kandidaatin tutkinto-ohjelma
Pääaine: Ohjelmistotekniikka
Tarkastaja: Tiina Schafeitel-Tähtinen

Avainsanat: Graafinen käyttöliittymä, ohjelmistoarkkitehtuurit, kandidaatintyö, kirjallisuusselvitys

Suurien ohjelmistojen hallinta on vaikeaa ja ohjelmistoarkkitehtuureja on monia. Oikean ohjelmistoarkkitehtuurin valinta voi olla vaikeaa ja niiden tutkiminen ja etsiminen työlästä. Tässä työssä kootaan yhteen neljä graafisten käyttöliittymien ohjelmistoarkkitehtuuria ja selvitetään, millaisten ohjelmistojen toteutukseen ne soveltuvat.

Jokaisesta tässä työssä esiteltävästä ohjelmistoarkkitehtuurista on selvitetty niiden toiminta ja erityispiirteet vertailua ja soveltuvuutta varten. Ohjelmistoarkkitehtuurin toiminnasta riippuen työssä on avattu myös matalamman tason suunnittelumalleja. Esiteltäviksi ohjelmistoarkkitehtuureiksi on valittu Smart UI -, MVC-, MVP- ja MVVM-arkkitehtuurit.

Smart UI -arkkitehtuuri on esiteltävistä ohjelmistoarkkitehtuureista yksinkertaisin ja soveltuu parhaiten pienien ohjelmistojen tekemiseen.

MVC-arkkitehtuuri on tämän työn vanhin ja tunnetuin ohjelmistoarkkitehtuuri. MVC-arkkitehtuuri mahdollistaa paremman ohjelmiston osien vastuujaon kuin Smart UI -arkkitehtuuri ja soveltuu ohjelmistojen toteuttamiseen, joissa pitää tehdä itse käyttäjän syötteiden lukeminen.

MVP-arkkitehtuuri on kehitetty MVC-arkkitehtuurin pohjalta yhdistämällä näkymä ja ohjain ja luomalla ohjaimen tilalle esittelijä. MVP-arkkitehtuurista on monia eri variaatioita, jotka eroavat toisistaan esittelijän roolin mukaan. MVP-arkkitehtuurin vahvuus on valmiiden käyttöliittymäkomponenttien käyttäminen ja esittelijän ja näkymän suora yhteys, joka helpottaa tiettyjen toimintojen toteuttamista MVC-arkkitehtuuriin verrattuna.

MVVM-arkkitehtuuri mahdollistaa ohjelmistoprojektin jakamisen käyttöliittymään ja käyttöliittymän logiikkaan. MVVM-arkkitehtuurin ajatuksena on toteuttaa käyttöliittymä ja sen logiikka eri työkaluilla ja kielillä, jotta visuaalisesti lahjakkaat ihmiset voivat keskittyä logiikan tekemisen sijaan ohjelmiston ulkonäköön. MVVM-arkkitehtuuri soveltuukin parhaiten ohjelmistoihin, joissa käyttöliittymä ja sen logiikka on tehty eri kielillä tai työkaluilla.

SISÄLLYSLUETTELO

KUVALUETTELO

LYHENTEET JA MERKINNÄT

GUI	engl. Grafical User Interface, Graafinen käyttöliittymä
MV	engl. Model-view, Malli–näkymä
MVC	engl. Model-view-controller, Malli–näkymä–ohjain
MVP	engl. Model-view-presenter, Malli–näkymä–esittelijä
MVVM	engl. Model-view-modelview, Malli–näkymä–näkymämalli
PAC	engl. Presentation-abstarction-control, Näkymä–abstraktio–ohjain
PM	engl. Presentation model, Esittelijä malli

1. JOHDANTO

Suurin osa ohjelmistoprojekteista epäonnistuu. Niiden budjetti ylittyy, ne ovat myöhässä, ominaisuudet eivät vastaa määriteltyjä tai niitä ei saateta ollenkaan loppuun. Vuonna 2000 onnistuneita ohjelmistoprojekteja oli vain 28 %. Ohjelmistoprojektin epäonnistumiselle on monia syitä, joista yksi on huono ohjelmiston suunnittelu. [6]

Ohjelmistojen koon kasvaessa myös ohjelmistojen monimutkaisuus kasvaa. Tällöin ohjelmiston rakenteen tärkeys tulee esiin. Koodin uudelleenkäytettävyys ja ohjelmistokomponenttien selkeät vastuualueet parantavat ohjelmistojen ylläpidettävyyttä, nopeuttavat ohjelmistojen kehitystä ja helpottavat ohjelmistojen testausta. [13] Koodin uudelleenkäytettävyyteen ja ohjelmistokomponenttien selkeisiin vastuualueisiin voidaan vaikuttaa ohjelmistoarkkitehtuurien avulla.

Ohjelmistoarkkitehtuureja on moniin eri tarkoituksiin. Ensimmäiset ohjelmistoarkkitehtuurit on kehitetty aikana, jolloin niillä kuvattiin koko ohjelmiston toimintaa, mutta graafisten käyttöliittymien ja näihin liittyvien viitekehysten yleistyessä ohjelmistoarkkitehtuureja on jaettu useampaan osaan [11]. Työn tavoitteena on esitellä yleisimpiä graafisiin käyttöliittymiin liittyviä ohjelmistoarkkitehtuureja. Näitä arkkitehtuureja on monia erilaisia ja niillä jokaisella on omat vahvuudet ja heikkoudet ja ne soveltuvat eri tilanteisiin ja ohjelmistotyyppeihin. Tässä työssä selvitetään, millaisissa tapauksissa kannattaa käyttää mitäkin ohjelmistoarkkitehtuuria ja miksi.

Tämä työ toimii siis suuntaa antavana oppaana ohjelmistoarkkitehtuurin valintaan tiettyä ohjelmistotyyppiä varten. Ohjelmistoarkkitehtuurin soveltuvuus tullaan määrittelemään aikaisempien tutkimusten pohjalta. Työssä esitellään myös ohjelmistotyyppejä, jotka tyypillisesti toteutetaan käyttäen tiettyä ohjelmistoarkkitehtuuria.

Työssä esitellään aluksi yleisesti ohjelmistoarkkitehtuurin käsite ja tämän jälkeen yleisimmät graafisiin käyttöliittymiin liittyvät ohjelmistoarkkitehtuurit. Esiteltävät ohjelmistoarkkitehtuurit on valittu niiden suosion perusteella, mikä ilmenee suurena määränä tieteellistä aineistoa kyseistä arkkitehtuureista. Ohjelmistoarkkitehtuurit esitellään aikajärjestyksessä, koska valituille arkkitehtuureille ominaista on, että uusi arkkitehtuuri on kehitetty ratkaisemaan vanhan arkkitehtuurin jonkin osa-alueen ongelmat. Tämän jälkeen johtopäätökset-luvussa pohditaan teorian perusteella, millaisiin ohjelmistoihin työssä esitellyt ohjelmistoarkkitehtuurit soveltuvat. Lopuksi kerrataan lyhyesti, mitä työssä on käsitelty ja analysoidaan työn kattavuutta.

2. OHJELMISTOARKKITEHTUURIT

Ohjelmistoarkkitehtuurit ovat muodostuneet selväksi ohjelmistotekniikan osa-alueeksi vasta 1990-luvun loppupuolella. Tätä ennen ohjelmistoarkkitehtuureja on pidetty lähinnä korkeamman tason suunnitteluna. Nykyään ohjelmistoarkkitehtuurien merkitys on jatkuvassa kasvussa ja niistä julkaistaankin kokoajan lisää tieteellistä materiaalia. [13] Ohjelmistoarkkitehtuurin luonteen ja abstraktiuden vuoksi sille on monia erilaisia määritelmiä monissa eri lähteissä.

Tässä luvussa käsitellään ohjelmistoarkkitehtuurin ISO/IEC/IEEE:n standardin mukainen määrittely ohjelmistoarkkitehtuurille, jotta ohjelmistoarkkitehtuurille saadaan yksiselitteinen määritelmä. Myöhemmissä luvuissa käsiteltävät ohjelmistoarkkitehtuurit täyttävät tämän määritelmän. Luvun lopussa selvitetään graafisten käyttöliittymien tehtävä, niihin liittyvät ongelmat ja ohjelmistoarkkitehtuurien vaikutus graafisten käyttöliittymien toteuttamiseen.

2.1 Määritelmä

ISO/IEC/IEEE:n standardi ohjelmistoarkkitehtuureihin liittyen määrittelee ohjelmistoarkkitehtuurin seuraavalla tavalla. ”<*system*> *fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution*” [10]. Vapaasti suomennettuna tämä tarkoittaa sitä, että ohjelmistoarkkitehtuuri sisältää systeemin perimmäiset konseptit tai ominaisuudet järjestelmän osien ja niiden suhteiden muodossa sekä systeemin suunnitelman periaatteet ja evoluution.

Ohjelmistoarkkitehtuuri kertoo siis, mihin osa-alueisiin ohjelma jaetaan, miten nämä osat liittyvät toisiinsa, miten nämä osat kommunikoivat keskenään ja ympäristönsä kanssa ja mitkä ovat näiden osien vastuualueet. Järjestelmän osien väliset suhteet kehittyvät usein ohjelman suorituksen aikana ja näin ollen ohjelmistoarkkitehtuurien piiriin kuuluu rakenteen lisäksi myös järjestelmän käyttäytyminen [13]. Ohjelmistoarkkitehtuuri on siis erittäin laaja käsite ja pitää sisällään suuren määrän tietoa kehitettävästä ohjelmasta.

2.2 Graafiset käyttöliittymät

Tietyt ongelmat ohjelmistonkehityksessä ovat niin yleisiä, että niille on vakiintunut tietyt hyväksi todetut toteutustavat. Näistä ongelmista hyvä esimerkki on graafisten käyttöliittymien toteutus. Graafisten käyttöliittymien tehtävänä on esittää tieto käyttäjälle ja vastaanottaa käyttäjän syötteet [14]. Tämä pätee ohjelmistosta riippumatta kaikille graafisille käyttöliittymille, joten niille on kehitetty monia eri ohjelmistoarkkitehtuurimalleja.

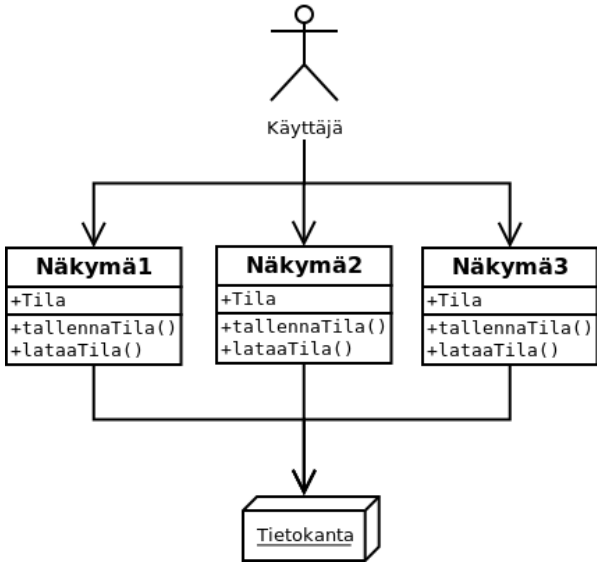
Graafisille käyttöliittymille ominaisia ongelmia ovat käyttöliittymänkomponenttien luonti ja yhdistäminen, käyttäjän syötteiden hallinta, käyttöliittymän vuorovaikutus muun ohjelmiston kanssa ja käyttöliittymän ja muun ohjelmiston yhdistäminen. Näistä kaksi ensimmäistä ongelmaa riippuvat yleensä käytetystä viitekehyksestä, joka määrittelee, miten käyttöliittymä luodaan ja miten käyttäjän syötteitä hallitaan. Käytetty viitekehys vaikuttaa siis ohjelmistoarkkitehtuurin valintaan. [12] Loput kaksi ongelmaa voidaan ratkaista ohjelmistoarkkitehtuurien avulla.

Koodin uudelleenkäytettävyyteen vaikuttaa käyttöliittymän ja muun ohjelmiston kommunikointi ja käyttöliittymän ja muun ohjelmiston yhdistäminen. Jos käyttöliittymä ja muu ohjelmisto on yhdistetty tiiviisti toisiinsa ja käyttöliittymä riippuu muusta ohjelmistosta, koodin uudelleen käyttäminen on erittäin haastavaa. Jos käyttöliittymä on tehty keveäksi ja muusta ohjelmistosta riippumattomaksi, niin kyseistä käyttöliittymää on helppo käyttää myös muissa ohjelmistoissa. Tällä tavalla myös ohjelmiston osien vastuualueet jakautuvat paremmin ja käyttöliittymää voidaan kehittää ohjelmistoprojektissa irrallaan muusta ohjelmistosta.

Seuraavissa luvuissa esitellään Smart UI -, malli–näkymä–ohjain- *(*engl. *model-view-controller,* lyh. MVC*)*, malli–näkymä–esittelijä-, (engl. *model-view-presenter,* lyh. MVP) ja malli–näkymä–näkymämalli- (engl. *model-view-viewmodel*, lyh. MVVM) arkkitehtuurit, joita esitellään useissa eri lähteissä [4][11][12][19][20]. Ennen ohjelmistoarkkitehtuurien yleistymistä käyttöliittymät toteutettiin Smart Ui -epäarkkitehtuurin kaltaisesti, mutta ohjelmistojen laajenemisen myötä graafisia käyttöliittymiä varten on kehitetty monia ohjelmistoarkkitehtuureja. Edellä mainitut ohjelmistoarkkitehtuurit on valittu tähän työhön, koska ne ovat yleisiä ja niiden käyttökohteet eroavat toisistaan. Näin saadaan laajempi kattaus eri käyttökohteita.

3. SMART UI

Ohjelmistoarkkitehtuurien avulla pyritään helpottamaan suurien projektien kehittämistä, mutta jos kyseessä on suhteellisen pieni ohjelmisto, kuten esimerkiksi yksinkertainen laskin, niin monimutkaisten ohjelmistoarkkitehtuurien käyttö vaikeuttaa ohjelmiston kehitystä. Tämä johtuu siitä, että luokkien määrä kasvaa tarpeettoman suureksi ohjelmiston kokoon nähden. Pienien ohjelmistojen yhteydessä Smart UI -arkkitehtuuri on hyvä ratkaisu, sillä se on yksinkertainen muihin yleisiin arkkitehtuurimalleihin verrattuna [19].



Kuva 1. Smart UI -arkkitehtuuri, perustuu lähteeseen [3].

Smart UI -arkkitehtuurin ideana on integroida koko ohjelmiston toiminta yhteen käyttöliittymään. Smart UI -arkkitehtuuri voidaan toteuttaa esimerkiksi olio-ohjelmoinnissa kirjoittamalla koko ohjelma yhden luokan sisään [12]. Tällä tavalla voidaan luoda toisistaan riippumattomia näkymiä, jotka voivat kommunikoida toistensa kanssa yhteisen tietokannan tai tietovaraston kautta. Jokainen näkymä toimii ikään kuin omana ohjelmanaan ja sisältää kaiken siihen liittyvän koodin. Smart UI -arkkitehtuuria sanotaankin usein epäarkkitehtuuriksi (engl. *anti-pattern*), sillä se ei sisällä lainkaan ohjelmiston vastuualueiden jakoa ja jokainen näkymä on yksilöllinen [19].

Kuvassa 1 esitellään Smart UI -arkkitehtuurin toteutus osana suurempaa ohjelmistoa. Kuvassa 1 tietokannan tiedolle on tehty kolme eri näkymää, jotka näyttävät tietokannan tiedon omalla tavallaan. Jokainen näkymä sisältää oman tilansa, käyttöliittymänsä ja logiikan tietokannan käsittelyyn.

Suurissa ohjelmistoissa Smart UI -arkkitehtuurin etuna on sen yksinkertaisuus, joka mahdollistaa valmiiden työkalujen käytön. Tämä nopeuttaa ohjelmistonkehitystä huomattavasti, koska näiden työkalujen avulla voidaan generoida huomattava osa ohjelmistosta hetkessä. Generoidut näkymät myös näyttävät samalta ja toimivat yhtäläisesti. Työkalujen käyttö tarkoittaa myös sitä, että ohjelmiston tekijän ei tarvitse tietää paljoa ohjelmoinnista. [3] Pieniä ohjelmistoja on helppoa ja nopeaa toteuttaa ilman työkaluja, mutta kun ohjelmiston koko kasvaa, niin kokonaisuuden hallinta vaikeutuu huomattavasti [19].

Suurin osa Smart UI -arkkitehtuurin hyödyistä saadaan työkalujen käytöstä, mutta näiden työkalujen käytöstä johtuu myös suurin osa haitoista. Automaattisesti generoitu koodi on usein erittäin vaikealukuista, mikä vaikeuttaa ohjelmiston ylläpidettävyyttä. Ohjelmiston muokkaaminen on vaikeaa, koska muutokset tulee tehdä työkalujen avulla. Helpompaa olisi tehdä muutokset suoraan näkymän lähdekoodiin, mutta jos koodi halutaan generoida uudelleen jonkin muutoksen takia, niin generaattori ylikirjoittaa lähdekoodiin tehdyt muutokset. Lisäksi jos näkymät kommunikoivat keskenään yhteisen tietokannan tai tietovaraston avulla, muutokset tietokantaan saattavat rikkoa koko ohjelman. [3] Edellä mainittujen syiden vuoksi Smart UI -arkkitehtuurilla tehdyn ohjelmiston ylläpito on erittäin haastavaa ja ohjelmiston toteutus on määriteltävä tarkasti hyvin aikaisessa vaiheessa, koska ohjelmistoa on vaikea muuttaa jälkeenpäin.

4. MALLI–NÄKYMÄ–OHJAIN

Malli–näkymä–ohjain-arkkitehtuurin (engl. *model-view-controller,* lyh. MVC) on alun perin kehittänyt 70-luvulla Trygve Reenskaug [11]. MVC-arkkitehtuuria käytettiin Smalltalk-80-kielessä käyttöliittymien rakentamiseen. Ennen MVC-arkkitehtuurin kehitystä tiedon säilytys, ohjelmiston toiminta ja käyttöliittymät niputettiin yhteen. MVC-arkkitehtuuri pyrkii erottamaan nämä toisistaan, jotta ohjelmistokomponenttien uudelleenkäytettävyys paranisi. [7]

MVC-arkkitehtuuri on vanha ja yksi tunnetuimmista ohjelmistoarkkitehtuureista, joka on muuttanut muotoaan ajan kuluessa. Tästä syystä MVC-arkkitehtuurin toiminta kuvataan eri tavalla eri lähteissä. MVC-arkkitehtuurin toteutus on riippuvainen toteutettavasta ohjelmasta. Esimerkiksi web-sovelluksen toteutus vaatii eri lähestymistapaa kuin työpöytäsovelluksen.

4.1 Toiminta

MVC-arkkitehtuurin kehityksen alussa sitä käytettiin kuvaamaan koko ohjelmistoa, mutta nykyään sen avulla toteutetaan ohjelmiston käyttöliittymäpuoli [11]. Alkuperäisessä MVC-arkkitehtuurissa ohjelmisto jaetaan kolmeen osaan. Malli (engl. *model*) sisältää ohjelmiston logiikan ja tiedon. Näkymä (engl. *view*) sisältää käyttöliittymän komponentit, niiden järjestyksen ja toiminnallisuuden. Ohjain (engl. *controller*) hallitsee käyttäjän syötteitä ja aktivoi syötteiden perusteella toimintoja mallista. [4]

Alkuperäisen MVC-arkkitehtuurin kehittämisen aikana käyttäjän syötteet luettiin suoraan näppäimistön ja hiiren muistiosoitteista, mutta nykyään näitä toimintoja varten on monia viitekehyksiä. Tästä syystä MVC-arkkitehtuurin ohjaimelle annetaan vastuuta ohjelmiston logiikasta. Tällöin MVC-arkkitehtuurin toteutus muistuttaa luvussa 5 läpi käytävää MVP-arkkitehtuuria. Tässä luvussa kuvataan alkuperäinen MVC-arkkitehtuuri.

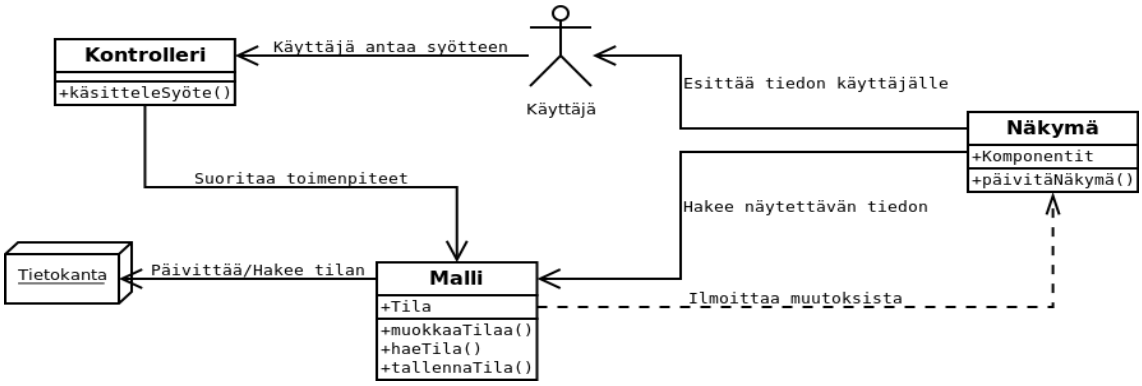
4.1.1 MVC-arkkitehtuurin osat

Nykyään MVC-arkkitehtuurin osien vastuualueet ovat hieman muuttuneet sen alkuajoista. Mallilla kuvataan yleensä järjestelmän käyttöliittymän tilaa, joka sisältää sen hetkisen käyttöliittymän tarvitseman tiedon ja logiikan tämän tiedon käsittelyyn [4]. Tämän lisäksi malli on yhteydessä muuhun ohjelmistoon, josta malli saa käyttöliittymän

tarvitseman tiedon. Jos esimerkiksi kaikkien järjestelmän käyttäjien tiedot on tallennettu relaatiotietokantaan ja käyttäjälle halutaan esittää vain käyttäjän omat tiedot, malliin on turha hakea kaikkien käyttäjien tietoja. Tässä tapauksessa mallin avulla päästään käsiksi tietokannan tietoihin ja näitä tietoja voidaan muokata mallin välityksellä.

MVC-arkkitehtuurin näkymä määrittelee, miten tieto esitetään käyttäjälle [12]. Näkymä siis sisältää käyttöliittymän komponentit, komponenttien suhteet toisiinsa, komponenttien sijainnit ja komponenttien ulkonäön. Näkymä saa näytettävän tiedon mallilta ja määrittelee, miten se esitetään käyttäjälle ja millaisia mahdollisuuksia käyttäjällä on muokata tätä tietoa. Malli ei saa kuitenkaan olla suoraan yhteydessä näkymään, sillä tämä sitoo mallin vain yhteen näkymään [18]. Useampia näkymiä yhdelle mallille voidaan tarvita esimerkiksi, jos halutaan esittää tietty tieto pylväsdiagrammina ja kuvaajana. Tällöin voidaan luoda molemmille esitystavoille omat näkymät.

Ohjain vastaanottaa käyttäjän syötteet ja tekee malliin tarvittavat muutokset mallin operaatioiden avulla [11]. Malli sisältää logiikan tiedon käsittelyyn, mutta joissain tapauksissa osa tästä logiikasta sijaitsee ohjaimessa. Ohjain voi esimerkiksi muuttaa mallin tilaa ja käskeä mallia tallentamaan tila tietokantaan.



Kuva 2. MVC-arkkitehtuuri, perustuu lähteeseen [4].

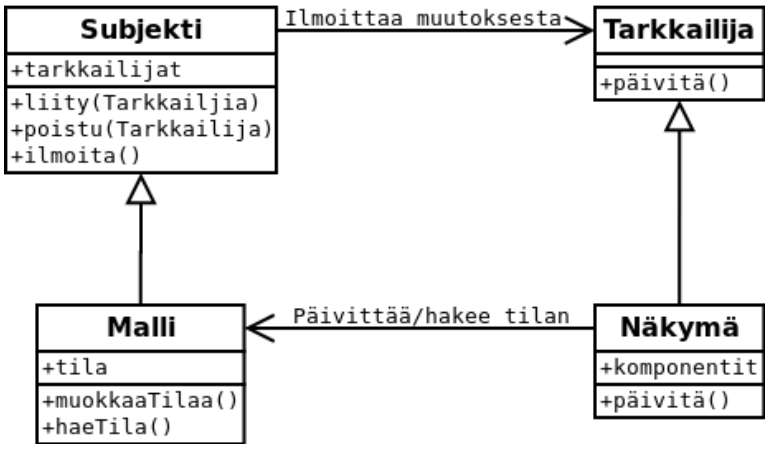
Kuvassa 2 esitellään MVC-arkkitehtuuri ja sen komponenttien suhteet ja vastuualueet. Kuvassa 2 näkymä hakee tiedon mallilta ja määrittelee, miten tieto esitetään käyttäjälle. Ohjain vastaanottaa ja käsittelee käyttäjän syötteen ja toteuttaa halutut toimenpiteet mallille. Ohjaimen rooli riippuu käytetystä viitekehyksestä.

Malli toimii käyttöliittymän rajapintana muuhun ohjelmistoon. Kuvan 2 tapauksessa malli toimii rajapintana tietokannalle ja se pystyy hakemaan käyttöliittymän tarvitseman tiedon tietokannasta, tekemään tietoon muutoksia ja tallentamaan uuden tiedon tietokantaan. Malli ilmoittaa näkymille muutoksista ja on siten epäsuorasti yhteydessä näkymään. Tästä epäsuorasta yhteydestä kerrotaan enemmän seuraavassa luvussa.

4.1.2 MVC-arkkitehtuurin osien välinen kommunikointi

Monen näkymän luominen yhtä mallia kohden synnyttää ongelman. Jos jonkin näkymän kautta muokataan mallia, muut näkymät eivät tiedä, että mallin tila on muuttunut, ja uusi tila ei päivity kaikkiin näkymiin. Tässä tapauksessa mallin tulee ilmoittaa näkymilleen, että sen tila on muuttunut. Malli ei voi kuitenkaan olla suoraan yhteydessä näkymään, sillä tämä sitoisi mallin vain yhteen näkymään. Toinen vaihtoehto on, että näkymät ilmoittavat toisilleen, kun malliin tehdään muutoksia, mutta tämä sitoo näkymät toisiinsa, eikä näkymiä voitaisi käyttää enää toisistaan riippumattomasti. Tähän ongelmaan ratkaisu on tarkkailija-suunnittelumalli. [18]

Tarkkailija-suunnittelumalli koostuu subjektista ja tarkkailijasta. Subjektiin voi liittyä mielivaltainen määrä tarkkailijoita. Subjekti ilmoittaa kaikille liittyneille tarkkailijoille, kun sen tilassa tapahtuu muutos. Tällöin tarkkailijat tietävät, että pitää pyytää subjektilta uusi tila. [7] Kun malli periytetään subjektista ja näkymät tarkkailijasta, mallin ei tarvitse tietää mitään näkymästä, mutta voi silti ilmoittaa kaikille liittyneille näkymille tilan muutoksesta. Edellä mainitulla tavalla tarkkailija-suunnittelumalli mahdollistaa usean näkymän luomisen yhtä mallia kohden.



Kuva 3. Tarkkailija-suunnittelumalli, perustuu lähteeseen [7].

Kuvassa 3 esitellään, miten malli voi olla epäsuorasti yhteydessä näkymään käyttäen tarkkailija-suunnittelumallia. Malli on periytetty subjektista ja näkymä tarkkailijasta. Tämän jälkeen useita näkymiä voi liittyä yhteen malliin ja malli voi ilmoittaa jokaiselle siihen liittyneelle näkymälle muutoksista. Ilmoitus on kuitenkin yksinkertainen eikä kuvan 3 tapauksessa välitä parametreja näkymälle, vaan näkymän tulee hakea uusi tila mallista.

4.2 Erityispiirteet

MVC-arkkitehtuurin etuna on sen mallin ja näkymän erottelu. Tämä mahdollistaa useamman näkymän luomisen yhtä mallia kohden ja mallin uudelleenkäytön melkein jokaisessa tilanteessa. Monet näkymät toimivat yhden mallin kanssa, joka sisältää tarvittavan tiedon ja logiikan mallin käyttöön, jolloin logiikkaa ei tarvitse kirjoittaa jokaiseen näkymään erikseen. Näin saman koodin kirjoittaminen moneen eri paikkaan vähentyy huomattavasti. Tämä oli erityisenä ongelmana Smart Ui -arkkitehtuurissa, jossa jokainen näkymä sisälsi ohjelmiston logiikan. [18]

Yksikkötestien kirjoittaminen logiikalle helpottuu huomattavasti, koska ohjelmiston logiikka on kirjoitettu yhteen paikkaan ja se on erillään käyttöliittymästä. Mallin ja näkymien erottelu mahdollistaa järkevän työjaon ohjelmistoprojektissa. Ohjelmistoprojektin logiikkaan ja grafiikkaan keskittyneet työntekijät voivat työskennellä rinnakkain ja toisistaan riippumattomasti. Tämä nopeuttaa ohjelmistonkehitystä, sillä grafiikan ei tarvitse olla valmista ennen logiikkaa vaan niitä voidaan työstää samanaikaisesti. [18]

MVC-arkkitehtuuri kuitenkin monimutkaistaa ohjelmistoa huomattavasti, koska sen toteutus vaatii huomattavan määrän luokkia. MVC-arkkitehtuurissa näkymä ja ohjain on yhdistetty kiinteästi toisiinsa, mikä johtaa siihen, että ohjainta ja näkymää on vaikea käyttää irrallaan toisistaan. [13]

Tarkkailija-suunnittelumallin soveltaminen aiheuttaa suurimman osan MVC-arkkitehtuurin haitoista. Tarkkailija-suunnittelumallin soveltamisen takia näkymille voi kertyä paljon päivittämiskutsuja, joihin ei välttämättä tarvitse reagoida mitenkään ja tämä johtaa turhaan työhön. Tällainen tilanne tapahtuu esimerkiksi kun kahdesta näkymästä toisessa esitetään kurssin opiskelijoiden keskimääräinen arvosana ja toisessa näkymässä opiskelijoiden kurssin läpipääsyprosentti. Jos yhden opiskelijan arvosana muutetaan kolmosesta neloseen, niin vain keskimääräisen arvosanan tarvitsee muuttua, mutta päivityskutsu menee myös näkymälle, joka näyttää läpipääsyprosentin. [13]

Mallin ilmoituksen jälkeen jokaisen näkymän on kysyttävä mallilta muuttunutta tietoa käyttäen mallin yleisiä operaatioita, mikä voi olla hidasta. Esimerkiksi, jos näkymän tarvitsee esittää tietyn opiskelijan tiedot kaikista yliopiston opiskelijoista, niin tämän opiskelijan etsiminen voi olla raskas operaatio ja se pitää suorittaa jokaisella kerralla kun mallin tilaa muutetaan. [13]

Mallin luonteen takia näkymät joutuvat suorittamaan hyvin samankaltaisia operaatioita mallin muutoksen jälkeen. Esimerkiksi jos sama tieto näytetään kahdessa eri näkymässä, molemmat näkymät joutuvat erikseen pyytämään tämän tiedon. Tämä johtaa turhaan työhön ja voi olla erittäin raskasta. [13]

5. MALLI–NÄKYMÄ–ESITTELIJÄ

MVC-arkkitehtuurin tavoin malli–näkymä–esittelijä-arkkitehtuurista (engl. *model-view-presenter*, lyh. MVP) on useita eri variaatioita riippuen käyttökohteesta. MVP-arkkitehtuuri on ensimmäistä kertaa esitetty Taligent käyttöjärjestelmän yhteydessä Potelin toimesta vuona 1996 [17]. Taligent MVP -arkkitehtuurissa ideana oli jakaa käyttöliittymien toteuttaminen tiedon käsittelyyn ja käyttöliittymään [17].

Taligent MVP -arkkitehtuurista inspiroituneina Bower ja McGlashan esittelivät vuonna 2000 oman versionsa MVP-arkkitehtuurista Dolphin Smalltalk -kieltä varten. MVC-arkkitehtuuri ei soveltunut sellaisenaan Dolphin Smalltalk -kieleen, joka toimi Windows-ympäristössä. Windows tarjoaa omia käyttöliittymäkomponentteja, jotka toteuttavat käyttöliittymän ja käyttäjän syötteiden lukemisen. Tässä tapauksessa MVC-arkkitehtuurin ohjain on siis turha. Toisena ongelmana oli, että malli ei voinut olla suoraan yhteydessä näkymään, jolloin tietyt yksinkertaiset operaatiot olivat vaikeita toteuttaa. [2]

MVP-arkkitehtuurin käytön yleistyessä siitä on kehitetty myös malli, jossa näkymä on täysin passiivinen. Tämä kehitettiin helpottamaan käyttöliittymien testaamista siirtämällä käyttöliittymän logiikka näkymästä esittelijään. [4]

5.1 Toiminta

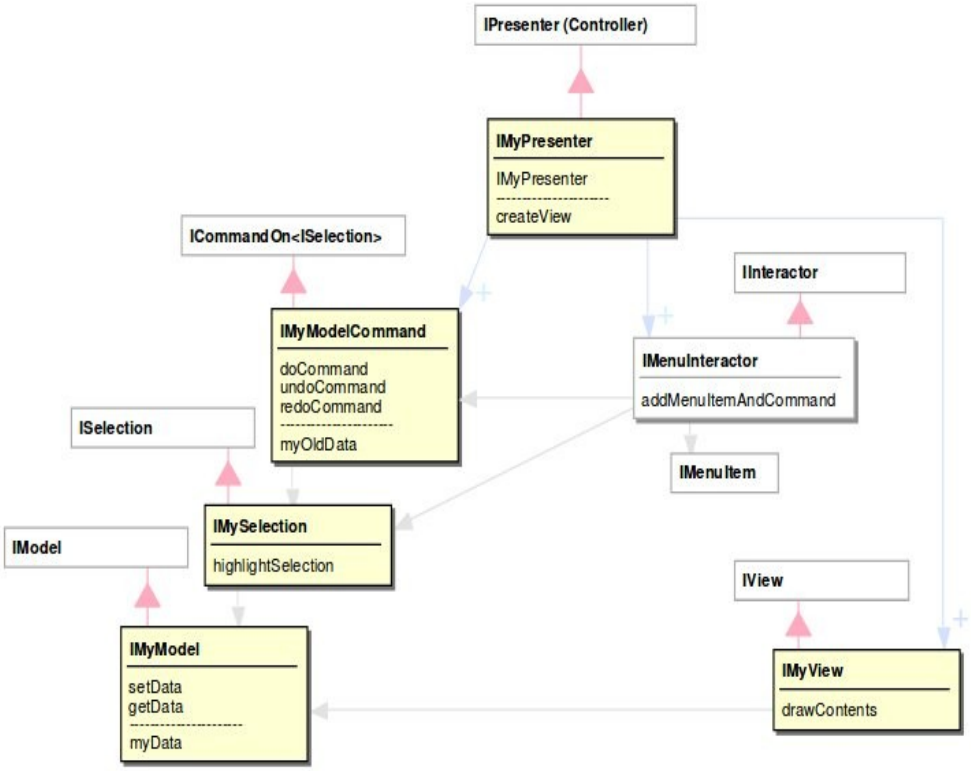
MVP-arkkitehtuuri on muunnelma MVC-arkkitehtuurista. MVP-arkkitehtuurista on useita eri versioita, mutta jokaisessa versiossa ohjelmiston komponentit ovat samat. MVP-arkkitehtuuri koostuu mallista (engl. *model*), näkymästä (engl. *view*) ja esittelijästä (engl. *presenter*). Suurimmat erot MVP-arkkitehtuurin eri versioiden välillä löytyvät esittelijän vastuualueista. [2][4][17]

Tässä luvussa esitellään kolme MVP-arkkitehtuurin variaatiota. Variaatiot on esitelty ikäjärjestyksessä vanhimmasta uusimpaan. Taligent MVP-arkkitehtuurin toiminta ja siihen liittyvä komento-suunnittelumalli avataan ensimmäisenä. Tämän jälkeen siirrytään Dolphin MVP-arkkitehtuuriin, joka on kehitetty Taligent MVP-arkkitehtuurin pohjalta. Lopuksi esitellään passiivisen näkymän MVP-arkkitehtuuri, jossa näkymä ei ole lainkaan yhteydessä malliin.

5.1.1 Taligent MVP-arkkitehtuuri

Taligent MVP -arkkitehtuurin versiossa käyttöliittymän toteuttaminen jaetaan tietojen käsittelyyn ja käyttöliittymään. Malli toimii samalla tavalla kuin MVC-arkkitehtuurissa ja kuuluu tietojen käsittelyyn. Näkymä ja ohjain yhdistetään ja ohjaimen tilalle luodaan esittelijä. Näkymä ja esittelijä kuuluvat käyttöliittymään. Esittelijä toimii samalla tavalla kuin ohjain, mutta se toteutetaan yhden näkymän sijaan koko ohjelmiston tasolla. [17]

Taligent MVP -arkkitehtuurin versiossa esittelijän rooli on luoda näkymät ja asettaa oikeat käskyt oikeille käyttöliittymä komponenteille. Käyttäjän syötteet käsitellään vuorovaikuttajien (engl. *interactor*) avulla, jotka sisältävät tiedon, mitä on valittu ja mikä komento suoritetaan. Vuorovaikuttaja kuuluu käyttöliittymään. Mallin käsittely tapahtuu käskyjen (engl. *commands*) ja valintojen (engl. *selections*) välityksellä, jotka määritellään kuuluvaksi tiedon käsittelyyn. Nämä vuorovaikuttajat, käskyt ja valinnat toteutetaan käytännössä komento-suunnittelumallin avulla. [17]



Kuva 4. Taligent MVP -arkkitehtuuri [17].

Komento-suunnittelumalli on malli, jossa suoritettava toiminto koostetaan omaksi oliokseen. Tällä tavalla käskyjä voidaan lähettää ja vastaanottaa ilman, että lähettäjän tai vastaanottajan tarvitsee tietää, mitä toimintoa ollaan suorittamassa. Näin sama toiminto voidaan helposti toteuttaa monia eri väyliä pitkin. Sama toiminto voidaan esimerkiksi

suorittaa jonkin käyttöliittymän läpi tai näppäinpainalluksella antamalla molemmille toiminnoille sama käskyolio suoritettavaksi. Toimintoja voidaan myös vaihtaa dynaamisesti ohjelman suorituksen aikana vaihtamalla suoritettava olio. [7] Komento-suunnittelumallin avulla koodin uudelleen kirjoittamista voidaan vähentää, sillä logiikka on kirjoitettu yhteen luokkaan, jota voidaan uudelleen käyttää kaikissa vuorovaikuttajissa. Kuvassa 4 esimerkiksi IMyModelCommand-luokka toteuttaa komento-suunnittelumallin käskyn.

Kuvassa 4 on esitelty yksinkertainen ohjelmisto, joka on toteutettu Taligent MVP -arkkitehtuuria käyttäen. Kuvasta 4 käy ilmi yksinkertaisen ohjelman komponentit, vastuualueet ja yhteydet komponenttien välillä. Esittelijä vastaa kuvan 4 tapauksessa käyttöliittymän rakentamisesta. Esittelijä luo tarvittavat näkymät, käyttöliittymäkomponentit, vuorovaikuttajat ja käskyt, mutta ei kommunikoi suoraan näkymän kanssa käyttöliittymän rakentamisen jälkeen. [17]

Esittelijä lisää vuorovaikuttajat tiettyihin käyttöliittymäkomponentteihin. Vuorovaikuttajille asetetaan omat valitsijat ja käskyt, jotka vuorovaikuttaja voi suorittaa kun käyttäjä aktivoi käyttöliittymäkomponentin. Komento-suunnittelumalli mahdollistaa minkä tahansa käskyn asettamisen vuorovaikuttajalle. [17]

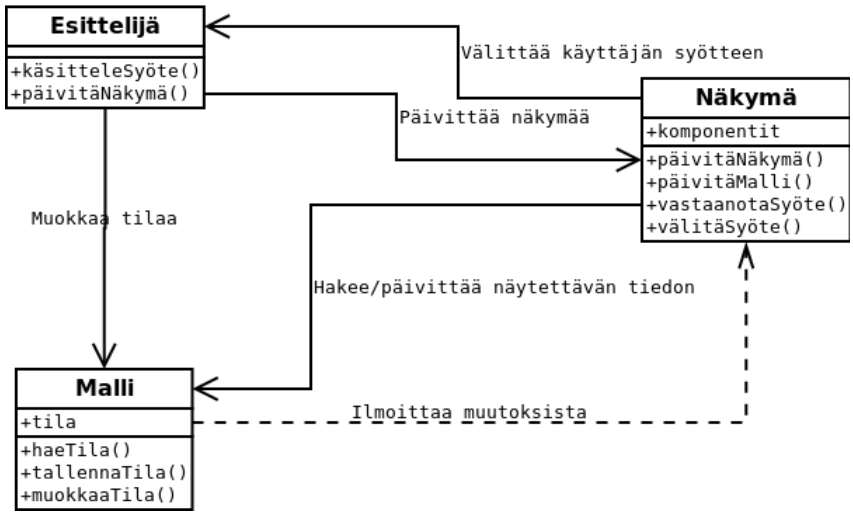
Valitsijat sisältävät tiedon siitä, mille mallin tiedolle käsky suorittaa sille määritellyn toimenpiteen. Näkymä voi esimerkiksi olla taulukko, joka esittää mallin sisältämää tietoa. Valitsija osoittaa tiettyyn soluun, jonka perusteella käsky tietää mille solulle se suorittaa sille määritellyn operaation. Tässä tapauksessa käskyn operaatio voi olla esimerkiksi solun sisällön muokkaaminen käyttäjän syötteiden perusteella. [17]

5.1.2 Dolphin Smalltalk MVP-arkkitehtuuri

Joissain tapauksissa mallista on hyvä päästä käsiksi näkymään [2]. Esimerkiksi jos halutaan, että näkymässä esitellään opiskelijoiden kurssin läpipääsyprosentti eri väreillä riippuen läpipääsyprosentin arvosta. Alhainen läpipääsyprosentti haluttaisiin esittää punaisena ja korkea vihreänä. Läpipääsyprosentin värin määrittely on visuaalinen toimenpide ja siten se kuuluu näkymän vastuualueeseen. Alhaisen ja korkean läpipääsyprosentin määrittely kuuluu mallin vastuualueeseen. Mallin tulee siis ilmoittaa erikseen näkymälle, milloin läpipääsyprosentti on alhainen tai korkea, jotta näkymä tietää, että millä värillä läpipääsyprosentti esitetään. Tämän yksinkertaisen toiminnon toteutus vaikeutuu huomattavasti, koska malli ei voi olla suoraan yhteydessä näkymään vaan edellä mainittu toiminto tulee toteuttaa tarkkailija-suunnittelumallin avulla [2].

Dolphin Smalltalk MVP -arkkitehtuurissa tämä ongelma ratkaistiin antamalla esittelijälle suora yhteys näkymään. Tällöin esittelijä voi tietyissä tapauksissa päivittää näkymää. Värin määrittely voidaan siis lisätä esittelijään, joka asettaa näkymän värin läpipääsyprosentin arvon mukaan. Esittelijän vastuualuetta siis laajennettiin näkymän luomisen lisäksi näkymän käsittelyyn. [2]

Dolphin Smalltalk MVP -arkkitehtuurissa mallilla ei ole mitään tietoa näkymästä ja se säilyttää käyttöliittymän tiedon. Näkymän tarkoitus on esittää mallilta saatu tieto käyttäjälle. Malli ilmoittaa muutoksista näkymälle tarkkailija-suunnittelumallin avulla ja tämä mahdollistaa useamman näkymän yhtä mallia kohden. Näkymän odotetaan kuitenkin käsittelevän osan käyttäjän syötteistä ja tekevän muutoksia suoraan malliin ilman esittelijää. Suurin osa mallin käsittelystä tapahtuu kuitenkin esittelijässä, joka määrittelee, miten mallia voidaan käsitellä. Esittelijä on suoraan yhteydessä malliin ja näkymään ja sisältää suurimman osa ohjelmiston logiikasta. Esittelijän ja näkymän suoran yhteyden takia esittelijä voi vaikuttaa, miten tieto esitetään käyttäjälle. [2]



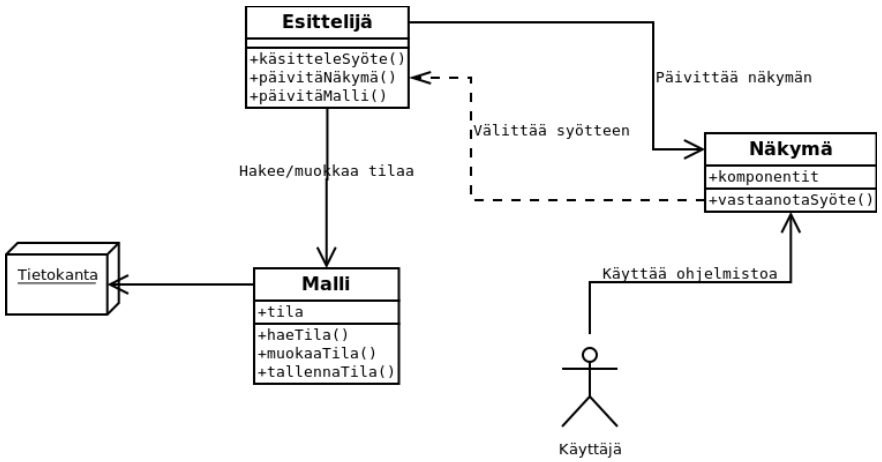
Kuva 5. Dolphin Smalltalk MVP -arkkitehtuuri, perustuu lähteeseen [2].

Kuvassa 5 on esitelty Dolphin Smalltalk MVP -arkkitehtuuri sen komponenttien ja komponenttien välisten yhteyksien muodossa. Näkymä sisältää käyttöliittymäkomponentit ja malli sisältää käyttöliittymän tilan. Näkymä hakee käyttäjälle esitettävän tiedon mallista, vastaanottaa käyttäjän syötteen ja tekee osan mallin päivityksestä. Monimutkaisemmat käskyt välitetään esittelijälle, joka käsittelee ne ja tekee tarvittavat muutokset malliin ja näkymään. Malli hakee tilan käyttöliittymän ulkopuolelta, kuten esimerkiksi tietokannasta ja ilmoittaa tilan muutoksesta näkymälle.

5.1.3 Passiivinen näkymä

MVP-arkkitehtuurista on kehitetty myös versio, jossa näkymä on täysin passiivinen [4]. Tässä versiossa esittelijä hallitsee näkymää ja kaikki mallin ja näkymän kanssakäyminen tapahtuu esittelijän kautta. Näkymä vastaanottaa käyttäjän syötteet ja välittää ne esittelijälle tarkkailija-suunnittelumallin avulla. Esittelijä käsittelee näkymältä saadut syötteet, tekee tarvittavat muutokset malliin ja päivittää näkymän tilan. [5]

Passiivisen näkymän MVP-arkkitehtuurissa näkymä saa näytettävän tiedon esittelijältä eikä ole millään tavalla suoraan yhteydessä malliin. Tällä tavalla ohjelmiston toteutuksessa ei tarvita suoraa tai epäsuoraa yhteyttä mallin ja näkymän välillä. Näkymä ei sisällä lainkaan sen toimintaan vaikuttavaa logiikkaa vaan esittelijä hallitsee näkymää. Näkymä voi esimerkiksi sisältää painike-komponentin, jonka sijainnin näkymä määrittelee, mutta painikkeen tilan määrittelee esittelijä. [5]



Kuva 6. Passiivisen näkymän MVP-arkkitehtuuri, perustuu lähteeseen [5].

Kuvassa 6 on esitelty passiivisen näkymän MVP-arkkitehtuuri. Näkymä sisältää käyttöliittymäkomponentit ja niiden sijainnit, mutta ei lainkaan käyttöliittymän logiikkaa. Näkymä vastaanottaa käyttäjän syötteet ja välittää ne tarkkailija-suunnittelumallin avulla esittelijälle, joka käsittelee syötteet. Syötteiden pohjalta esittelijä päivittää mallia ja näkymää. Näytettävän tiedon näkymä saa esittelijältä, joka saa näytettävän tiedon mallilta, johon käyttöliittymän tila on tallennettu. Malli toimii rajapintana muuhun ohjelmistoon. Kuvan 6 tapauksessa malli toimii rajapintana tietokannalle.

5.2 Erityispiirteet

Taligent MVP-arkkitehtuurin näkymät ovat riippumattomia esittelijästä tai mallista, joten samalle sovellukselle voidaan luoda monia erilaisia näkymiä. Valitsijoiden luonteen vuoksi malli on riippumaton muusta ohjelmasta, jolloin tiedon tallentamistapaa voidaan muuttaa ilman, että se vaikuttaa tiedon käsittelyyn tai sen esittämiseen. Tämä mahdollistaa mallin muuttamisen alustasta tai kohdemaasta riippuen siten, että muuta ohjelmistoa ei tarvitse muuttaa. Komentojen luonteen vuoksi toimintoja voidaan käyttää monissa eri saman arkkitehtuurin ohjelmistoissa, mikä lisää koodin uudelleenkäytettävyyttä. Komennot suoritetaan valitsijoiden avulla, joten sama komento voidaan toteuttaa yhdelle tai usealle elementille ilman, että komennolle tarvitsee kirjoittaa erikoistapauksia usean elementin käsittelylle. Vuorovaikuttajien luonteen vuoksi Taligent MVP-arkkitehtuurissa voidaan vaihtaa käyttöliittymä komponentteja ilman, että se vaikuttaa siihen, miten syötteet käsitellään. [17]

Dolphin Smalltalk MVP-arkkitehtuurin etuna on sen esittelijän ja näkymän yhteys. Tämä helpottaa tietyn tyyppisten ominaisuuksien toteutuksen, koska osa näkymän toiminnallisuudesta voidaan suorittaa esittelijästä käsin. [2] Tällaisesta ominaisuudesta ja toteutuksesta on esitetty esimerkki luvussa 5.1.2. Mallin ja näkymän epäsuora yhteys tarkkailija-suunnittelumallin avulla aiheuttaa samoja hyötyjä ja haittoja kuin MVC-arkkitehtuurissakin. Tarkkailija-suunnittelumalliin liittyvistä hyödyistä ja haitoista on puhuttu MVC-arkkitehtuurin erityispiirteiden yhteydessä luvussa 4.2.

Passiivisen näkymän etuna on sen testattavuus. Tämä johtuu siitä, että kaikki käyttöliittymän toiminallisuus sijaitsee esittelijässä, jolloin ei tarvita monimutkaista graafisen käyttöliittymän läpi testaamista, vaan näkymä voidaan korvata testausmoduulilla, joka tekee testit käyttäen esittelijän metodeja. Passiivinen näkymä poistaa myös kaikki tarkkailija-suunnittelumalliin liittyvät heikkoudet, koska passiivisen näkymän toteutuksessa tarkkalija-suunnittelumallin avulla ei ilmoiteta enää mallin muutoksista näkymään, vaan välitetään käyttäjän syöte esittelijälle, joka päivittää näkymät. Siis tieto haetaan mallista vain kerran ja vain tarvittava tieto päivitetään kuhunkin näkymään. Passiivinen näkymä on myös paljon stabiilimpi, koska näkymä ei voi tehdä enää suoraan muutoksia malliin, vaan kaikki muutokset malliin tapahtuvat esittelijän kautta. [5]

6. MALLI–NÄKYMÄ–NÄKYMÄMALLI

Malli–näkymä–näkymämalli-arkkitehtuuri (engl. *model-view-viewmodel*, lyh. MVVM) on alunperin esitelty John Gossmanin blogissa vuonna 2005 [20]. MVVM-arkkitehtuuri on suunniteltu moderneja ohjelmistoprojekteja varten, joissa työntekijöiden vastuualueet on jaettu käyttöliittymän suunnitteluun ja ohjelmiston toimintaan. Tässä tapauksessa käyttöliittymän suunnittelulla tarkoitetaan ohjelmiston ja käyttäjän rajapintaa. Käyttöliittymän suunnittelu vaatii eri taitoja kuin ohjelmiston toiminnan toteuttaminen. Tästä syystä nämä kaksi osa-aluetta halutaan erottaa toisistaan. [9]

Käyttöliittymät toteutetaan nykyään yleensä paljon yksinkertaisemmalla kielellä kuin ohjelmisto, kuten esimerkiksi HTML, XAML tai QML. Käyttöliittymät voidaan suunnitella ja toteuttaa myös käyttäen työkaluja. [9] Työkalujen avulla käyttöliittymän suunnittelija voi suunnitella käyttöliittymän tietämättä mitään sen todellisesta toteutuksesta.

Käyttöliittymä ja ohjelmisto toteutetaan siis eri työkalujen avulla. MVVM-arkkitehtuuri on kehitetty MVC-arkkitehtuurin pohjalta pitäen mielessä käyttöliittymän ja ohjelmiston toteutuksen erot. MVC-arkkitehtuurissa ohjelmisto ja käyttöliittymä on kehitetty käyttäen samaa kieltä ja kehitysympäristöä. [9]

6.1 Toiminta

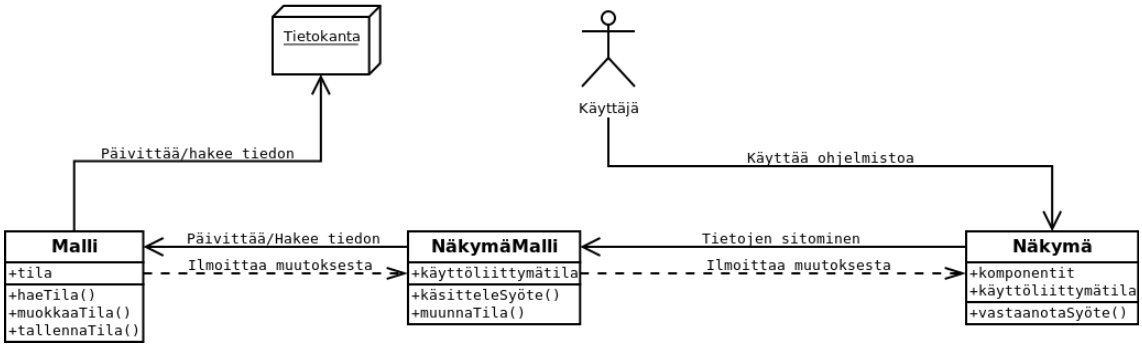
MVVM-arkkitehtuuri koostuu kolmesta komponentista. Mallista (engl. m*odel*), näkymästä (engl. *view*) ja näkymämallista (engl. *viewmodel*) [9]. MVVM-arkkitehtuuri on toiminnaltaan hyvin samankaltainen MVC- ja MVP-arkkitehtuurien kanssa.

MVVM-arkkitehtuurissa malli toimii samalla tavalla kuin MVC- ja MVP-arkkitehtuurissa. Sen vastuualueisiin kuuluu siis käyttöliittymän tarvitseman tiedon tallennus ja tämän tiedon muokkaamiseen tarvittavan logiikan toteutus. Lisäksi malli on täysin käyttöliittymäriippumaton. [9] Malli ei ole millään tavalla tietoinen näkymästä ja kaikki muokkaukset malliin tapahtuvat näkymämallin kautta [20].

Näkymä sisältää kaikki käyttöliittymäkomponentit ja vastaanottaa käyttäjän syötteet. Käyttöliittymät on toteutettu melkein aina deklaratiivisilla kielillä tai työkalujen avulla. [9] Ideaalisessa tapauksessa näkymä ei sisällä lainkaan logiikkaa [16]. Näkymä ei ole millään tavalla tietoinen mallista ja kaikki tieto näkymään tulee näkymämallin kautta [20].

Näkymä on siis usein kirjoitettu eri kielellä kuin malli, mutta näkymän on jollain tavalla saatava käyttäjälle näytettävä tieto mallilta. Tämä mahdollistetaan näkymämallin avulla, joka luo rajapinnan näkymälle, jonka kautta näkymää voidaan hallita [9]. Näkymämalli on kirjoitettu samalla kielellä kuin malli ja on vastuussa näkymän logiikasta, mallin tiedon paljastamisesta ja muokkaamisesta näkymälle sopivaksi ja käyttäjän syötteiden käsittelystä. Näkymämalli vastaanottaa käyttäjän syötteet näkymältä, käsittelee ne, tekee tarvittavat toimenpiteet mallille ja päivittää näkymän. [16]

Suurin ero passiivisen näkymän MVP-arkkitehtuurin ja MVVM-arkkitehtuurin välillä on näkymän ja näkymämallin välinen kommunikointi, joka tapahtuu tietojen sitomisella (engl. *data binding*) [9]. Tietojen sitomiseen kuuluu kaksi osapuolta. Nämä osapuolet ovat kohde (engl. *binding target*) ja lähde (engl. *binding source*). Kohde sisältää jonkin ominaisuuden, johon lähde haluaa sitoutua. Tietojen sitomista on kolme eri tyyppiä, yksisuuntainen tietojen sitominen, kaksisuuntainen tietojen sitominen ja yksisuuntainen tietojen sitominen lähteeseen. [15]



Kuva 7. MVVM-arkkitehtuuri, perustuu lähteeseen [16].

Yksisuuntaisessa tietojen sitomisessa kohteen ominaisuus päivittyy automaattisesti kun lähteen ominaisuutta muutetaan, mutta jos kohteen ominaisuutta muutetaan, niin lähteen ominaisuus ei päivity. Kaksisuuntaisessa tietojen sitomisessa kummankin osapuolen ominaisuuden muutos vaikuttaa toiseen osapuoleen. Kolmas tietojen sitomisen tyyppi on yksisuuntaisuus lähteeseen, jossa kohteen ominaisuuden muuttaminen päivittyy lähteen ominaisuuteen. [15]

Tietojen sitominen tapahtuu yleensä paljastamalla jokin kohteen sisältämä arvo lähteelle siten, että lähde voi kutsuta ja muokata kohteen arvoa. Kaksisuuntaisuus mahdollistetaan tarkkailija-suunnittelumallin avulla. Kun kohteen arvoa muutetaan, niin lähetetään ilmoitus lähteelle, että arvo on muuttunut, jolloin lähde osaa päivittää oman arvonsa. [15]

Kuvassa 7 esitetään MVVM-arkkitehtuurin komponentit ja niiden suhteet. Käyttäjä käyttää ohjelmistoa näkymän kautta, joka esittää tiedon ja vastaanottaa syötteet. Näkymä saa näytettävän tiedon näkymämallilta, johon näkymä on yhteydessä tietojen sitomisen avulla.

Kuvassa 7 näkymämalli on epäsuorasti yhteydessä näkymään. Tämä epäsuoruus johtuu tietojen sitomisesta. Kun näkymämallin jokin ominaisuus muuttuu, lähetetään tästä muutoksesta ilmoitus näkymälle tarkkailija-suunnittelumallin avulla. Ilmoitus lähetetään erikseen jokaisen ominaisuuden muutoksesta ja ilmoitus on yksilöllinen jokaiselle ominaisuudelle, jolloin näkymä tietää tarkalleen, mikä ominaisuus on muuttunut [15].

Kuvasta 7 käy ilmi, että käyttöliittymän tila on tallennettu näkymän lisäksi myös näkymämalliin. Tämä johtuu tietojen sitomisesta. Näkymämalli hakee tarvittavan tiedon mallista, muuntaa tiedon näkymälle sopivaksi ja päivittää käyttöliittymän tilan. Näkymämallin käyttöliittymän tila päivittyy näkymään tietojen sitomisen avulla.

Näkymässä esitettävän tiedon näkymämalli saa mallista, joka puolestaan hakee tiedon kuvan 7 tapauksessa tietokannasta. Näkymämalli käsittelee näkymältä saadut syötteet, tekee tarvittavat muutokset malliin ja käyttöliittymän tilan.

6.2 Erityispiirteet

MVVM-arkkitehtuurin käyttö helpottaa ohjelmiston ylläpidettävyyttä, koska ohjelmisto on jaettu osa-alueisiin siten, että yhden osa-alueen muuttaminen ei vaikuta muihin. MVVM-arkkitehtuurin osa-aluejako mahdollistaa myös komponenttien vaihtamisen ohjelmistoprojektin myöhemmissä vaiheissa tai jopa ohjelman suorituksen aikana. [16]

MVVM-arkkitehtuurin näkymän passiivisuuden vuoksi MVVM-arkkitehtuurin avulla tehtyjä ohjelmistoja on helppo testata, koska näkymä voidaan korvata testausmoduulilla, jolloin ohjelmistoa ei tarvitse testata graafisen käyttöliittymän läpi [8][16][20]. Näkymän passiivisuus helpottaa myös useiden näkymien kirjoittamista yhdelle mallille [9]. Näkymän passiivisuus mahdollistaa paremman vastuualueiden jaon ohjelmistoprojektin sisällä. Näkymiä voidaan kehittää erillään muusta ohjelmistosta eri työkaluilla kuin muu ohjelmisto. Tämä mahdollistaa grafiikkaan ja käyttäjäkokemukseen erikoistuneiden henkilöiden työskentelyn erillään ohjelmistoon erikoistuneista henkilöistä. [16][20]

MVVM-arkkitehtuuri mahdollistaa jo olemassa olevan mallin käytön ilman, että mallin lähdekoodiin tarvitsee koskea. Tämä onnistuu näkymämallin avulla, joka toimii sovittimena näkymän ja mallin välillä. Näkymämalli siis muokkaa mallin antaman tiedon näkymälle sopivaksi. Olemassa olevan lähdekoodin muokkaaminen voi olla riskialtista, koska muokkaajan on vaikea tietää, että miten muutokset vaikuttavat muihin ohjelmiston osiin. [9]

MVVM-arkkitehtuurissa korjataan suurin osa tarkkailija-suunnittelumalliin liittyvistä ongelmista. MVVM-arkkitehtuurissa tarkkailija-suunnittelumallia käytetään tietojen sitomisen yhteydessä ilmoittamaan ominaisuuden muutoksesta. Näkymään päivitetään siis vain tarvittava tieto, koska jokaisesta ominaisuuden muutoksesta tehdään erikseen ilmoitus [15]. Tämän lisäksi mallista haetaan vain tarvittava muuttunut tieto, koska näkymä ei hae tilaansa suoraan mallista vaan näkymän tila päivitetään näkymämallin kautta, joka on päivittänyt mallin ja siten tietää, että mitä tietoa pitää päivittää myös näkymään [16].

MVVM-arkkitehtuurin haittana on sen monimutkaisuus, joka vaikeuttaa pienten ohjelmistojen tekemistä. Tämän lisäksi näkymämallin suunnittelu etukäteen voi olla hankalaa. MVVM-arkkitehtuuri myös hankaloittaa koodin luettavuutta sen käyttämän tietojen sitomisen vuoksi. Tietojen sitominen on suhteellisen tehokas tapa välittää tietoa näkymämallin ja näkymän välillä, mutta sille on luontaista käyttää paljon muistia. [8]

7. JOHTOPÄÄTÖKSET

Ohjelmistotyyppien määrittely ohjelmistoarkkitehtuureille on tehty lähteiden ja teoriaosuudesta tehtyjen päätelmien perusteella. Tämä johtuu siitä, että ohjelmistoarkkitehtuureille tyypillisistä ohjelmistoista ei juurikaan löytynyt lähteitä. MVC- ja MVP-arkkitehtuureille löytyi lähde sopivista ohjelmistotyypeistä ja muille arkkitehtuureille ohjelmistotyyppi pääteltiin arkkitehtuurin toiminnan ja erityispiirteiden avulla.

Luvussa 3 esiteltiin Smart UI -arkkitehtuuri. Luvun 3 perusteella Smart UI -arkkitehtuuri on yksinkertainen arkkitehtuuri ja tästä syystä sen avulla on nopea tuottaa toimiva ohjelma. Smart UI -arkkitehtuurin käyttö vaikeuttaa ohjelmiston ylläpitoa, koska ohjelmiston logiikka sijaitsee erikseen jokaisessa näkymässä. Jos halutaan siis muuttaa ohjelmiston toimintaa, muutos on tehtävä jokaiseen näkymään erikseen. Tästä syystä ohjelmiston koon kasvaessa, sen ylläpito vaikeutuu huomattavasti. Edellä mainittujen piirteiden perusteella Smart UI-arkkitehtuurin avulla kannattaa toteuttaa pienet sovellukset ja ohjelmistot, joita ei muokata myöhemmin, kuten esimerkiksi prototyypit tai demot.

Smart UI -arkkitehtuuria paremman komponenttien vastuujaon mahdollistaa MVC-arkkitehtuuri. MVC-arkkitehtuuria ei kannata käyttää, mikäli ohjelmisto toteutetaan syötteiden lukemisen sisältävien käyttöliittymäkomponenttien avulla. Tämä johtuu luvussa 4.1 esitellyn ohjaimen roolista. MVC-arkkitehtuurin ohjain on suunniteltu hoitamaan syötteiden lukeminen ja toimintojen aktivoiminen mallista, jossa sijaitsee ohjelmiston logiikka. Tästä syystä jos käyttöliittymäkomponentit sisältävät syötteiden lukemisen, MVC-arkkitehtuurin ohjaimen rooli on vain välittää nämä syötteet mallille. MVC-arkkitehtuurin ja käyttöliittymäkomponenttien yhteensopimattomuus käy ilmi myös Dolphin Smalltalk MVP-arkkitehtuurin kehityksestä kertovassa dokumentissa [2]. Edellä mainittujen MVC-arkkitehtuurin piirteiden perusteella, MVC-arkkitehtuuri soveltuu siis hyvin ohjelmistoihin, joihin pitää itse toteuttaa syötteiden lukeminen. Tällaisia ohjelmistoja ovat esimerkiksi sulautetut järjestelmät ja web-sovellukset. Web-sovelluksissa ohjaimen rooli on vastaanottaa HTTP-pyynnöt.

Käyttöliittymäkomponenteista koostuvat ohjelmistot, siis useimmat työpöytäsovellukset kannattaa toteuttaa MVC-arkkitehtuurin sijaan MVP-arkkitehtuurilla, joka käy ilmi Bowerin ja McClashan kirjoittamasta dokumentista Twisting the Triad [2]. MVP-arkkitehtuurissa ajatellaan ohjaimen kuuluvan näkymään, joka on ominaista käyttöliittymäkomponenteille. MVC-arkkitehtuurin ohjaimen tilalle on luotu esittelijä,

joka toimii ohjainta korkeammalla tasolla. Esittelijän rooli vaihtelee MVP-arkkitehtuurin eri variaatioissa. MVP-arkkitehtuurin variaatio voidaan valita halutun esittelijän roolin ja näkymän passiivisuuden perusteella. Näkymän passiivisuus helpottaa ohjelmiston testattavuutta, mutta laajentaa esittelijän kokoa. MVP-arkkitehtuurin eri variaatioiden erityispiirteitä on esitelty luvussa 5.2.

MVVM-arkkitehtuuri muistuttaa passiivisen näkymän MVP-arkkitehtuuria, mutta ne eroavat esittelijän/näkymämallin ja näkymän yhteydessä. Passiivisen näkymän MVP-arkkitehtuurissa esittelijä on suoraan yhteydessä näkymään. Tämä tarkoittaa, että passiivisen näkymän MVP-arkkitehtuurissa näkymä on yksilöllinen esittelijälle. MVVM-arkkitehtuurissa näkymämallin ja näkymän yhteys on toisinpäin kuin pasiivisen näkymän MVP-arkkitehtuurissa, joten yhdelle näkymämallille voidaan luoda useita eri näkymiä. Toisaalta MVP-arkkitehtuurissa esittelijä hallitsee useita näkymiä, joten sille voi luoda monta näkymää, joita esittelijä hallitsee. Tämä kuitenkin kasvattaa esittelijän kokoa ja vastuualuetta, koska jokaisen näkymän logiikka tulee esittelijään.

Luvun 6 perusteella MVVM-arkitehtuuri on suunniteltu käytettäväksi ohjelmistoissa, joissa käytetään tietojen sitomista, mutta se soveltuu myös käytettäväksi käyttöliittymäkomponenteista koostuvaan ohjelmistoon. Käyttöliittymäkomponenteista koostuvassa ohjelmassa tietojen sitominen voidaan korvata tarkkailija-suunnittelumallilla. MVVM-arkkitehtuurissa näkymämallin tehtävä ohjelmistossa on toimia eri kielellä tai työkaluilla tehdyn käyttöliittymän rajapintana. Tästä syystä käyttöliittymäkomponenteista koostuvan ohjelmiston näkymämalli voi olla lähinnä välittäjä, joka tekee mallin ominaisuuksille vain tarvittavat muunnokset näkymää varten. Edellä mainittujen MVVM-arkkitehtuurin piirteiden perusteella MVVM-arkkitehtuuri soveltuu parhaiten käytettäväksi, kun ohjelmiston käyttöliittymä on tehty eri kielellä kuin käyttöliittymän logiikka. Tällainen ohjelmisto voi olla esimerkiksi Qt-viitekehyksen QML-kielellä tehty sovellus.

Oikean ohjelmistoarkkitehtuurin valinta riippuu paljolti käytetystä viitekehyksestä ja sen toteutustavasta. Ohjelmistoarkkitehtuuri kannattaakin yleensä valita käytetyn viitekehyksen perusteella ja mahdollisesti muokata valittu ohjelmistoarkkitehtuuri soveltumaan paremmin käytettyyn viitekehykseen. Dolphin Smalltalk MVP-arkkitehtuurin kehitys tapahtui soveltamalla ohjelmistoarkkitehtuuri käytettyyn viitekehykseen. Dolphin Smalltalk -kielen kehityksen alussa, Dolphin Smalltalk -kieli yritettiin toteuttaa MVC-arkkitehtuurilla, mutta Windows-ympäristössä oli käytössä valmiit käyttöliittymäkomponentit, jotka toteuttivat syötteiden lukemisen. Järkevän ohjainrakenteen luominen olisi vaatinut käyttöliittymäkomponenttien purkua osiin. Dolphin Smalltalk -kielen tapauksessa oli siis helpompi käyttää MVP-arkkitehtuuria, jossa ohjain ja näkymä oli jo valmiiksi yhdistetty. Taligent MVP-arkkitehtuuri ei kuitenkaan sellaisenaan soveltunut Dolphin Smalltalk -kieltä varten. Tämä korjattiin Dolphin Smalltalk -kielen tapauksessa muokkaamalla MVP-arkkitehtuurin esittelijän ja näkymän suhdetta käyttökohteeseen paremmin soveltuvaksi. [2]

8. YHTEENVETO

Aluksi työssä esiteltiin graafisten käyttöliittymien tehtävä, joka oli esittää tieto käyttäjälle ja vastaanottaa syöte. Graafisten käyttöliittymien ongelmat olivat samankaltaiset alustasta tai käyttöympäristöstä riippumatta, mutta graafisten käyttöliittymien toteutus vaihteli ohjelmiston tyypistä riippuen.

Lisäksi työssä käytiin läpi neljän suositun graafisen käyttöliittymän ohjelmistoarkkitehtuurin toiminta ja erityispiirteet ja lopuksi selvitettiin niille sopivia ohjelmistotyyppejä. Smart UI -arkkitehtuurin toteutus selvitettiin ensimmäiseksi sen yksinkertaisuuden perusteella. Smart UI -arkkitehtuuri ei sisältänyt lainkaan ohjelmiston osa-alueisiin jakoa ja soveltui erityisen hyvin yksinkertaisiin sovelluksiin.

Smart UI -arkkitehtuurin jälkeen esiteltiin MVC-arkkitehtuuri, jossa on parempi ohjelmiston osien vastuujako. Ohjelmisto jaettiin kolmeen osaan, joilla jokaisella oli oma tehtävänsä. Nämä osat olivat malli, näkymä ja ohjain. MVC-arkkitehtuuri on yksi ensimmäisistä ja kuuluisimmista graafisiin käyttöliittymiin liittyvistä ohjelmistoarkkitehtuureista. MVC-arkkitehtuuri soveltui sulautettuihin järjestelmiin tai web-sovelluksiin. Tämä johtui MVC-arkkitehtuurin ohjaimen roolista.

MVP-arkkitehtuurin toiminta selitettiin MVC-arkkitehtuurin jälkeen. MVP-arkkitehtuurin ideana oli yhdistää MVC-arkkitehtuurin ohjain ja näkymä ja luoda ohjaimen tilalle esittelijä, joka toimii korkeammalla tasolla kuin ohjain. MVP-arkkitehtuuri jaettiin kolmeen eri variaatioon esittelijän vastuualueen perusteella. Taligent MVP-arkkitehtuurissa esittelijä loi käyttöliittymäkomponentit, mutta ei ollut tämän jälkeen suoraan yhteydessä näkymään. Dolphin Smalltalk MVP-arkkitehtuurissa esittelijän annettiin hoitaa osa näkymän logiikasta ja passiivisen näkymän MVP-arkkitehtuurissa kaikki näkymän toiminnallisuus siirrettiin esittelijään. MVP-arkkitehtuuri helpotti graafisten käyttöliittymien testaamista ja soveltui käytettäväksi käyttöliittymäkomponentteja käyttävässä sovelluksessa.

Passiivisen näkymän MVP-arkkitehtuurin kanssa samankaltainen MVVM-arkkitehtuuri oli viimeinen esiteltävä ohjelmistoarkkitehtuuri. MVVM-arkkitehtuurin ideana oli jakaa ohjelmistoprojekti kahteen osaan. Nämä osat olivat käyttöliittymän ulkonäkö ja toiminallisuus. Käyttöliittymän ulkonäkö tehtäisiin eri työkaluilla tai kielellä kuin toiminallisuus ja nämä piti yhdistää jollain tavalla. Ratkaisuksi tähän ongelmaan esiteltiin näkymämalli, joka toimi sovittimena mallin ja näkymän välissä. Näkymä yhdistettiin näkymämalliin tietojen sitomisen avulla.

Työssä esiteltyjen ohjelmistoarkkitehtuurien läpi käyminen onnistui kattavasti. Työssä ohjelmistoarkkitehtuureista saa kuvan niiden toiminnasta ja jokaiselle ohjelmistoarkkitehtuurille löytyi paljon erityispiirteitä. Ohjelmistoarkkitehtuurit eroavat sopivasti toisistaan, jotta saadaan katettua muutama yleinen ohjelmistotyyppi jokaiselle arkkitehtuurille. Ongelmaksi muodostui ohjelmistotyyppien määrittely. Ohjelmistoarkkitehtuurien soveltuvuudesta tiettyyn ohjelmistotyyppiin ei löytynyt juurikaan tietoa lähteistä. Työn toiminnalliseksi osuudeksi muodostui siis sopivien ohjelmistotyyppien määritteleminen ohjelmistoarkkitehtuureille niiden toiminnallisuuden ja erityispiirteiden perusteella.

Työhön valitut ohjelmistoarkkitehtuurit kattavat hyvin graafisten käyttöliittymien ohjelmistoarkkitehtuurit. Työn edetessä vastaan tuli muitakin ohjelmistoarkkitehtuureja, mutta yleensä ne muistuttivat työssä esiteltäviä arkkitehtuureja. Tästä syystä niitä ei ole esitelty tässä työssä. Muita vastaan tulleita ohjelmistoarkkitehtuureja olivat esittelijä malli – (engl. *Presentation model*, lyh. PM), näkymä–abstraktio–ohjain- (engl. *presentation-abstraction-control*, lyh. PAC) ja malli–näkymä- (engl. *model-view*, lyh. MV) arkkitehtuurit. Edellä mainituista ohjelmistoarkkitehtuureista PAC-arkkitehtuuri muistuttaa paljon MVC-arkkitehtuuria ja PM-arkkitehtuuri passiivisen näkymän MVP-arkkitehtuuria tai MVVM-arkkitehtuuria. MV-arkkitehtuuri on Smart UI – ja MVC-arkkitehtuurien välimaastossa.

LÄHTEET

[1] J. Blanchette, M. Summerfield, C++ GUI Programmin with Qt 4 Second Edition, Trolltech Press, 2008

[2] A. Bower, B. McGlashan, Twisting the Triad, 2000, pdf. Saatavissa (viitattu: 19.10.2017): http://www.object-arts.com/downloads/papers/TwistingTheTriad.PDF

[3] M. Ever, SmartUI Architecture Pattern, 2015, verkkosivu. Saatavissa (viitattu 03.10.2017): http://www.markewer.com/2015/10/21/smartui-architecture-pattern/

[4] M. Fowler, GUI Architectures, 2006, verkkosivu. Saatavissa (viitattu 15.09.2017): https://martinfowler.com/eaaDev/uiArchs.html

[5] M. Fowler, Passive View, 2006, verkkosivu. Saatavissa (viitattu: 25.10.2017): https://martinfowler.com/eaaDev/PassiveScreen.html

[6] R. Frese, V. Sauter, Project success and failure: what is success, what is failure, and how can you improve your odds for success?, 2003, verkkosivu. Saatavissa (viitattu 19.10.2017): http://www.umsl.edu/~sauterv/analysis/6840_f03_papers/frese/

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Olio-ohjelmointi Suunnittelumallit Design Patterns, IT Press, 2001.

[8] J. Gossman, Advantages and disadvantages of M-V-V-M, 2006, verkkosivu. Saatavissa (viitattu 05.11.2017): https://blogs.msdn.microsoft.com/johngossman/2006/03/04/advantages-and-disadvantages-of-m-v-vm/

[9] J. Gossman, Introduction to Model/View/ViewModel pattern for building WPF apps, 2005, verkkosivu. Saatavissa (viitattu 03.11.2017): https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/

[10] Systems and software engineering – Architecture description ISO/IEC/IEEE 42010-2011, 2011.

[11] M. Jaggavarapu, Presentation Patterns,2012, verkkosivut. Saatavissa (viitattu 15.09.2017): https://manojjaggavarapu.wordpress.com/2012/05/02/presentation-patterns-mvc-mvp-pm-mvvm/

[12] A. Karagkasidis, Developing GUI Applications: Architectural Patterns Revisited, 2008, pdf. Saatavissa (viitattu 03.10.2017): http://ceur-ws.org/Vol-610/paper11.pdf

[13] K. Koskimies, T. Mikkonen, Ohjelmistoarkkitehtuurit, Talentum, 2005, 16 s. Pdf. Saatavissa (viitattu 28.09.2017): http://www.cs.tut.fi/~ohar/OHAR2012_13-KIRJA-KoskimiesMikkonen.pdf

[14] The Linux Information Project, GUI Definition, 2004, verkkosivu. Saatavissa (viitattu 03.10.2017): http://www.linfo.org/gui.html

[15] Microsoft, Data Binding Overview, 2017, verkkosivu. Saatavissa (viitattu 5.11.2017): https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/data-binding-overview

[16] Microsoft, The MVVM Pattern, 2012, verkkosivu. Saatavissa (viitattu 03.11.2017): https://msdn.microsoft.com/en-us/library/hh848246.aspx

[17] M. Potel, MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java, 1996, pdf. Saatavissa (viitattu 19.10.2017): http://www.wildcrest.com/Potel/Portfolio/mvp.pdf

[18] M. Reddy, API Design for C++, Elsevier/Morgan Kaufmann, 2011.

[19] S. Sanderson, Pro ASP.NET MVC 2 Framework Second Edition, Apress, 2010.

[20] J. Smith, Patterns – WPF Apps With The Model-View-ViewModel Design Pattern, 2009, verkkosivu. Saatavissa (viitattu 03.11.2017): https://msdn.microsoft.com/en-us/magazine/dd419663.aspx