Kerroksittainen ja malli-näkymä-ohjain -ohjelmistoarkkitehtuurityyli

Kahden perinteisen arkkitehtuurityylin esittely ja vertailu

Ville Rapa

16.6.2008

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

Tiivistelmä

Ohjelmistoarkkitehtuurilla tarkoitetaan ohjelmiston rakennetta ilman, että kiinnitetään huomiota toteutuksen yksityiskohtiin, kuten algoritmeihin ja tietorakenteisiin. Arkkitehtuuritasolla komponentteja ja niiden välisiä vuorovaikutussuhteita käsitellään mustan laatikon kaltaisesti. Suuria ohjelmistoja kehitettäessä ohjelmistoarkkitehtuuri on tärkeä tekijä. Jokainen kehityksessä mukana oleva taho voi hyödyntää arkkitehtuuria, joka mahdollistaa ohjelmiston analysoinnin hyvin aikaisessa vaiheessa kehitystyötä. Arkkitehtuurityylien avulla arkkitehtuureja voidaan kierrättää samankaltaisia ohjelmistoja kehitettäessä. Arkkitehtuurityylit auttavat löytämään kehitettävälle ohjelmistolle parhaiten soveltuvan arkkitehtuuriratkaisun.

Tässä tutkielmassa selvitetään, mitä tarkoitetaan ohjelmistoarkkitehtuurilla ja ohjelmistoarkkitehtuurityylillä. Tutkielmassa perehdytään syvemmin sekä kerroksittaiseen että malli-näkymä-ohjain -ohjelmistoarkkitehtuurityyliin. Tutkielmaan sisältyy vertaileva tutkimus, jossa pyritään tuomaan esille käsiteltävien ohjelmistoarkkitehtuurityylien eroja niiden arkkitehtuurisia ominaisuuksia vertaamalla. Käsiteltävien arkkitehtuurityylien ymmärtämisen sekä vertailun tueksi on tutkielmassa toteutettu yksinkertainen vaalijärjestelmä sekä kerroksittaisella että malli-näkymä-ohjain -ohjelmistoarkkitehtuurityylillä.

ACM-luokat (ACM Computing Classification System, 1998 version): D.2.11, K.6.3

Avainsanat: Ohjelmistoarkkitehtuuri, kerroksittainen tyyli, malli-näkymä-ohjain

Esipuhe

Haluaisin tässä yhteydessä lämpimästi kiittää vanhempiani kaikesta siitä avusta ja kannustuksesta, jota olen heiltä koko opiskeluaikani saanut. Tämän tutkielman valmistumisesta kuuluu suuri kiitos rakkaalle puolisolleni Sinille, joka jaksoi auttaa ja patistaa minua, vaikka se ei aina kovin helppoa ollutkaan. Kiitos kuuluu myös tyttärelleni Innalle hänen positiivista energiaa säteilevästä vaikutuksestaan.

Sisältö

1 Johdanto

Hyvä ohjelmisto on laadukas ja loppukäyttäjän tarpeet täyttävä. Jos tähän lopputulokseen päästään suunnitellussa ajassa ylittämättä käytettyjä resursseja, voidaan itse ohjelmiston ja sen tuottamistavan olettaa sopivan liike-elämään. Laadukasta ohjelmistoa kehitettäessä mallintaminen on tärkeä toiminto lopputuloksen kannalta. Ohjelmiston arkkitehtuuri-malli mahdollistaa korkean tason mallintamisen jo aikaisessa vaiheessa ohjelmistotuotantoa, mikä puolestaan mahdollistaa ohjelmiston tarkastelun ja analysoinnin hyvissä ajoin. Tällä tavoin voidaan vähentää tai jopa merkittävästi välttää ohjelmiston kehitykseen liittyviä riskejä (Garlan & Shaw, 1998).

Tässä tutkielmassa käsitellään kerroksittaista ja malli-näkymä-ohjain -ohjelmistoarkkitehtuureja. Kerroksittainen arkkitehtuuri on yksi käytetymmistä arkkitehtuureista, vaikka se on usein huonosti määritelty ja jopa väärin ymmärretty. Hyvin määritelty ja todellinen kerroksittainen arkkitehtuurityyli tarjoaa hyvät ja usein ohjelmiston vaatimuksiin liittyvät ominaisuudet, kuten muunneltavuuden ja siirrettävyyden. Malli-näkymä-ohjain -arkkitehtuuri on myös hyvin laajasti käytetty arkkitehtuuri. Ohjelmistoarkkitehtuurina malli-näkymä-ohjain on kerroksittaisen arkkitehtuurin tavoin hyvin vanha ja sen pääasiallinen käyttö on graafisen käyttöliittymän sisältävissä järjestelmissä. Malli-näkymä-ohjain -arkkitehtuuri on tehnyt uuden tulemisen johtuen sen hyvästä soveltuvuudesta web-käyttöliittymän sisältäviin järjestelmiin.

Tutkielma alkaa johdatuksella ohjelmistoarkkitehtuuriin ja erilaisiin arkkitehtuurityyleihin. Pääpaino on luvuissa 3 ja 4, jotka jakautuvat alilukuihin, joista jokainen käsittelee jotakin tärkeää ohjelmistoarkkitehtuuriin liittyvää ominaisuutta. Luvussa 5 tutkitaan käsiteltyjen arkkitehtuurityylien eroja ja niiden vaikutuksia arkkitehtuurityyliä valittaessa. Lopuksi luvussa 6 on yhteenveto tutkielman keskeisistä asioista.

2 Ohjelmistoarkkitehtuuri

Tässä luvussa pohditaan yleisellä tasolla, mitä tarkoitetaan ohjelmistoarkkitehtuurilla. Ensimmäiseksi tutustutaan pintapuolisesti ohjelmoinnin ja ohjelmistokehityksen historiaan ohjelmistoarkkitehtuurin näkökulmasta, minkä jälkeen määritellään tarkemmin käsite ohjelmistoarkkitehtuuri sekä siihen liittyvä terminologia. Tämän lisäksi selvitetään miten ohjelmistoarkkitehtuuri ja ohjelmistoarkkitehti kytkeytyvät ohjelmistotuotantoon. Lopuksi siirrytään lähemmäksi tutkielman aihetta ja tutustutaan arkkitehtuuritason malleihin eli arkkitehtuurityyleihin.

2.1 Yleistä

Garlanin & Shaw'n (1994) mukaan ohjelmointikielten ja ohjelmistokehitystyökalujen kehityksessä on havaittavissa niiden abstraktiotason säännöllinen kasvaminen. 1950-luvulla ohjelmatoteuttaja kirjoitti itse konekieliset ohjelmat sekä siirsi ohjelman tarvitsemat tiedot suoraan tietokoneen muistiin. Kuitenkin 1950-luvun lopulla keksittiin, että osa tästä prosessista voitaisiin automatisoida ja prosessia voitaisiin helpottaa. Kehitettiin symboleja ymmärtävä konekieli (assembler), josta rakenteiden kehityttyä päästiinkin ensimmäisiin korkean tason ohjelmointikieliin, kuten Fortraniin. Korkean tason ohjelmointikielet mahdollistivat monimutkaisempien ja laajempien ohjelmistojen kehityksen, mikä johti abstraktien tietorakenteiden kehittymiseen. 1970-luvulla tiettyä tarkoitusta varten tehdyt ohjelmanosat opittiin jakamaan omiin moduuleihinsa, mikä selkeytti huomattavasti ohjelmistojen rakennetta. Edellä kuvattu ohjelmistokehityksen abstraktiotason asteittainen kohoaminen on johdettavissa aina ohjelmistoarkkitehtuuriin asti, ja siinä missä 1970-luvun hyvät ohjelmoijat näkivät ohjelmistojen rakenteissa modulaarisia piirteitä ja käyttivät niitä hyväkseen ohjelmistoja kehittäessään, tämän päivän ohjelmistokehittäjät pystyvät hahmottamaan ohjelmiston kokonaisrakenteen: ohjelmistoarkkitehtuurin.

Ohjelmistoarkkitehtuurilla tarkoitetaan ohjelmiston rakennetta kiinnittämättä huomiota toteutuksen yksityiskohtiin, kuten algoritmeihin ja tietorakenteisiin. Arkkitehtuuritasolla käsitellään komponentteja ja niiden välisiä vuorovaikutus-suhteita mustan laatikon kaltaisesti.

Järjestelmien kasvaessa myös ohjelmistoarkkitehtuurista on tullut tärkeä osa ohjelmis-

totuotantoa (Garlan & Shaw, 1998). Mitä suuremmista järjestelmistä on kyse, sitä tärkeämpää roolia järjestelmän arkkitehtuuri näyttelee. Yleisesti hyväksyttyä ja yhdenmukaista määritystä ohjelmistoarkkitehtuurille ei ole olemassa, mutta Bassin *& al.*:n (1998) määritelmä on ainakin yksi lainatuimmista: Ohjelman tai ohjelmiston arkkitehtuuri on koko sen systeemin rakenne tai rakenteet, joka muodostuu ohjelmistokomponenteista ja niiden ulkoisista ominaisuuksista, sekä komponenttien välisistä suhteista. Tässä tapauksessa ulkoisilla ominaisuuksilla viitataan oletuksiin siitä, mitä muut komponentit voivat tehdä toisten komponenttien suhteen. Ulkoisia ominaisuuksia voisivat olla esimerkiksi tarjotut palvelut, virheiden hallinta, jaetut resurssit, jne.

Yhteinen piirre ohjelmistoarkkitehtuurin määritelmille on kuitenkin ohjelmiston jakaminen pienempiin osiin ja koko ohjelmiston kuvaaminen syntyneiden osien avulla sekä niiden välisillä yhteyksillä. On myös huomioitava, että ohjelmisto voi olla myös osa suurempaa ohjelmistoa, jolloin pienempi ohjelmisto toimii komponentin roolissa osana suurempaa ohjelmistoa. Arkkitehtuurin avulla pystytään siis hahmottamaan hyvin monimutkaisia järjestelmiä jakamalla ne pienempiin, helpommin ymmärrettäviin osiin.

Määritelmän mukaisesti arkkitehtuurin yksi keskeisimpiä käsitteitä on komponentti, mitä ei tule sekoittaa ohjelmistokomponentti-käsitteeseen. Arkkitehtuuri sisältää tietoa siitä, kuinka komponentit liittyvät toisiinsa ja kommunikoivat keskenään. Toisaalta tieto, joka ei liity komponenttien väliseen kommunikointiin tai niiden suhteisiin, jätetään arkkitehtuurikuvauksesta pois (The Software Engineering Institute, 2004). Komponenteissa kiinnostaa vain se kuinka ne käyttäytyvät, kuinka niitä käytetään, ja kuinka ne liittyvät ja kommunikoivat toisten komponenttien kanssa.

Määritelmästä seuraa myös se, että jokaisella ohjelmistolla on arkkitehtuuri, koska jokainen järjestelmä voidaan jakaa komponentteihin ja komponenttien väliset suhteet voidaan jollakin tavalla ilmaista. Yksinkertaisimmillaan ohjelmisto koostuu yhdestä komponentista, mikä kuitenkaan ei ole arkkitehtuurisesti kovinkaan mielenkiintoista (The Software Engineering Institute, 2004). Jokaisella komponentilla on myös sisäinen arkkitehtuuri, joka voi erota huomattavasti koko järjestelmän arkkitehtuurista. Myös osa komponentin toiminnallisuudesta voi olla ohjelmistoarkkitehtuuria, sikäli kuin toiminnallisuus vaikuttaa muihin komponentteihin tai on tärkeää muiden komponenttien kannalta.

Ohjelmistoarkkitehtuuri yhdistää ohjelmistolle kohdistuvat vaatimukset ja rakennetta-

van ohjelmiston komponentit sekä järjellistää ja edesauttaa suunnittelupäätöksiä. Arkkitehtuuritasolla ohjelmiston ominaisuudet ovat seuraavat: kapasiteetti, suorituskyky, johdonmukaisuus sekä komponenttien yhteensopivuus.

Monesti ohjelmistoarkkitehtuurin suunnittelu yhdistetään suoraan ohjelmiston suunnitteluun. Tietenkin arkkitehtuurisuunnittelu on myös ohjelmiston suunnittelua, mutta ohjelmiston suunnittelusta puhuttaessa tarkoitetaan ohjelman toteutuksen yksityiskohtien suunnittelua, kun taas puhuttaessa arkkitehtuurisuunnittelusta tarkoitetaan ohjelmiston keskeisten rakenteiden suunnittelua korkealla abstraktiotasolla.

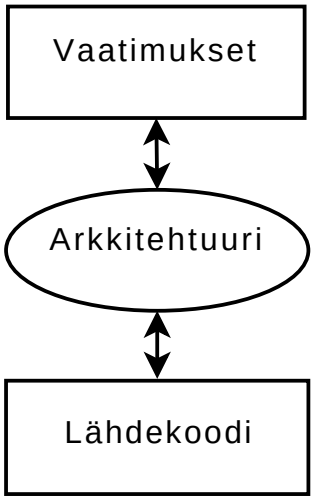
2.2 Ohjelmistoarkkitehtuurin merkitys

Luomalla abstraktiomalli ohjelmistosta tuodaan esille joitakin järjestelmän ominaisuuksia ja samalla piilotetaan toisia. Näkyville tulevat ominaisuudet ovat juuri sellaisia ominaisuuksia, joilla on suuri vaikutus koko ohjelmistoprojektin onnistumiselle.

Bass *& al.*:a (1998) esittävät ainakin kolme todella merkittävää syytä sille, miksi ohjelmistoarkkitehtuuri on tärkeä: tiedonvälitys, suunnittelun linjaus ja abstraktion siirrettävyys. Tiedonvälityksellä tarkoitetaan sitä, että arkkitehtuurilla voidaan esittää korkean tason abstraktiomalli järjestelmästä, mikä helpottaa muiden perehdyttämistä järjestelmään sekä lisää tietoutta järjestelmästä. Arkkitehtuurimalli mahdollistaa kaikkien osapuolten välisen kommunikoinnin rakennettavasta ohjelmistosta. Suunnittelun linjauksella luodaan sekä kiinnitetään pohja suunnittelulle, joka säilyy koko ohjelmiston elinkaaren ajan. Tähän liittyy myös läheisesti analysointi. Koska arkkitehtuuri mahdollistaa järjestelmän analysoinnin kehityksen aikaisessa vaiheessa, tuo se mukanaan mahdollisuuden pohtia erilaisia rakenteellisia perusratkaisuja vaiheessa, jossa isojen muutosten teko on vielä mahdollista. Samoin kuin on tärkeää kierrättää komponentteja, on myös arkkitehtuurien kierrättäminen tärkeää. Abstraktion siirrettävyydellä tarkoitetaan arkkitehtuurin kierrätettävyyttä, se mahdollistaa arkkitehtuurin uudelleen käytön samoja piirteitä vaativia ohjelmistoja toteutettaessa.

Edellä esitetyt arkkitehtuurin tuomat ominaisuudet vähentävät järjestelmän kehitykseen liittyviä riskejä, koska järjestelmää voidaan analysoida aikaisessa vaiheessa ja sitä suunniteltaessa voidaan käyttää valmiita hyväksi todettuja arkkitehtuurimalleja (Bass *& al.*, 1998). Arkkitehtuurimallista heijastuu jo varhaisessa vaiheessa suunnitteluratkaisut, joilla on paljon vaikutusta koko järjestelmän onnistumiseen.

Ohjelmistoarkkitehtuurista puhuttaessa ohjelman kehitys on hyvin korkealla abstraktiotasolla. Garlan (2000) nostaa kuitenkin vahvasti esille arkkitehtuurin roolin vaatimusten ja toteutuksen siltana kuvan 1 esittämällä tavalla.



Kuva 1: Ohjelmistoarkkitehtuuri siltana (Garlan, 2000).

Arkkitehtuurin merkitys järjestelmiä kehitettäessä on tiedetty jo vuosia, mutta se ei ole vielä lähelläkään sillä tasolla, jolla perinteinen arkkitehtuuri on. Talonrakennuksen ja ohjelmankehityksen historiat ovat toki hieman erimittaiset. Nuoruudella yleensä selitetäänkin, miksi ohjelmistoarkkitehtuurin tuomia etuja ei vielä osata täysin hyödyntää.

2.3 Ohjelmistoarkkitehdin rooli

Ohjelmiston kehityksessä kiinnitetään yhä enemmän huomiota sen arkkitehtuuriin, joten myös arkkitehtien rooli kasvaa ja heidän tarpeensa lisääntyy. Bredemeyerin ja Malanin (2000) mukaan ohjelmistoarkkitehti vastaa ohjelmiston arkkitehtuurisista ratkaisuista. Arkkitehti joutuu valitsemaan tai suunnittelemaan vaatimuksiin parhaiten sopivan arkkitehtuurin sekä dokumentoimaan komponentit, rajapinnat ja niiden välisen vuorovaikutuksen. Arkkitehti tarvitsee hyvän tietämyksen suunniteltavan ohjelmiston käyttötarkoituksesta ja siihen liittyvästä toimialasta sekä riittävät tiedot käytettävissä olevista teknologioista ja kehitysprosesseista.

2.4 Arkkitehtuurityylit

Ohjelmistoarkkitehtuurityylit ovat jossain määrin analogisia rakennusten arkkitehtuurityylien kanssa. Kummatkin sisältävät arkkitehtuurin tärkeimmät ominaisuudet ja säännöt siitä, kuinka ominaisuuksia käytetään arkkitehtuurinen yhtenäisyys säilyttäen.

Shaw (1996) on tutkinut ohjelmistojen kuvauksia ja löytänyt niistä yhdenmukaisia malleja. Näistä malleista Shaw on luonut erilaisia arkkitehtuurityylejä, joista tunnetuimmat ovat:

- putki ja suodatin
- tietovirta
- keskustelevat prosessit
- tapahtuma (implicit invocation)
- tietovarasto
- tulkki
- pääohjelma-aliohjelma
- kerroksittaisuus.

Arkkitehtuurityylit ovat korkean tason ratkaisuja arkkitehtuurille. Garlanin ja Shaw'n (1998) mukaan arkkitehtuurityyli (architectural styles, architectural patterns ja architechtural idiom) määrittelee komponenttien termistön, *liittimien* tyypit ja rajoitteet komponenttien ja liittimien yhdistelemiselle. Liittimellä tarkoitetaan tapaa, millä komponentit yhdistetään toisiinsa. Tyypillisiä liittimiä ovat esimerkiksi aliohjelmakutsu, putki ja tapahtuma (event).

Komponenttien termistöllä tarkoitetaan keskeisten komponenttien nimiä, kuten putki, suodatin, kerros, asiakas, palvelin ja tietokanta. Rajoitteet komponenttien ja liittimien yhdistämiselle tarkoittavat järjestelmään liittyviä topologisia rajoitteita. Esimerkiksi asiakas-palvelin arkkitehtuurityylissä voi asiakkaan ja palvelimen suhde olla yksi-moneen. Palvelin voi siis palvella monia eri asiakkaita samanaikaisesti, mutta asiakas ei voi samanaikaisesti pyytää palveluja usealta eri palvelimelta.

Arkkitehtuurityylejä voidaan tarkastella vastaamalla seuraaviin arkkitehtuurityyliä koskeviin kysymyksiin (Garlan & Shaw, 1998):

- Mikä on käytetty sanasto (komponentit ja liittimet)?
- Mitkä ovat sallitut rakenteet?
- Mikä on laskentamallin perusta?
- Mikä on tyylin muuttumaton ydin?
- Mikä on tyylin yleisin käyttökohde?
- Mitkä ovat tyylin edut ja haitat?
- Millaiselle erityisaloille tyyli sopii parhaiten?

Arkkitehtuurityylit toimivat runkona suunniteltaessa tietylle tyylille sopivia ohjelmistoja. Arkkitehtuurityylit auttavat ohjelmistoon perehtyvää myös ymmärtämään ohjelmiston rakennetta korkealla tasolla, jos perehtyjällä on aikaisempaa tietämystä samasta tyylistä ja hän pystyy yhdistämään sen johonkin tyyliperheeseen kuuluvaan ohjelmistoon. Ohjelmistot tosin ovat harvoin puhtaasti yhden tyylin mukaisia.

Bassin *& al*.:in (1998) mukaan tyylit auttavat löytämään soveltuvimman arkkitehtuuriratkaisun kehitettävälle ohjelmistolle, ja niiden avulla muiden on helpompi ymmärtää järjestelmän rakenne, jos käytettävä tyyli on entuudestaan tuttu.

3 Kerroksittainen ohjelmistoarkkitehtuurityyli

Edellä on käsitelty pintapuolisesti ohjelmistoarkkitehtuuria ja arkkitehtuurityylejä. Tässä luvussa esitellään yksityiskohtaisesti kerroksittainen ohjelmistoarkkitehtuurityyli: määritellään mitä sillä tarkoitetaan, miten sitä käytetään, mitä etuja sillä saavutetaan ja millaisia rajoitteita siihen liittyy. Luvussa esitetään myös tunnetuimpia kerroksittaiseen arkkitehtuurityyliin pohjautuvia ohjelmistoja sekä standardeja. Tässä luvussa hahmotellaan yksinkertainen kerroksittaiseen arkkitehtuurityyliin perustuva vaalijärjestelmä. Vaalijärjestelmä pitää kirjaa puolueiden äänimääristä ja havainnostaa vaalien äänijakaumaa graafisesti.

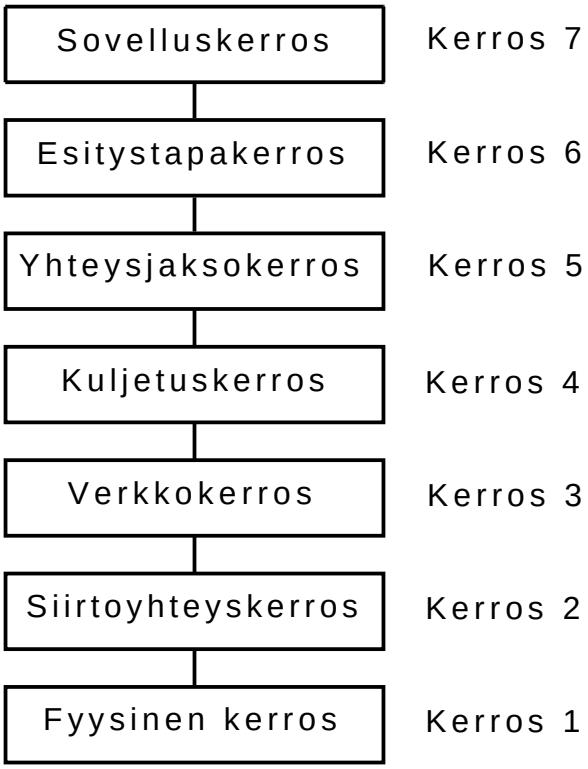
3.1 Johdatus kerroksittaiseen arkkitehtuurityyliin

Kerroksittaisen arkkitehtuurin merkitystä ohjelmistotuotannossa ei voi missään tapauksessa vähätellä, ovathan ensimmäiset viitteet sen käytöstä jo yli kolmekymmentä vuotta vanhoja. Esimerkiksi Dijkstran (1968) mukaan hierarkkinen systeemi osoittautui keskeiseksi tekijäksi THE-käyttöjärjestelmän kehityksessä. Käyttöjärjestelmien lisäksi kerroksittainen arkkitehtuuri on tärkeässä asemassa verkkoprotokollien (network protocol, communication protocol) yhteydessä. ISO:n (The International standardization Organization) seitsemänkerroksinen OSI-malli (Open System Interconnection) on hyvin tunnettu esimerkki kerroksittaisesta rakenteesta (Zimmermann, 1980). Kuva 2 esittää ISO:n OSI-mallia, ja siihen on sisällytetty myös kerrosten nimet, jotka kuvaavat kerrosten toimintaa. Myös tietokantajärjestelmät ovat sellainen mainittava sovellusalue, jossa kerroksittaista arkkitehtuuria on totuttu näkemään (Garlan & Shaw, 1994).

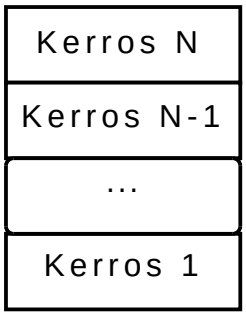
3.2 Rakenne ja terminologia

Kerroksittaisessa arkkitehtuurityylissä komponentit on ryhmitelty päällekkäisiksi kerroksiksi, mikä on havainnollistettu kuvassa 3. Koska kerros muodostuu yhdestä, ja vain yhdestä komponentista, käytetään komponentista nimitystä *kerros* (layer). Kerrokset jakavat järjestelmän loogisesti yhtenäisiin, mutta eri käsitetasolla oleviin osiin.

Bachmann *& al.* (2000) määrittelee kerroksittaisen arkkitehtuurityylin hieman poikkeuksellisella tavalla, mutta kuitenkin hyvin ymmärrettävästi ja hieman käytännön-



Kuva 2: ISO:n OSI-malli (Zimmermann, 1980).



Kuva 3: Kerroksittainen arkkitehtuuri.

läheisemmin kuin esimerkiksi Garlan ja Shaw (1994). Bachmannin *& al*.:in (2000) mukaan kerroksittainen arkkitehtuurityyli, kuten kaikki arkkitehtuurityylit, kuvaavat ohjelmiston jakamista pienempiin osiin. Kerroksittaisen tyylin tapauksessa osia kutsutaan kerroksiksi ja jokainen kerros on itsenäinen *virtuaalikone*. Kerroksittaisen jaottelun päämääränä on jakaa ohjelmisto kerroksiksi virtuaalikoneita, jotka ovat tarpeeksi suppeita ymmärrettävyyden kannalta, mutta kuitenkin niin yhtenäisiä, että muutokset ohjelmistoon vaikuttavat vain yhteen kerrokseen.

Virtuaalikone on joukko ohjelmia tai ohjelman osia, jotka yhdessä tarjoavat yhtenäisen joukon palveluja, joita muut ohjelmat voivat käyttää tietämättä kuinka palvelut on toteutettu (Bachmann *& al*., 2000). Virtuaalikoneen tarjoamien palvelujen joukon tulee olla yhtenäinen jonkin kriteerin mukaan. Palvelut voivat olla esimerkiksi jonkin ohjelmiston tarpeisiin liittyviä graafisia toimintoja. Kuitenkin komponentti, jolla on yleinen rajapinta ja sen myötä joukko määriteltyjä palveluja, ei välttämättä muodosta virtuaalikonetta. Palvelut voivat liittyä vaikkapa johonkin erityisalaan, kuten matematiikkaan tai tiekantapalveluihin.

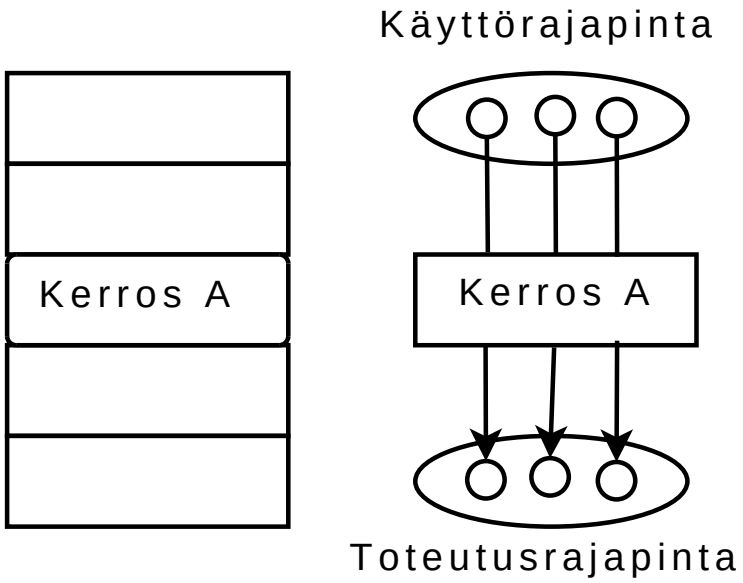
Sen lisäksi, että kerroksittaisessa tyylissä ohjelmisto jaetaan eri kerroksiin ja kerrokset tarjoavat toisilleen palveluja, sen oleellisiin ominaisuuksiin kuuluvat myös säännöt kerrosten väliselle vuorovaikutukselle. Kerrosten välinen vuorovaikutus on siis tarkkaan määrätty ja puhtaassa (strict) kerroksittaisessa arkkitehtuurissa kerroksen A ja kerroksen A+1 välillä vallitsee tarkkaan määrätty suhde. Kerros A+1 on kerroksen A yläpuolella, mikä tarkoittaa seuraavia asioita:

1. Kerroksen A+1 toteutuksen sallitaan käyttävän vain kerroksen A tarjoamia palveluja
2. Kerroksen A palvelut on sallittu vain kerrokselle A+1

Kerroksittaisessa tyylissä voidaan käyttää myös väljää (relax) suhdetta kerrosten välillä. Tämä tarkoittaa sitä, että kerros A voi käyttää kaikkien sen alapuolisten kerrosten palveluja. Kuitenkaan alemman kerroksen ei missään tapauksessa sallita käyttävän ylemmän kerroksen palveluja; sellaisessa tapauksessa ei voida enää puhu kerroksittaisesta tyylistä. Jos yläpuolisten palvelujen käyttö sallitaan, niin kerroksittaisen tyylin tuomat edut menetetään. Yhteenvetona voidaan sanoa, että kerroksittaisessa tyylissä komponenttia kutsutaan kerrokseksi ja kerrosten välinen kommunikointi tapahtuu korkeammalta tasolta matalammalle.

3.3 Kerroksen rajapinnat

Boochin *& al.*:in (1999) mukaan rajapinta toimii linkkinä toiminnallisuuden määritelmän ja toteutuksen välillä. Rajapinta on kokoelma operaatioita, joilla määritellään komponentin tarjoamat palvelut. Kerroksittaisen arkkitehtuurityylin komponentin rajapinnalla, tai pikemminkin rajapinnoilla, tarkoitetaan seuraavaa. Tarkastellaan kerroksittaisen arkkitehtuurin yhtä kerrosta, joka sijaitsee ylimmän kerroksen ja alimman kerroksen välissä, ja irrotetaan se arkkitehtuurista. Tämän kerroksen ylemmälle kerrokselle tarjoamien palvelujen rajapintaa kutsutaan *käyttörajapinnaksi*. Vastaavasti kerroksen käyttämät alemman tason palvelut muodostavat *toteutusrajapinnan*. Kerroksen käyttörajapinta on siis sama kuin seuraavaksi ylemmän kerroksen toteutusrajapinta. Alimman kerroksen toteutusrajapinta ja ylimmän kerroksen käyttörajapinta yhdistävät järjestelmän ulkopuoliseen maailmaan. Kuvassa 4 on havainnollistettu irrotetun kerroksen käyttö- ja toteutusrajapintoja.



Kuva 4: Kerroksen rajapinnat.

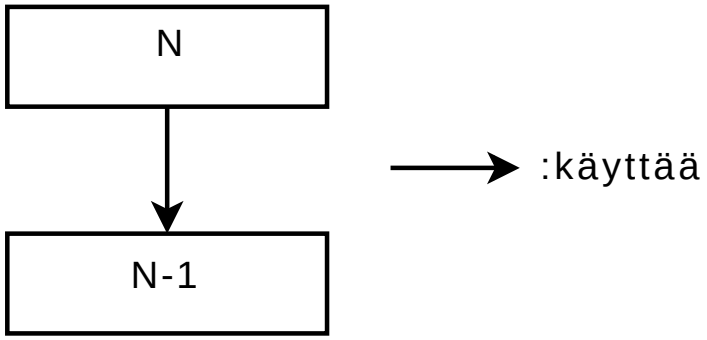
Kerroksittaisen ohjelmiston kehityksen kannalta mustalaatikko-menetelmä on parempi muihin menetelmiin verrattuna (Bachmann *& al.*, 2002). Kerroksittainen arkkitehtuurityyli tukee erinomaisesti mustalaatikkomaista kehitystä. Ollakseen mustalaatikko kerros tarvitsee tuekseen hyvin määritellyn ja dokumentoidun rajapinnan. Rajapintamääritykset ovat hyvin tärkeitä, koska rajapinta liittää kerroksen muihin kerroksiin, ja muutokset rajapinnassa voivat vaikuttaa myös muihin kerroksiin. Alkuvaiheessa hyvin

määritellyt ja kiinnitetyt rajapinnat edesauttavat komponenttien erillistä kehittämistä ja uudelleenkäyttöä. Rajapinta on myös avainasemassa, jos ohjelmistosta halutaan vaihtaa jonkin kerroksen toteutus toiseen toteutukseen.

3.4 Kontrollin kulku

Yksinkertaisimmillaan kerroksittaisessa arkkitehtuurityylissä kontrolli kulkee järjestyksessä kaikkien kerrosten läpi alkaen kerroksesta N ja päättyen kerrokseen 1. Tällainen tapaus esiintyy kuitenkin vain puhtaassa ja yksinkertaisimmassa muodossa kerroksittaisesta arkkitehtuurista. Buschmannin *& al.*:in (2001) mukaan kerroksittaisen arkkitehtuurityylin dynaaminen käyttäytyminen ja kontrollin kulku kerrosten välillä voidaan jakaa viiteen erilaiseen tapaukseen seuraavaksi esitettävällä tavalla.

Ensimmäinen tapaus on samankaltainen kuin edellisessä kappaleessa kuvattu, mutta hieman tarkennettuna. Kerros N pyytää kerroksen N-1 palveluja. Kerros N-1 ei selviydy pyynnöstä itsenäisesti ja se pyytää kerroksen N-2 palveluja. Näin kutsu siirtyy kerrokselta toiselle saavuttaen lopulta kerroksen 1. Kerros 1 lähettää tarvittaessa vastauksen kerrokselle 2, joka puolestaan jatkaa sen kerrokselle 3, ja näin vastaus siirtyy kerrokselta toiselle, kunnes saavutetaan kerros N. Tällä tavalla kontrolli siirtyy kerrokselta toiselle, ja kerroksen N tehtävä tulee suoritettua. Kuvassa 5 on havainnollistettu tämän tapaista kontrollinkulkua kerroksittaisessa arkkitehtuurissa; kerros N käyttää siinä kerroksen N-1 tarjoamia palveluja.



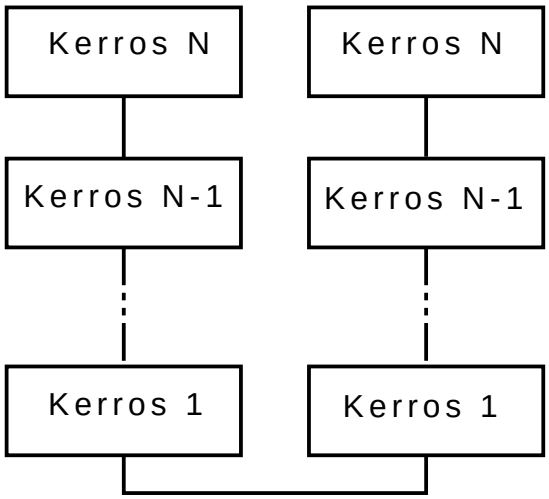
Kuva 5: Kontrollin siirtyminen alemmalle kerrokselle.

Edellä kuvattua tapaa voidaan kutsua ylhäältä-alas-kommunikoinniksi, ja luonteenomaista sille on, että kerroksen N palvelupyyntö jakautuu kerroksessa N-1 useaksi palvelupyynnöksi kerrokselle N-2. Tämä johtuu siitä, että kerros N on korkeammalla

abstraktiotasolla kuin kerros N-1, ja kerros N-1 on puolestaan korkeammalla abstraktiotasolla kuin kerros N-2. Tällöin ylemmän kerroksen palvelupyyntö yleensä jakautuu useammaksi yksinkertaisemmaksi palveluksi.

Toinen tapaus on edellisen kaltainen siten, että kutsu käy jokaisessa kerroksessa, mutta se eroaa edellisestä kuitenkin siten, että ensimmäisen palvelun pyytäjä on kerros 1. Tällä tavalla kutsu siirtyy alhaalta ylöspäin saavuttaen ylimmän kerroksen. Ensimmäisessä tapauksessa kerros tekee pyyntöjä (request) alemmalle kerrokselle, kun taas tässä tapauksessa voidaan puhua palveluilmoituksesta (notifications). Yleensä alemman tason ilmoitukset nivoutuvat yhteen, kun abstraktiotasolla mennään ylemmäksi, mutta ne voivat pysyä myös yhtenä ilmoituksena. Tämän tapaista kontrollin kulkua voidaan ajatella tapahtuvan, kun jokin laite havaitsee syötteen ja siirtää kontrollin laiteajurille ja sitä kautta käyttöjärjestelmälle (Bowman, 1998). Tällaisesta kontrollin kulusta käytetään nimitystä alhaalta-ylös.

Seuraavat kaksi tapausta ovat ensimmäisen ja toisen tapauksen muunnoksia. Erona on vain se, että kontrollin ei välttämättä tarvitse mennä aina jokaiselle kerrokselle. Perusperiaate kutsun siirtymisestä seuraavalle kerrokselle säilyy, mutta jos kerros N voi suorittaa siltä halutun palvelun ilman, että se kutsuu kerrosta N-1, se voi olla kutsumatta sitä ja toimia niin kuin se olisi viimeinen kerros. Tällainen dynamiikka voisi tulla kyseeseen silloin, kun yksi kerros toimii välimuistin tavoin.



Kuva 6: Kontrollin kulku kahdessa pinossa Buschmann *& al*. (2001) mukaan.

Viimeinen tapaus on yhdistelmä edellisiä tapauksia. Siinä on kaksi N-kerroksista pinoa, jotka keskustelevat keskenään. Toinen kerroksista on ylhäältä-alas-tyylinen, ja toinen alhaalta-ylös-tyylinen. Kumpikin kerros toimii itsenäisenä edellisten tapausten

mukaisesti, mutta sillä poikkeuksella, että viimeinen kerros ensimmäisestä pinosta kutsuu toisen kerroksen viimeistä kerrosta. Kuva 6 havainnollistaa tällaista dynamiikkaa. Kuvassa kutsu alkaa vasemman pinon kerroksesta N päättyen oikeanpuoleisen pinon kerrokseen N. Vasemmanpuoleisessa pinossa kutsu kulkee ylhäältä alaspäin, minkä jälkeen pinon ensimmäinen kerros kutsuu oikeanpuoleisen pinon ensimmäistä kerrosta. Vasemmanpuoleisessa pinossa kerrokset ilmoittavat vuorollaan tapahtumasta ylemmälle kerrokselle ja näin kontrolli siirtyy lopulta kerrokseen N. Edellä kuvattu kahden kerroksittaisen pinon käyttö on yleistä kehiteltäessä protokollapinoja. Tunnetuin esimerkki lienee ISO:n OSI-malli.

3.5 Suunnittelu ja toteutus

Arkkitehtuurin suunnittelu on se vaihe ohjelmistokehitystä, jossa arkkitehtuurityyli ja arkkitehtuuritason ratkaisut kiinnitetään. Ohjelmistotuotannossa se tarkoittaa yleensä vaihetta analyysi- ja suunnitteluvaiheiden välissä. Suunniteltaessa ja toteutettaessa kerroksittaista arkkitehtuuria voidaan käyttää *Buschmannin & al.*:in (2001) esittelemiä työvaiheita, jotka ilmenevät taulukosta 1. Nämä vaiheet eivät välttämättä sovi kaikille järjestelmille ja suunnittelumalleille; joskus esimerkiksi “jojo“-lähestymistapa (yo-yo approach) on parempi. Myöskään kaikkia vaiheita ei tarvitse orjallisesti noudattaa, vaan joitakin vaiheita voi jättää pois, jos niitä ei tarvita. Joissakin tapauksissa esimerkiksi rajapinnat pohjautuvat johonkin yleiseen standardiin; silloin rajapinnan määrittelevän suunnitteluvaiheen voi jättää kokonaan pois. Seuraavaksi esitetään kaikki Buschmannin *& al.*:n (2001) esittelemät vaiheet, sen jälkeen tarkastellaan jokaista vaihetta erikseen hieman tarkemmin. Samassa yhteydessä hahmotellaan myös vaalijärjestelmäesimerkin toteutusta. Kerroksittaiseen arkkitehtuurityyliin perustuva vaalijärjestelmän lähdekoodi on kokonaisuudessaan liitteessä 1.

Ensimmäisessä vaiheessa määritellään kriteerit sille, millä abstraktiotasolla ohjelmiston kerroksiin jaottelu tulee tapahtumaan. Esimerkiksi shakkipeli voisi sisältää seuraavanlaisen jaottelun: pelinappulat, liikkuminen, keskitason taktiikka ja koko pelin strategia.

Toisessa vaiheessa päätetään abstraktiotasojen lukumäärä ensimmäisessä vaiheessa määriteltyjen abstraktiokriteerien mukaan. Jokainen abstraktiotaso vastaa yhtä kerrosta. Joskus abstraktiotasojen kartoitus kerroksiksi ei ole ilmiselvää. Liiallinen kerrosten

määrä voi aiheuttaa tarpeetonta työtä, kun taas liian vähäinen määrä kerroksia yleensä johtaa huonoon ohjelmiston rakenteeseen.

Kolmannessa vaiheessa nimetään kerrokset ja kiinnitetään niille tehtävät. Ylimmän kerroksen tehtävä on suunniteltavan ohjelmiston kokonaistehtävä. Voidaankin sanoa, että alempien kerrosten tehtävä on auttaa ylintä kerrosta suoriutumaan tehtävästään. Jos suunnittelutapa on “alhaalta-ylös“, niin alin kerros toimii perusrakenteena, jonka varaan ylemmät kerrokset voidaan rakentaa.

Neljännessä vaiheessa määritellään kerrosten tarjoamat palvelut. Tässä vaiheessa on tärkeää muistaa, että kerrosten tulee olla itsenäisiä. Hyvänä ohjesääntönä voidaan sanoa, että kerroksen tarjoamien palvelujen määrän tulisi vähentyä sitä mukaa, mitä alemmas kerroksissa mennään. Tätä ohjetta kutsutaan ylösalaisin olevaksi uudelleenkäytön pyramidiksi (inverted pyramide of reuse).

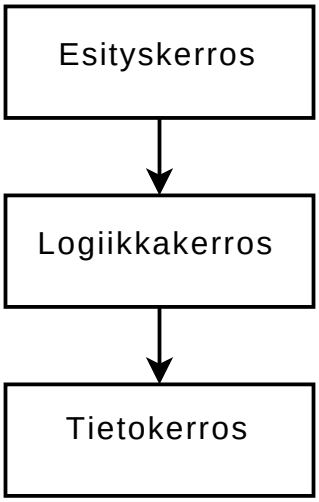
Viidennessä vaiheessa hiotaan ja “puhdistetaan“ kerrosjakoa. Tämä tapahtuu siten, että toistetaan vaiheet 1-4. Yleensä on mahdotonta määrittää abstraktiokriteerejä täsmällisesti, ennen kuin on hieman perehdytty ja suunniteltu kerrosten sisältöä ja palveluja. Buschmannin *& al*.:in (2001) mukaan on yleensä väärin tehdä suunnittelu siten, että ensiksi määritellään komponentit ja palvelut, minkä jälkeen niistä kootaan kerroksittainen arkkitehtuuri komponenttien välisten käyttösuhteiden mukaan.

Kuvassa 7 on vaalijärjestelmän kerrokset viidennen vaiheen jälkeen. Ylimmän kerrok-

Taulukko 1: Kerroksittaisen tyylin suunnitteluvaiheet (Buschmann *& al*., 2001).

Vaihe	*Tehtävä*
1	Määritä abstraktiokriteerit tehtävien jakamiseksi kerroksiksi.
2	Päätä abstraktiotasojen lukumäärä abstraktiokriteerien mukaan.
3	Nimeä kerrokset ja kiinnitä niiden tehtävät.
4	Määrittele kerrosten tarjoamat palvelut.
5	Jalosta ja kehitä kerroksia.
6	Määritä rajapinta jokaiselle kerrokselle.
7	Jäsennä yksittäiset kerrokset.
8	Määritä yhteydet viereisten kerrosten välille.
9	Erota viereiset kerrokset.
10	Suunnittele virheidenkäsittelystrategia.

sen eli esityskerroksen vastuulla on järjestelmän käyttöliittymä. Keskimmäinen kerros eli logiikkakerros huolehtii järjestelmään kohdistuvasta laskennasta. Alin kerros eli tietokerros huolehtii tiedon varastoinnista. Tässä tapauksessa se tarkoittaa puolueiden ja niiden saamien äänien varastointia.



Kuva 7: Vaalijärjestelmän kerrokset.

Kuudennessa vaiheessa määritellään rajapinnat jokaiselle kerrokselle. Jos kerros A ei tiedä mitään kerroksen A-1 sisäisestä rakenteesta, niin kerrokselle A-1 tulisi määritellä mahdollisimman suppea rajapinta. Lasilaatikko-lähestymisellä taas puolestaan tarkoitetaan sitä, että kerros A näkee kerroksen A-1 rakenteen, ja rajapinta koostuu kerroksen A-1 komponenttien rajapinnoista. Mustalaatikon ja lasilaatikon yhdistelmä on nimeltään harmaalaatikko, missä kerros A tietää kerroksen A-1 rakenteen ja käyttää jokaista kerroksen A-1 komponenttia erikseen, mutta ei ole tietoinen yksittäisen komponentin toiminnasta.

Hyvien suunnitteluperiaatteiden mukaan tulisi käyttää mustalaatikko-menetelmää, jos vain suinkin mahdollista, koska se tukee järjestelmän kehittymistä ja siihen tehtäviä muutoksia muita esiteltyjä lähestymistapoja paremmin (Buschmann *& al*., 2001).

Seitsemännessä vaiheessa määritellään yksittäisen kerroksen rakenne. Kun yksittäinen kerros on monimutkainen, tulisi se jakaa useammaksi erilliseksi komponentiksi. On huomioitava, että myös yksittäisen kerroksen arkkitehtuuria voidaan tarkastella kuten koko järjestelmän arkkitehtuuria. Yksittäisen kerroksen arkkitehtuurityyli voikin olla esimerkiksi putkia ja suodattimia -tyylinen.

Kahdeksannessa vaiheessa määritellään päällekkäisten kerrosten välinen viestintätapa.

Yleisin viestintätapa on niin sanottu työntömalli (push model). Kun kerros A kysyy palvelua kerrokselta A-1, kaikki kyselyn informaatio “työnnetään“ samalla tälle kerrokselle. Erilaiset viestintätavat ovat huomattavasti tärkeämmässä roolissa putkia ja suodattimia -arkkitehtuurityylissä.

Yhdeksännessä vaiheessa erotellaan päällekkäiset kerrokset. Tämä voi tapahtua monella tavalla. Erottelulla tarkoitetaan sitä, että kerros tehdään riippumattomaksi naapurikerroksista, ja se voidaan haluttaessa vaihtaa toiseen kerrokseen, myös suorituksen aikana.

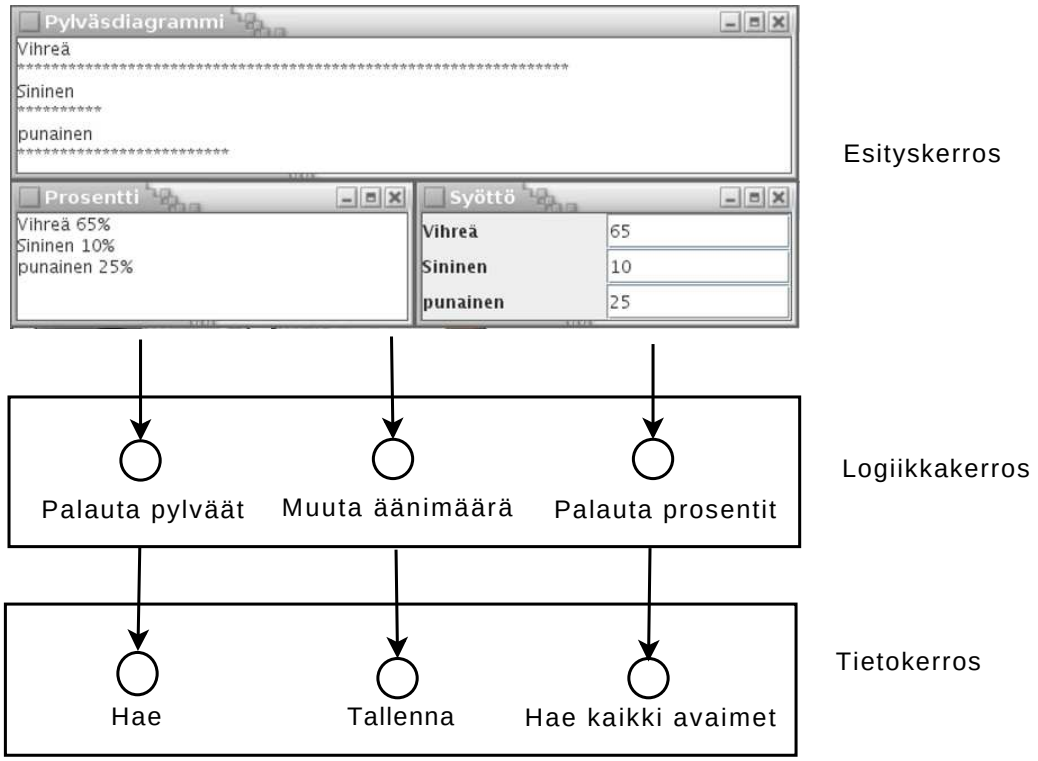
Viimeisessä vaiheessa suunnitellaan virheidenkäsittelystrategia. Virheidenkäsittely voi kerroksittaisessa tyylissä olla suorituskyvyn suhteen aika raskasta. Kerroksessa tapahtuva virhe voidaan käsitellä joko samassa kerroksessa tai se voidaan lähettää ylemmälle kerrokselle. Ensimmäisessä tapauksessa kerroksen täytyy muuntaa virheestä saatu informaatio ylemmän kerroksen ymmärtämään muotoon. Yleinen ohje virheiden käsittelylle Buschmannin *& al*.:in (2001) mukaan on, että virhe tulisi käsitellä mahdollisimman alhaisella tasolla. Tällä tavalla vältetään virheiden kasautumista ylimpiin kerroksiin.

Vaikka järjestelmä voitaisiin loogisesti jakaa kerroksiin, voi olla vaikeaa löytää oikea abstraktiotaso kerroksille. Jakoa kerroksiin vaikeuttavat myös mahdolliset tehokkuusvaatimukset. Kun kerroksia on liian vähän menetetään kerroksittaisen tyylin tarjoamia ominaisuuksia. Näitä ominaisuuksia ovat esimerkiksi uudelleenkäytettävyys, muutettavuus ja siirrettävyys. Toisaalta liiallinen määrä kerroksia tuo tarpeetonta kompleksisuutta (Buschmann *& al*., 2001).

3.6 Kerroksittaisella arkkitehtuurityylillä toteutettu esimerkkijärjestelmä

Tässä alaluvussa esitellään kerroksittaiseen ohjelmistoarkkitehtuurityyliin perustuvan vaalijärjestelmäesimerkin toteutus. Toteutus on kokonaisuudessaan liitteessä 1. Vaalijärjestelmä on hyvin yksinkertainen. Siinä on graafinen käyttöliittymä, jolla päivitetään vaalin tilanne ja näytetään vaalin äänijakauma. Vaalijärjestelmä pitää kirjaa vaaliin liittyvistä puolueista ja niiden saamista äänimääristä.

Vaalijärjestelmän kerrosjako hahmoteltiin jo luvussa 3.5. Siinä ylin kerros, eli esitys-



Kuva 8: Luonnos vaalijärjestelmästä.

kerros, huolehtii järjestelmän käyttöliittymästä. Keskimmäinen kerros, eli logiikkakerros, huolehtii vaaliin liittyvästä laskennasta ja toiminnallisuudesta. Alin kerros, eli tietokerros, toimii tietovaraston tavoin huolehtien järjestelmän tietojen varastoinnista. Kerrokset on toteutettu mustanlaatikon omaisesti ja kerrosten välinen kommunikointi tapahtuu ylhäältä-alas — esityskerros kutsuu siis vain logiikkakerroksen palveluja ja logiikkakerros vain tietokerroksen palveluja. Kuvassa 8 on esitetty vaalijärjestelmän käyttöliittymä sekä hahmotelma kerrosten välisestä kommunikoinnista.

Kerroksittaista arkkitehtuurityyliä käytettäessä on tärkeintä jakaa kerrokset oikein ja määritellä kerroksille selkeät palvelut. Kerrosten toteutukset eivät itsessään ole tyylin kannalta merkityksellisiä. Kuvassa 9 ovat kerroksittaisella tyylillä toteutetun vaalijärjestelmän kannalta hyvin oleelliset asiat, siis kerrosjako ja kerrosten rajapinnat. Esityskerroksen rajapinta koostuu järjestelmän käyttöliittymän alustuksessa tarvittavista palveluista. Logiikkakerroksessa ovat kaikki esityskerroksen käyttämät vaalijärjestelmälle ominaiset palvelut. Tietokerros tarjoaa nimensä mukaisesti tiedon käsittelyyn liittyvät palvelut.

```
interface EsityskerrosR extends ActionListener {
  public void alustaIkkunat();
  public void lisaaPuolueet(List<String> puolueet);
}
interface LogiikkakerrosR {
  public void alustaVaalit(List<String> puolueet);
  public void muutaAanimaara(
    String puolue, String aanimaara);
  public Map<String, String> palautaAanimaarat();
  public String palautaPylvaat();
  public String palautaProsentit();
}
interface TietokerrosR {
  public Long hae(String avain);
  public Set <String> haeKaikkiAvaimet();
  public boolean tallenna(String avain, Long arvo);
}
```

Kuva 9: Vaalijärjestelmän kerrosten toteutusrajapinnat

3.7 Kerroksittaisen arkkitehtuurin edut

Kerroksittaisella arkkitehtuurityylillä on useita hyödyllisiä ja haluttuja ominaisuuksia. Näistä merkittävin on sen selkeä ja yksinkertainen rakenne. Kerroksittainen rakenne mahdollistaa suunnittelun osittamisen kasvavan abstraktiotason mukaan (Garlan & Shaw, 1998). Tällä tavalla kompleksinen systeemi voidaan jakaa useaksi osatehtäväksi, joita on helpompi käsittää ja hallita. Kerroksittainen tyyli parantaa ylläpidettävyyttä ja muokattavuutta. Muutokset yhteen kerrokseen vaikuttavat korkeintaan naapurikerroksiin, hyvin suunniteltuna ei niihinkään. Kerroksittainen tyyli mahdollistaa uudelleen käytettävyyden. Tyyli ei ota kantaa kerroksen toteutukseen, vaan kerroksella voi olla useita erilaisia toteutuksia. Tästä johtuen kerrosta voidaan käyttää uudelleen myös muissakin ohjelmistoissa. Kerrosten uudelleenkäyttö vaatii viereisten kerrosten tuen rajapinnoille. Useimmiten kerroksittaisesta arkkitehtuurista irrotetaan tarvittava määrä alimpia kerroksia ja päälle lisätään uutta toiminnallisuutta. Hyviä esimerkkejä kerrosten uudelleenkäytöstä ovat OSI:n ISO-malli sekä jotkin X-ikkunointiprotokollat (Buschmann *& al.*, 2001).

Testaus hyötyy kerroksittaisen tyylin hyvin määritellyistä rajapinnoista (Microsoft Corporation, 2008a). Ensinnäkin yksittäisen kerroksen testaus hyötyy paljon kerroksen täsmällisesti ja tarkasti määritellyitä rajapinnoista. Toiseksi kerroksen rajapinnat ovat tärkeä osa kerroksittaista tyyliä, ja siitä johtuen myös niiden suunnitteluun ja dokumentointiin kiinnitetään paljon huomiota.

Buschmann *& al*. (2001) luettelee eduksi myös tuen standardoinnille. Hyvin määritelty ja yleisesti hyväksytty abstraktiotaso mahdollistaa tehtävän ja rajapinnan standardoinnin. Tämä taas mahdollistaa esimerkiksi sen, että eri valmistajat voivat toisistaan riippumatta tehdä kerroksen toteutuksen. Hyvänä esimerkkinä tästä voisi mainita POSIX-ohjelmointirajapinnan.

Jos ohjelmiston toteutuksen työvaiheet voidaan jaotella kerrosten mukaan, niiden erillinen toteutus on mahdollista ilman, että muita kerroksia on toteutettu tai toteutetaan yhtä aikaa.

3.8 Kerroksittaisen arkkitehtuurin haittapuolet

Kerroksittainen arkkitehtuurityyli voi tuoda mukanaan myös joukon huonoja ominaisuuksia. Buschmannin *& al*.:in (2001) mukaan kerroksittainen tyyli on yleensä tehottomampi kuin esimerkiksi monoliittinen rakenne. Tämä johtuu siitä, että jokainen kerros tekee yleensä muutoksia käsiteltävään tietoon huolimatta siitä, olisiko se lopputuloksen kannalta välttämätöntä. Sama koskee myös tulosten palautusta ja virhetilanteiden käsittelyä. Alimmalla kerroksella tapahtunut virhe matkaa jokaisen kerroksen läpi, ja jokainen kerros käsittelee sitä sille kuuluvalla tavalla, vaikka virhe voitaisiin tehokkaammin käsitellä suoraan ylimmällä kerroksella ohittamalla väliin jäävät kerrokset. Tällä tavalla menetettäisiin kuitenkin tärkeitä kerroksittaisen tyylin ominaisuuksia, kuten esimerkiksi uudelleenkäyttö.

Kerroksittaista tyyliä suunniteltaessa tulee esiin tyylin suunnittelun vaikeus; kuinka jaotella ohjelmisto oikealla tavalla kerroksiin. Liian vähän kerroksia heikentää tyylin ominaisuuksia tai hävittää ne kokonaan, kun taas liiallinen määrä kerroksia merkitsee tehottomampaa ratkaisua ja rakenteen monimutkaistumista. Päätös ohjelmiston kerroksien jaottelusta on yleensä vaikea ja todella tärkeä, jotta tyylin kaikki hyvät ominaisuudet pääsevät tulemaan esille.

Muutokset alimpien kerrosten rajapintoihin saattavat vaikuttaa ylempien kerrosten rajapintoihin (Microsoft Corporation, 2008a). Tämä ongelma on kuitenkin helposti vältettävissä, jos sen olemassaolo on tiedossa kerrosten rajapintoja kiinnitettäessä.

4 Malli-näkymä-ohjain -ohjelmistoarkkitehtuurityyli

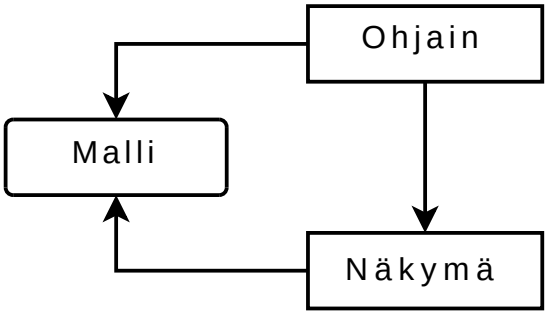
Tässä luvussa esitellään malli-näkymä-ohjain -ohjelmistoarkkitehtuurityyli. Luvun rakenne on edellisen luvun kaltainen, jotta se helpottaisi lukijaa hahmottamaan tyylien välisiä eroja ja luomaan yhdenmukaista näkökulmaa tutkielmassa käsiteltäville arkkitehtuurityyleille. Tässä luvussa hahmotellaan edellisen luvun tapaan yksinkertainen vaalijärjestelmä käyttäen tässä luvussa käsiteltävää ohjelmistoarkkitehtuurityyliä. Esimerkkijärjestelmän tarkoituksena on havainnollistaa kappaleissa esitettyjä asioita.

4.1 Johdatus malli-näkymä-ohjain -ohjelmistoarkkitehtuurityyliin

Malli-näkymä-ohjain -ohjelmistoarkkitehtuurityyli (Model-View-Controller) kuuluu luvussa 2.4 esitellyn ohjelmistoarkkitehtuurityylien jaottelun mukaan “vuorovaikutteiset järjestelmät“ -ryhmään (interactive systems) (Buschmann *& al.*, 2001). Tähän ryhmään kuuluville tyyleille on ominaista niiden soveltuvuus järjestelmiin, joissa tarvitaan ihmisen ja järjestelmän välistä vuorovaikutusta. Tämä vuorovaikutus tapahtuu yleensä graafisen käyttöliittymän välityksellä.

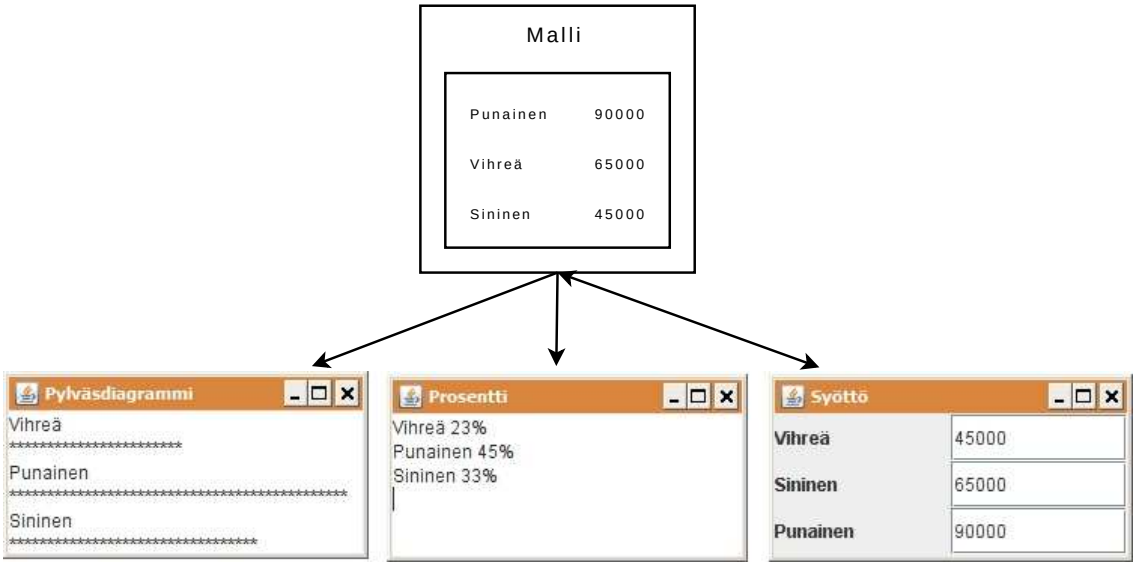
Malli-näkymä-ohjain -ohjelmistoarkkitehtuurityylin esitteli ensimmäistä kertaa Trygve M. H. Reenskaug vuonna 1978 työskennellessään Xerox PARC:n Smalltalk-ohjelmointikielen kehitysryhmässä (Reenskaug, 1979). SmallTalk-80^{TM}-kehittäjät pyrkivät luomaan joukon yleiskäyttöisiä komponentteja, joiden avulla he pystyisivät vaivattomasti ja nopeasti kehittämään vuorovaikutteisia graafisia järjestelmiä. Malli-näkymä-ohjain -ohjelmistoarkkitehtuurityyli syntyi tämän luomistyön tuloksena. Tyyliä käytettiin itse SmallTalk-80^{TM}-kehitysympäristön käyttöliittymän arkkitehtuurina, ja sitä on sen jälkeen käytetty myös useissa vuorovaikutteisissa graafisen käyttöliittymän sisältävissä järjestelmissä ja ohjelmakehikoissa, kuten Mac OS X:n Cocoa-sovelluskehitysympäristön kehyksissä (Apple, 2007).

Burbeckin (1992) ja Reenskaugin (1979) mukaan Malli-näkymä-ohjain -ohjelmistoarkkitehtuurityylissä reaalimaailman mallinnus, käyttäjän syötteet ja visuaalinen palaute on jaettu kolmeen toisistaan selvästi erotettuihin komponentteihin, joista jokainen on erikoistunut omaan tehtäväänsä. Nämä komponentit ovat nimeltään malli (model), näkymä (view) ja ohjain (controller). Kuvassa 10 on pelkistetty esitys malli-näkymä-ohjain -ohjelmistoarkkitehtuurista (Microsoft Corporation, 2008b).



Kuva 10: Malli-näkymä-ohjain -ohjelmistoarkkitehtuurityylin komponentit (Microsoft Corporation, 2008b).

Alkuperäinen malli-näkymä-ohjain -ohjelmistoarkkitehtuurityylin määritelmä on Reenskaugin (1979) luoma. Hän määrittää tyylin komponenttien vastuut niin, että malli-komponentti on vastuussa järjestelmän ydintoiminnallisuudesta ja sisältää laskentaan liittyvät rakenteet, tiedot ja tilan. Näkymäkomponentti, nimensä mukaan, luo käyttäjälle visuaalisen näkymän malli-komponentista. Eri näkymien avulla mallia voidaan tarkastella eri konteksteissa tai vain eri näkökulmista. Ohjainkomponentti on vastuussa käyttäjän ja järjestelmän välisen vuorovaikutuksen ohjaamisesta sekä järjestelmän syötteiden käsittelystä. Näkymä- ja ohjainkomponentit yhdessä muodostavat järjestelmän käyttöliittymän. Mallikomponenttiin kuuluu myös muutostenhallintamekanismi (change-propagation mechanism), joka pitää huolen käyttöliittymän ja mallin tietoeheydestä. Tämän avulla mallissa tapahtuvat muutokset heijastuvat käyttöliittymälle.

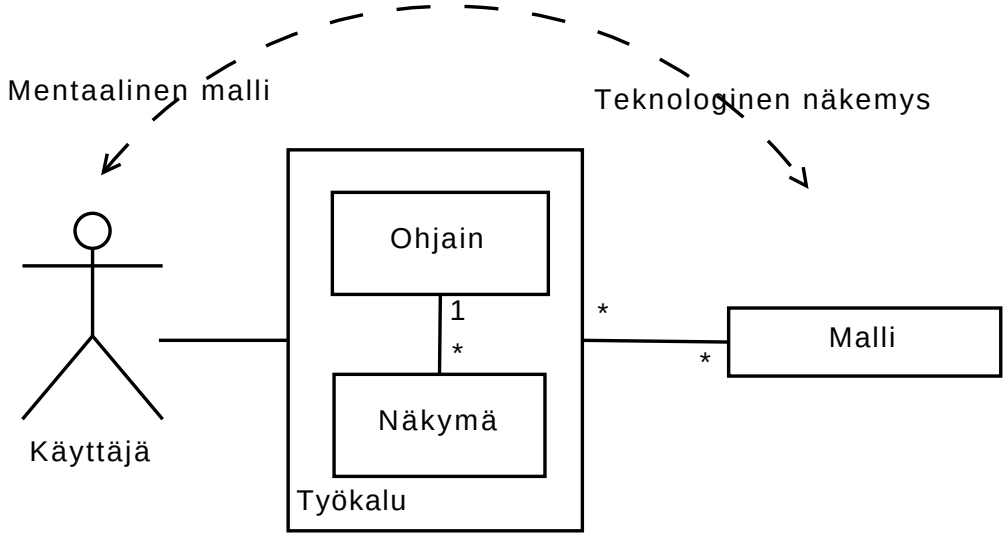


Kuva 11: Vaalijärjestelmäesimerkin näkymät ja mallin tila.

Malli-näkymä-ohjain -tyyliä voidaan havainnollistaa kuvan 11 mukaisella vaalijärjestelmällä. Järjestelmä mukailee Buschmann & *al*.:n (2001) esittämää vaaliesimerkkiä. Siinä mallikomponentti pitää kirjaa vaaliin liittyvistä puolueista ja niiden saamista äänimääristä. Järjestelmässä on kolme erillistä näkymää, joista kukin havainnollistaa äänijakaumaa hieman eri tavalla. Kuvan kaksi vasemmanpuoleista näkymää ainoastaan esittää tiedon, kun taas oikeanpuoleisimman näkymän välityksellä voidaan olla myös vuorovaikutuksessa järjestelmän kanssa.

4.2 Rakenne ja terminologia

Kuten jo aiemmin on mainittu, malli-näkymä-ohjain -ohjelmistoarkkitehtuurityyli jakaa järjestelmän kolmeen komponenttiin, jotka ovat malli-, näkymä- ja ohjauskomponentit. Järjestelmän kannalta tärkein komponentti on malli. Mallikomponentti huolehtii järjestelmän ydintoiminnallisuudesta, toisin sanoen sen vastuulla on järjestelmään kohdistuva laskennallinen toiminnallisuus. Mallikomponentin nimi tulee Reenskaugin (1979) esittämästä ajatuksesta, jossa järjestelmällä pyritään mallintamaan ihmisen mentaalista mallia. Kuva 12 havainnollistaa tätä ajatusta.



Kuva 12: Reenskaugin (1997) hahmotelma malli-näkymä-ohjain -ohjelmistoarkkitehtuurityylistä.

Mallikomponentti on hyvin itsenäinen. Sillä on ainoastaan julkinen rajapinta, jota sekä näkymä- että ohjainkomponentti kutsuvat. Mallikomponentin julkinen rajapinta sisältää kaikki mallin muille komponenteille tarjoamat palvelut.

Mallikomponenttiin liittyy muutostenhallintamekanismi, johon mallin tilassa tapahtuvista muutoksista kiinnostuneet komponentit voivat rekisteröityä. Rekisteröityneet komponentit saavat muutoksista ilmoituksen, johon ne voivat reagoida haluamallaan tavalla. Vaalijärjestelmän tapauksessa kaikki näkymäkomponentit ovat kiinnostuneita mallikomponentin tilan muutoksista. Kun äänijakaumassa tapahtuu muutos, näkymäkomponentit saavat siitä tiedon, minkä jälkeen ne voivat päivittää tuottamansa graafiset esitykset vastaamaan sen hetkistä äänijakaumaa. Malli kommunikoi näkymien tai kontrollien kanssa ainoastaan muutostenhallintamekanismin avulla. Muutostenhallintamekanismin toteuttamisessa voidaan käyttää hyödyksi tarkkailijasuunnittelumallia (Observer) (Gamma *& al*., 1995).

Näkymäkomponentti esittää — yleensä visuaalisesti — mallikomponentin tiedot käyttäjälle (Buschmann *& al*., 2001). Eri näkymät esittävät tiedot eri tavoilla. Esimerkiksi vaalijärjestelmässä äänijakauma esitetään sekä pylväsdiagrammina että prosentuaalisena jakaumana. Jokainen näkymä määrittää näkymänpäivitysrutiinin, jota käytetään silloin, kun muutostenhallintamekanismi ilmoittaa näkymään vaikuttavasta mallikomponentin tilan muutoksesta. Jokainen järjestelmän näkymä tulee liittää malliin. Liitos tapahtuu muutostenhallintamekanismiin rekisteröitymällä. Jokainen näkymä liitetään vain yhteen mallikomponenttiin (Krasner & Pope, 1988). Usein näkymät tarjoavat toiminallisuutta, jonka avulla ohjain voi vaikuttaa siihen, miten käyttäjä näkee näkymän. Tämä on tärkeä ominaisuus sellaisten toiminnallisuuksien kannalta, joilla ei ole vaikutusta malliin. Jos esimerkiksi vieritetään ikkunan vierityspalkkia, näkymä muuttuu, vaikka mallissa ei tapahdu muutosta (Buschmann *& al*., 2001). Jos näkymä taas ainoastaan esittää mallin tietoja, se ei tarvitse ohjainta lainkaan, tai ohjain voi olla ilman varsinaista toiminnallisuutta (Buschmann *& al*., 2001).

Ohjainkomponentti toimii liittimenä järjestelmän ja käyttäjän välillä. Ohjainkomponentti ottaa käyttäjän syötteet tapahtumina vastaan. Käyttöliittymän tyypistä ja suoritusympäristöstä riippuu, kuinka nämä tapahtumat välitetään ohjaimelle. Yksinkertaistaen voidaan sanoa, että jokainen ohjain sisältää tapahtumankäsittelyrutiinin, jota kutsutaan, kun sille sopiva tapahtuma on havaittu. Ohjain käsittelee tapahtuman ja muokkaa siitä kutsun tapauksesta riippuen joko mallille tai näkymälle. Jos ohjaimen toiminnallisuus on riippuvainen mallista, ohjain rekisteröityy myös muutostenhallintamekanismille ja toteuttaa riippuvuuden vaatiman päivitystoiminnon.

Krasner & Pope (1988) kuvaa malli-näkymä-ohjain -ohjelmistoarkkitehtuurin komponenttien välistä vuorovaikutusta seuraavilla vaiheilla. Ensimmäisessä vaiheessa käyttä-

jä suorittaa jonkin toiminnon, jonka ohjainkomponentti käsittelee. Toisessa vaiheessa ohjain kutsuu mallia. Kolmannessa vaiheessa malli muuttaa tilaansa ja ilmoittaa muutoksesta siitä kiinnostuneille komponenteille, yleensä näkymille. Neljännessä vaiheessa näkymäkomponentti päivittää itsensä vastaamaan mallikomponentin senhetkistä tilaa.

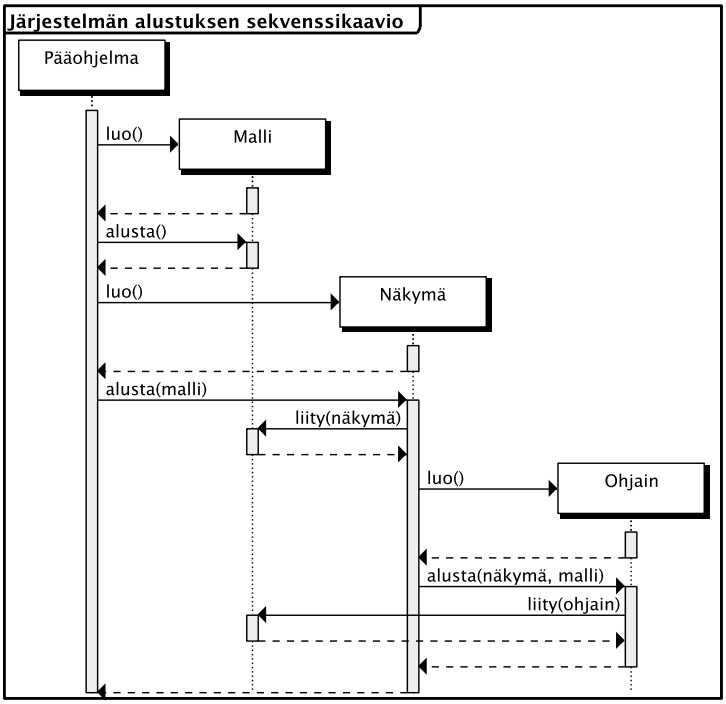
4.3 Kontrollin kulku

Malli-näkymä-ohjain -arkkitehtuuriin perustuvan järjestelmän komponenttien välistä vuorovaikutussuhdetta suorituksen aikana voidaan kuvata sekvenssikaavion avulla (Buschmann *& al*., 2001). Kontrollin kulku komponenttien välillä on jaettu kahteen tapaukseen, ja kummassakin tapauksessa seurataan vain yhtä näkymä-ohjaus -paria. Ensimmäisessä tapauksessa käydään läpi järjestelmän alustus. Toinen tapaus kuvaa mallin tilaa muuttavan tapahtuman käsittelyä.

Järjestelmän alustus alkaa mallikomponentin alustuksella. Tämän jälkeen alustetaan näkymäkomponentti, jolle annetaan viittaus mallikomponenttiin. Seuraavaksi näkymä rekisteröityy mallikomponentin muutostenhallintamekanismiin ja luo itselleen ohjaimen. Näkymä välittää ohjaimelle mallikomponentistaan tiedon. Tarvittaessa ohjainkomponentti rekisteröityy mallikomponentin muutostenhallintamekanismiin. Nyt järjestelmä on alustettu, ja ohjain-komponentti on valmis käsittelemään järjestelmän tapahtumia. Kuvassa 13 on kuvattu järjestelmän alustuksen vaiheet (Buschmann *& al*., 2001).

Malli-näkymä-ohjain -arkkitehtuurityyliin perustuvan järjestelmän alustukseen on myös kiinnitettävä huomiota, koska komponenttien välinen vuorovaikutus on hieman poikkeuksellinen verrattuna esimerkiksi puhtaan kerroksittaisen arkkitehtuurin komponenttien väliseen vuorovaikutukseen. Vuorovaikutussuhteet luodaan yleensä alustettaessa, mutta ne voidaan luoda myös dynaamisesti suorituksen aikana (Chow *& al*., 1996).

Ohjainkomponentti ottaa tapahtumankäsittelyrutiinissa käyttäjän syötteen vastaan ja muokkaa tapahtumasta kutsun mallikomponentille (kuva 14). Malli suorittaa palvelukutsun vaatimat toimenpiteet, kuten esimerkiksi muuttuneiden tietojen tallentamisen tietovarastoon. Tämä tapahtuma muuttaa mallin tilaa. Malli ilmoittaa muutoksesta kaikille muutostenhallintamekanismiin rekisteröityneille kutsumalla niiden päivitysrutii-



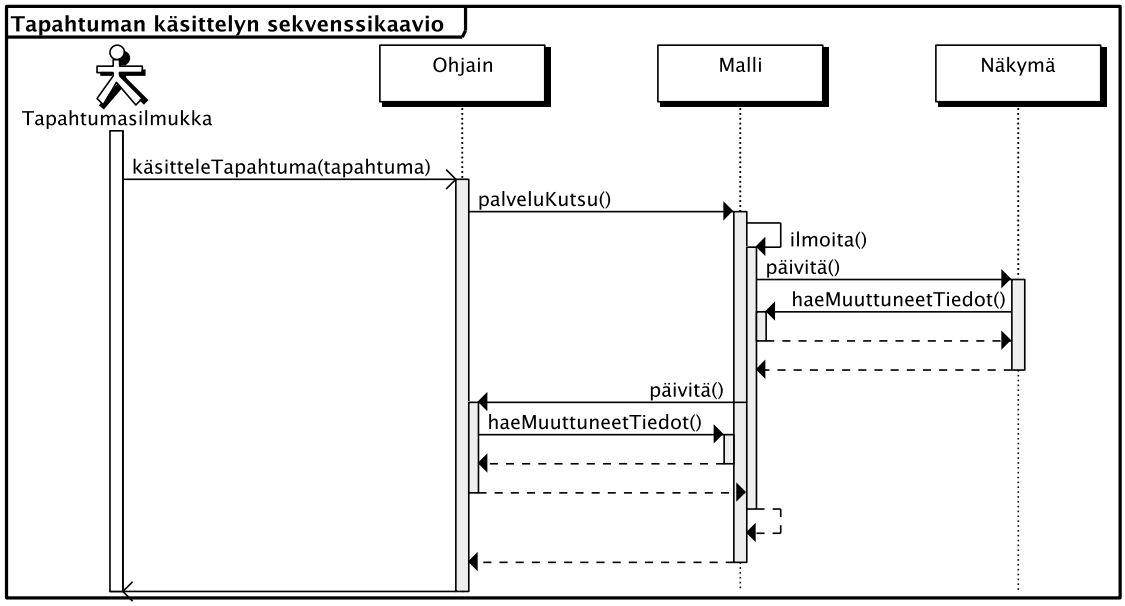
Kuva 13: Sekvenssikaavio alustuksen vaiheista (Buschmann *& al.*, 2001).

neja. Jokainen näkymä hakee muuttuneet tiedot ja päivittää omat näyttönsä. Myös jokainen ohjain hakee muuttuneet tiedot ja suorittaa muutoksen aiheuttamat toiminnot, esimerkiksi poistaa käytöstä joitakin alasvetovalikkoja. Kun tapahtuma on käsitelty, poistutaan tapahtuman käsitelleestä ohjaimesta, ja suoritus siirtyy odottamaan seuraavaa tapahtumaa esimerkiksi jonkinlaiseen tapahtumasilmukkaan.

4.4 Suunnittelu ja toteutus

Buschmann *& al*. (2001) jakaa toteutuksen 10 kohtaan, joista kuusi ensimmäistä sopii perustaksi kaikkiin malli-näkymä-ohjain -arkkitehtuurityyliin perustuviin järjestelmiin. Kohdat seitsemästä kymmeneen tuovat järjestelmään lisää joustavuutta ja vapautta, ja ne vievät järjestelmän rakennetta sovelluskehikon suuntaan.

Toteutuksen ensimmäisessä vaiheessa erotetaan ihmisen ja tietokoneen välinen vuorovaikutus ydintoiminnallisuudesta, eli luodaan mallikomponentti. Mallikomponentin tietorakenteet ja ytimelle välttämättömimmät toiminnot suunnitellaan. Sitten päätetään, mitkä osat mallikomponentista paljastetaan ohjainkomponentille, ja luodaan paljastettujen osien mukainen julkinen rajapinta.



Kuva 14: Järjestelmän toiminta suorituksen aikana (Buschmann *& al*.,2001).

Toisessa vaiheessa toteutetaan mallikomponentin muutostenhallintamekanismi. Toteutuksessa voi käyttää hyväksi julkaise-tilaa -suunnittelumallia (publish-subscribe) (Gamma *& al*., 1995) niin, että malli-komponentti asetetaan julkaisijan rooliin. Muutostenhallintamekanismi toteutetaan siten, että mallikomponentti pitää kirjaa kaikista rekisteröityneistä tilaajista ja ilmoittaa niille tapahtuneista muutoksista.

Kolmannessa vaiheessa jokaisen näkymän ulkoasu suunnitellaan, ja niiden näyttämisessä tarvittavat rutiinit toteutetaan. Sitten toteutetaan näkymien päivitysrutiinit, jotka liittyvät muutostenhallintamekanismiin. Myös näkymäkomponentin alustusrutiinit suunnitellaan ja toteutetaan tässä vaiheessa. Alustuksessa näkymäkomponentti rekisteröityy mallin muutostenhallintamekanismiin ja luo suhteen siihen liittyvään ohjainkomponenttiin. Näkymäkomponentti on alustuksen jälkeen yleensä käyttäjän nähtävissä.

Neljännessä vaiheessa suunnitellaan ja toteutetaan ohjainkomponentti. Ohjaimelle luodaan alustusrutiini, joka sitoo ohjainkomponentin malli- ja näkymäkomponentteihin. Sitten alustusrutiini valmistautuu käsittelemään komponentille välitettäviä tapahtumia, eli järjestelmän syötteitä. Tapahtumien välitys ohjainkomponentille ei kuulu itse arkkitehtuurityyliin, mutta se voisi esimerkiksi olla ikkunointijärjestelmän toteuttama takaisinkutsu (Buschmann *& al*., 2001).

Koska ohjainkomponentti on riippuvainen mallikomponentin tarjoamasta julkisesta ra-

japinnasta, on tärkeää, että tämä julkinen rajapinta pysyy mahdollisemman muuttumattomana. Ohjain- ja mallikomponenttien välistä riippuvuutta voidaan vähentää käyttämällä oikeanlaisia suunnittelumalleja, kuten esimerkiksi komentosuunnittelumallia (command) (Buschmann*& al*., 2001, Gamma *& al*., 1995).

Viidennessä vaiheessa toteutetaan näkymä- ja ohjainkomponenttien välinen suhde. Siihen liittyvä ohjainkomponentti luodaan yleensä näkymän alustusvaiheessa. Näkymä- ja ohjainkomponenttien suhteen luonnissa voidaan käyttää hyväksi oliotehdassuunnittelumallia (Gamma *& al*., 1995).

Kuudennessa vaiheessa toteutetaan järjestelmän alustus. Ensiksi luodaan ja alustetaan mallikomponentti, sen jälkeen luodaan näkymä- ja ohjainkomponentit. Alustuksen jälkeen järjestelmän pitäisi olla toimintakunnossa, jolloin se on valmis ottamaan vastaan järjestelmään kohdistuvia syötteitä. Kuvassa 13 esitetty tapahtumasekvenssikaavio havainnollistaa järjestelmän alustusvaiheita.

Seuraavat kolme vaihetta tuovat järjestelmään lisää ominaisuuksia, mutta ne eivät ole millään tavalla pakollisia malli-näkymä-ohjain -arkkitehtuurin kannalta. Jos järjestelmän vaatimuksiin kuuluu, että näkymiä tulisi voida luoda ja tuhota suorituksen aikana, Buschmann *& al*. suosittelee toteuttamaan erillisen komponentin huolehtimaan tästä toiminnallisuudesta. Toteutukseen voi soveltaa esimerkiksi näkymänkäsittelysuunnittelumallia (View Handler). Toteutuksen viimeiset vaiheet luovat järjestelmään ohjelmakehikkomaisuutta ja vaikuttavat vain järjestelmän komponenttien välisiin siteisiin.

4.5 Malli-näkymä-ohjain -arkkitehtuurityylillä toteutettu esimerkkijärjestelmä

Seuraavaksi käydään läpi kohdassa 4.1 esitellyn yksinkertaisen vaalijärjestelmän malli-, näkymä- ja ohjainkomponenttien toteutusten rungot. Vaalijärjestelmän toteutus löytyy kokonaisuudessaan liitteestä 2.

Vaalijärjestelmän mallikomponentti on kuvan 15 mukainen. Tämä mallikomponentti on erittäin yksinkertainen, ja se koostuu yhdestä Java-ohjelmointikielen luokasta. Esimerkkiluokan julkiset metodit muodostavat komponentin julkisen rajapinnan. Mallikomponentilla on kolme julkista palvelua. Palvelulla muutaAanimaara voidaan muuttaa puolueen äänimäärää. PalautaPuolueetJaProsentit -palvelu palauttaa vaaliin liitty-

vät puolueet ja niiden prosentuaalisen äänijakauman. PalautaPuolueetJaAanet -palvelu palauttaa vaaliin liittyvät puolueet ja niiden äänimäärän.

```
public class Vaalijarjestelma extends Malli {

  private long kokonaisAanimaara = 0;
  private HashMap<String, Long> puolueetJaAanet;

  public Vaalijarjestelma(List<String> puolueet) {
    ...
  }

  // Komponentin julkiset palvelut
  public void muutaAanimaara(String puolue,
    Long aanet) {...}

  public HashMap<String, Long>
    palautaPuolueetJaProsentit() {...}

  public HashMap<String, Long>
    palautaPuolueetJaAanet() {
      ...
  }
}

public class Malli extends Julkaisija {...}
```

Kuva 15: Yksinkertaisen malli-komponentin toteutus.

Kuvassa 16 on esitetty mallikomponenttiin liittyvän muutostenhallintamekanismin toteutus vaalijärjestelmässä. Toteutuksessa on käytetty julkaise-tilaa -suunnittelumallia.

Kuvassa 17 on vaalijärjestelmän näkymäkomponentin rungon toteutus. Näkymäkomponentti toteuttaa tilaajarajapinnan, mikä mahdollistaa sen liittymisen mallin muutostenhallintamekanismiin. Näkymä on tietoinen sekä siihen liittyvästä ohjainkomponentista että sen käyttämästä mallikomponentista. Vaalijärjestelmän konkreettiset näkymät toteuttavat näkymälle ominaiset alustus- ja päivitysrutiinit.

Kuvassa 18 on vaalijärjestelmän ohjainkomponentin rungon toteutus. Näkymäkomponentin tavoin ohjainkomponentti toteuttaa tilaajarajapinnan, mikä mahdollistaa sen

liittymisen mallikomponentin muutostenhallintamekanismiin. Ohjainkomponentin tärkeimmät tehtävät eli syötteiden käsittelyn ja mallin palveluiden kutsun hoitaa kasitteleTapahtuma -metodi.

```
public class Julkaisija {

  private ArrayList<Tilaaja> tilaajat =
    new ArrayList<Tilaaja>();
  public boolean liita(Tilaaja tarkkailija) {
    return this.tilaajat.add(tarkkailija);
  }
  public boolean irroita(Tilaaja tarkkailija) {
    return this.tilaajat.remove(tarkkailija);
  }
  public void ilmoita() {
    for (Tilaaja t : tilaajat) {
     t.paivita(;
    }
  }
}
```

Kuva 16: Vaalijärjestelmän muutostenhallintamekanismin toteutus.

```
public abstract class Nakyma implements Tilaaja {

  protected Malli malli;
  protected Ohjain ohjain;

  protected Nakyma(Malli malli) {
    this.malli = malli;
    this.malli.liita(this);
  }
  abstract public void alusta();
  abstract public void paivita();
}
```

Kuva 17: Vaalijärjestelmän näkymäkomponentin rungon toteutus.

```
public abstract class Ohjain implements
  Tilaaja, ActionListener {

  protected Malli malli;
  protected Nakyma nakyma;

  public Ohjain(Nakyma nakyma) {
    this.nakyma = nakyma;
    this.malli = nakyma.malli;
    this.malli.liita(this);
  }
  public abstract void kasitteleTapahtuma(
    Tapahtuma tapahtuma);

  public void actionPerformed(ActionEvente) {
    kasitteleTapahtuma(new Tapahtuma(e));
  }
}
```

Kuva 18: Vaalijärjestelmän ohjainkomponentti.

4.6 Malli-näkymä-ohjain -arkkitehtuurityylin edut

Malli-näkymä-ohjain -ohjelmistoarkkitehtuurityylissä komponentit ovat selvästi toisistaan eroteltuja, ja niillä kaikilla on hyvin erilaiset vastuut ja tehtävät. Tämä helpottaa järjestelmän ymmärrettävyyttä, koska komponentteja voidaan käsitellä ja ymmärtää ilman sen suurempaa tietoutta muiden komponenttien toiminnasta ja rakenteesta (Krasner & Pope, 1988).

Koska mallikomponentti muodostaa järjestelmän toiminnallisen ytimen ja on erotettu selvästi järjestelmän käyttöliittymästä, voidaan se kehittää ja testata itsenäisesti (Microsoft Corporation, 2008b; Carrington & Hussay, 1995). Toisaalta myös käyttöliittymä voidaan kehittää ja testata itsenäisesti, kunhan vain tiedetään mallikomponentin julkinen rajapinta (Carrington & Hussay, 1995). Käyttöliittymän erottaminen toiminnallisesta ytimestä parantaa myös uudelleenkäytettävyyttä.

Näkymien eriyttäminen mallista mahdollistaa saman mallin esittämisen eri yhteyksissä. Tällöin mallia voidaan myös tarkastella eri näkökulmista. Käyttäjän mahdollisuus

ymmärtää mallin senhetkistä tilaa paranee. Ohjainkomponentin erottaminen mallikomponentista helpottaa syötteiden käsittely- ja ohjaustavan vaihtamista, koska malli- ja näkymä komponentteja ei tarvitse muuttaa (Krasner & Pope, 1988).

Ydintoiminnallisuuden erottamisella käyttöliittymästä mahdollistuu myös käyttöliittymän suorituksen aikana tapahtuva muuttaminen. Suorituksen aikana voidaan lisätä ja vaihtaa niin ohjain- kuin näkymäkomponentteja niin, että edellä kuvatut ominaisuudet saavutetaan dynaamisesti. Tällä tavoin voidaan rakentaa hyvin mukautettavia järjestelmiä. Näkymä- ja ohjainkomponenttien dynaamisen lisäämisen ja vaihtamisen avulla saadaan järjestelmälle luotua vaihdettava ulkoasu ja tuntuma (look and feel).

Muutostenhallintamekanismi mahdollistaa sen, että kaikki näkymät päivittyvät yhdenaikaisesti. Näkymät pysyvät siten koko ajan tietoeheinä mallin suhteen (Buschmann *& al*., 2001).

4.7 Malli-näkymä-ohjain -arkkitehtuurityylin rajoitteet

Vaikka malli-näkymä-ohjain -tyylillä saavutetaan useita hyviä ominaisuuksia, tuo se mukanaan myös joitakin vähemmän haluttuja. Puhtaasti tyylin mukainen toteutus lisää sovelluksen kompleksisuutta, mikä ei aina ole tarpeellista (Buschmann *& al*., 2001).

Näkymä- ja ohjainkomponenttien välillä ei pitäisi olla riippuvuutta, mutta todellisuudessa niiden käyttö yksittäisinä kierrätettävinä komponentteina on vähäistä (Coutaz, 1987). Tästä johtuen vain sellaiset ohjaimet, jotka eivät muokkaa syötteitä, vaan ainoastaan ohjaavat järjestelmälle tulleita tapahtumia, ovat uudelleen käytettäviä ilman suurempia muutoksia.

Koska näkymä- ja ohjainkomponentit tekevät suoria kutsuja mallikomponentille, vaikuttavat mallikomponentin julkiseen rajapintaan kohdistuvat muutokset myös muihin rajapintaa käyttäviin komponentteihin. Tällä voi olla laaja vaikutus järjestelmään, jossa mallikomponenttia käyttää suuri määrä näkymä- ja ohjainkomponentteja (Buschmann *& al*., 2001). Tätä ongelmaa voidaan vähentää käyttämällä käskyoliosuunnittelumallia mallikomponentin palvelujen kutsumiseen (Gamma *& al*., 1995).

Mallin julkisesta rajapinnasta johtuen tietojen päivitys voi olla tehotonta, sillä on mahdollista, että näkymä joutuu tekemään mallille useita kutsuja saadakseen päivitettyä kaikki tarvitsemansa tiedot. Tämä vaikuttaa suorituskykyyn, jos päivityksiä tehdään

usein. Tiedon välivarastoinnilla (cache) tätä ongelmaa voidaan pienentää. Myös mahdollisten muuttumattomien tietojen päivitys voi tuoda mukanaan samankaltaista suorituskykyrasitetta. Näkymä- ja ohjainkomponenttien dynaamisen liitettävyyden takia muutostenhallintamekanismin järjestelmälle aiheuttaman kuorman arvioiminen on vaikeaa. Myös päivitystarpeen havaitseminen on vaikeaa. Esimerkiksi ikonisoitua ikkunaa ei käyttäjän näkökulmasta tarvitse päivittää, vaikka se olisi rekisteröitynyt mallin muutostenhallintamekanismiin (Microsoft Corporation, 2008b).

5 Kerroksittaisen ja malli-näkymä-ohjain -tyylin ominaisuuksien vertailu

Tässä luvussa vertaillaan tutkielmassa käsiteltyjä ohjelmistoarkkitehtuurityylejä. Vertailulla ei pyritä järjestämään tyylejä niiden ominaisuuksien mukaan paremmuusjärjestykseen, vaan sillä pyritään selvittämään, vaikuttaako arkkitehtuurityylin valinta kehitettävän järjestelmän ominaisuuksiin. Tämänkaltaista vertailua voi hyödyntää, kun valitaan arkkitehtuurisia perusratkaisuja.

Ensimmäinen kappale käsittelee yleisesti ohjelmistoarkkitehtuurien ja ohjelmistoarkkitehtuurityylien vertailua ja siihen liittyviä ongelmia. Se myös pohjustaa tässä luvussa käsiteltävää vertailua. Kappaleessa 5.2 esitellään vertailussa käytettävät ohjelmistoarkkitehtuureihin liittyvät ominaisuudet sekä menetelmät ominaisuuksien vertailuun. Kappaleessa 5.3 esitellään vertailun tulokset. Viimeisessä kappaleessa analysoidaan vertailun tuloksia sekä tarkastellaan vertailun järkevyyttä ja hyödyllisyyttä arkkitehtuurityyliä valittaessa.

5.1 Yleiskuva

Arkkitehtuurityylin valintaan vaikuttavat eniten laatuominaisuuksille asetetut vaatimukset (*Bass & al*. 2005). Fieldingin (1996) mukaan arkkitehtuurityylin tulee tukea järjestelmälle asetettuja vaatimuksia, ja tästä syystä arkkitehtuurityylien luokittelun pitäisi perustua arkkitehtuurisiin ominaisuuksiin. Arkkitehtuuriset ominaisuudet voivat olla sekä toiminnallisia että ei-toiminnallisia.

5.2 Menetelmä ja aineisto

Vertailumenetelmä perustuu Fieldingin (1996) tapaan taulukoida arkkitehtuurityylien ominaisuuksia. Siinä ilmaistaan, tuleeko tietty ominaisuus esille tiettyä arkkitehtuurityyliä käytettäessä. Pyrin vertailussa taulukoimaan käsittelemäni arkkitehtuurityylit ja niihin liittyvät ominaisuudet kirjallisuuteen perustuen. Tutkielman lähteistä olen etsinyt käsiteltäviin arkkitehtuurityyleihin liittyviä ominaisuuksia. Käsiteltävät ominaisuudet ovat yhdistelmä Fieldingin (1996) ja *Bass & al.:in*. (2005) käsittelemiä laatuominaisuuksia. Vertailussa käytettävät laatuominaisuudet ovat seuraavat:

- *Laajennettavuus:* Laajennettavuudella tarkoitetaan sitä, kuinka helposti järjestelmän toiminnallisuutta pystytään laajentamaan vaikuttamatta jo olemassa olevaan toiminnallisuuteen. Tämä on hyvin tärkeä ominaisuus, koska nykyisin oletetaan, että järjestelmät pystyvät helposti ja nopeasti vastaamaan esimerkiksi liike-elämän hektisiin tarpeisiin.

- *Luotettavuus:* Ohjelmistoarkkitehtuurin tasolla tarkasteltaessa luotettavuudella tarkoitetaan koko järjestelmän alttiutta häiriöön, jos jossakin järjestelmän osassa tapahtuu virhe. Arkkitehtuurityyli voi vaikuttaa luotettavuuteen esimerkiksi ehkäisemällä yksittäisen vikaantumispisteen esiintymistä (SPOF, single point of failure), monistamalla kriittiset komponentit sekä kiinnittämällä huomiota virheistäpalautumistoimintoihin.

- *Muunneltavuus:* Muunneltavuudella tarkoitetaan arkkitehtuuriin kohdistuvia muutoksia. Yksinkertaistaen voidaan sanoa sen tarkoittavan sitä, kuinka helposti järjestelmään voidaan tehdä muutoksia vaikuttamatta sen olemassa olevaan toiminnallisuuteen. Muunneltavuus ja laajennettavuus ovat hyvin lähellä toisiaan.

- *Siirrettävyys:* Järjestelmä on siirrettävä, jos se pystytään suorittamaan erilaisissa ympäristöissä (Fielding, 2000). Siirrettävyydellä tarkoitetaan sitä, kuinka hyvin erilaisista ympäristöistä johtuvia eroja voidaan vähentää, ja kuinka hyvin järjestelmä voi toimia itsenäisenä komponenttina.

- *Skaalautuvuus:* Skaalautuvuudella tarkoitetaan arkkitehtuurityylin kykyä reagoida suorituksen aikaisten komponenttien ilmentymien lukumäärään sekä komponenttien vuorovaikutuksen intensiteetin vaihteluun (Fielding, 2000).

- *Suorituskyky:* Suorituskyvyllä voidaan eri yhteyksissä tarkoittaa eri asioita. Eri järjestelmissä on erilaisia suorituskykyvaatimuksia, mutta yleisesti voidaan ajatella suorituskyvyn tarkoittavan järjestelmään kohdistuvan laskennan vasteaikaa niin suurella kuin pienelläkin kuormalla.

- *Testattavuus:* Testattavuudella tarkoitetaan sitä, kuinka helposti testausvaatimukset voidaan täyttää. Testattavuuteen vaikuttavat esimerkiksi komponenttien itsenäisyysaste ja riippuvuudet muihin komponentteihin.

- *Turvallisuus:* Turvallisuus tarkoittaa arkkitehtuurin tukea järjestelyille, joilla pyritään varmistamaan tiedon käytettävyys, eheys ja luottamuksellisuus. Käytettävyys tarkoittaa tietoturvan yhteydessä sitä, että tieto on siihen oikeutettujen hyö-

dynnettävissä haluttuna aikana. Eheys tarkoittaa tiedon yhtäpitävyyttä alkuperäisen tiedon kanssa, ja luottamuksellisuus sitä, ettei kukaan sivullinen saa tietoa.

- *Uudelleenkäytettävyys:* Uudelleenkäytettävyys tarkoittaa sitä, kuinka vaivattomasti järjestelmä tai järjestelmän osa voidaan muokata uudeksi järjestelmäksi tai osaksi uutta järjestelmää.

- *Yhteentoimivuus:* Yhteentoimivuus viittaa kykyyn, jossa kaksi tai useampi järjestelmää tai komponenttia siirtää tietoa toisille ja käyttää sitä hyödykseen (IEEE, 1990).

- *Yksinkertaisuus:* Fieldingin (1996) arkkitehtuuriin liittyvä yksinkertaisuus kuvaa sitä, kuinka hyvin arkkitehtuurityyli osittaa toiminnallisuuden selkeiksi ja helposti ymmärrettäviksi komponenteiksi. Tähän ominaisuuteen liittyy läheisesti myös tyylin yleinen kompleksisuus ja ymmärrettävyys. Myös yleiskäyttöisyys kuuluu yksinkertaisuuteen, koska se vähentää tyyliin kohdistuvien muutosten tarvetta.

Lista ei ole täydellinen, sillä siinä ei ole kaikkia ominaisuuksia, jotka järjestelmistä pystytään havaitsemaan, tai joihin arkkitehtuurilla on merkitystä. Fieldingin (1996) ja *Bassin & al.* (2005) kuvaamista ominaisuuksista olen valinnut tähän tapaukseen sopivat. Yksi tärkeä kriteerini oli ottaa mukaan ne ominaisuudet, jotka korostavat verrattavien arkkitehtuurityylien eroja. On huomioitava, että tässä ominaisuuksia tutkitaan pelkästään tyylin "puhtaimman"määrityksen mukaan. Käsiteltävään tyyliin joko liittyy jokin tietty ominaisuus, tai sitten ei. Siihen, miten selvästi ominaisuus tulee esille, ei tässä vertailussa oteta kantaa. Todellisuudessa ominaisuuden liittyminen arkkitehtuurityyliin ei ole näin jyrkkä. Arkkitehtuuriset ominaisuudet ovat hyvin alttiita muutoksille. Arkkitehtuurityyliä muokkaamalla ominaisuuksia voidaan lisätä, vähentää, korostaa tai peittää.

Kerroksittaista arkkitehtuurityyliä varten ei aineistoa ole erikseen tarvinnut kerätä, koska kerroksittainen arkkitehtuurityyli on mukana Fieldingin (1996) vertailussa. Sen sijaan malli-näkymä-ohjain -arkkitehtuurityylistä ei tällaista vertailua tai aineistoa löytynyt, joten vertailuni aineisto on kerätty tutkielman lähteenä olevasta aineistosta.

5.3 Tulokset ja tulosten tarkastelu

Kerroksittaiseen ja malli-näkymä-ohjain -ohjelmistoarkkitehtuurityyleihin liittyvät ominaisuudet on koottu taulukkoon 2. Taulukon ensimmäisessä sarakkeessa on ominaisuuden nimi, toisessa sarakkeessa kerroksittaiseen tyyliin liittyvät ominaisuudet, ja kolmannessa sarakkeessa malli-näkymä-ohjain -tyyliin liittyvät ominaisuudet. Jos tyyliin liittyy jokin tietty ominaisuus, on ominaisuutta ilmaisevalla rivillä pukki tyylin sarakkeessa.

Taulukko 2: Arkkitehtuurityylien vertailun tulokset.

Ominaisuus	*Kerroksittainen tyyli*	*malli-näkymä-ohjain*
laajennettavuus		✓
luotettavuus		
muunneltavuus	✓	✓
siirrettävyys	✓	✓
skaalautuvuus	✓	
suorituskyky		
testattavuus	✓	✓
turvallisuus		
uudelleenkäytettävyys		✓
yhteentoimivuus		
yksinkertaisuus		

Kuten kappaleessa 5.2 on mainittu, kerroksittaista tyyliä koskeva aineisto on Fieldingin (1996) tutkimuksesta, mutta malli-näkymä-ohjain -tyyliin liittyvä aineisto on kerätty seuraavista lähdeaineistoista: Apple (2007), Burbeck (1992), Buschmann *& al*. (2001), Chow *& al*. (1996), Krasner & Pope (1988), Microsoft (2008b), Reenskaug (2003) ja Reenskaug (1979). Kahdesta lähteestä, Reenskaug (2003) sekä Reenskaug (1979), en löytänyt yhtään viittausta tyyliin liittyviin arkkitehtuurisiin ominaisuuksiin. Muista aineistoista viittauksia löytyi, joskaan ei aina kovin selkeästi. Seuraavaksi käyn läpi jokaisen ominaisuuden erikseen ja selvitän, kuinka taulukon 2 tuloksiin on päädytty.

Laajennettavuus tuli esille yhdestä lähteestä. Applen (2007) mukaan malli-näkymä-ohjain -tyyli on laajennettava, koska komponenttien välinen selkeä jaottelu lisää komponentin kierrätettävyyttä ja järjestelmän laajennettavuutta. Järjestelmät ovat kaiken

kaikkiaan mukautuvampia muuttuviin vaatimuksiin, toisin sanoen ne ovat paljon helpommin laajennettavia kuin järjestelmät, jotka eivät perustu tähän tyyliin.

Fieldingin (1996) tutkimuksen mukaan kerroksittaisella tyylillä ei ole luotettavuuden suhteen positiivista eikä negatiivista vaikutusta. Myöskään malli-näkymä-ohjain -tyylin ja luotettavuuden välille ei löytynyt yhtymäkohtia.

Fieldingin (1996) mukaan kerroksittaisen tyylin käyttö lisää järjestelmän muunneltavuutta. Muunneltavuus tulee esille myös malli-näkymä-ohjain -tyyliä käytettäessä. Burbeckin (1992) mukaan malli-näkymä-ohjain -tyyli edesauttaa toiminnallisuuden periyttämistä, lisäämistä ja mukauttamista, jolloin muunneltavan ja tehokkaan järjestelmän toteuttaminen mahdollistuu.

Fielding (1996) on tutkimustuloksiinsa merkinnyt, että kerroksittaisella tyylillä on positiivinen vaikutus siirrettävyyteen, mutta ei ilmaise kuinka se tulee ilmi. Fielding (1996) kyllä mainitsee, että kerroksittaista tyyliä käytetään paljon laitteistorajapintakirjastoissa. Buschmannin (2001) mukaan malli-näkymä-ohjain -arkkitehtuurityyli parantaa siirrettävyyttä, koska malli ei ole riippuvainen käyttöliittymästä. Järjestelmän siirto uuteen suoritusympäristöön ei siis vaadi muutoksia mallikomponenttiin eli toiminnalliseen ytimeen.

Fieldingin (1996) mukaan kerroksittaisella arkkitehtuurityylillä on positiivinen vaikutus järjestelmän skaalautuvuuteen, mutta hän ei ilmaise tälle selvää syytä. Myöskään malli-näkymä-ohjain -tyylin ja skaalautuvuuden välille ei löytynyt mitään selkeää yhteyttä.

Fieldingin (1996) mukaan kerroksittaisella tyylillä on positiivinen vaikutus testattavuuteen, mutta ei selitä kuinka se tulee esille. Fieldingin (1996) tutkimuksessa mainitaan, että tyyli vähentää komponenttien välisiä riippuvuuksia. Oletan, että tämä lause sisältää epäsuoran viittauksen testattavuuteen. Microsoftin (2008b) mukaan käyttöliittymätestauksen automatisointi on huomattavasti vaikeampaa ja hitaampaa kuin toiminnallisen ytimen testauksen automatisointi. Malli-näkymä-ohjain -arkkitehtuurityyli minimoi käyttöliittymän riippuvuuden parantaen järjestelmän testattavuutta.

Buschmannin (2001) mukaan ohjelmistokehikko voidaan toteuttaa malli-näkymä-ohjain -arkkitehtuurityyliä käyttäen. Koska ohjelmistokehikon yksi käyttötarkoitus on mahdollistaa kehikon uudelleenkäyttö, voidaan sanoa, että malli-näkymä-ohjain -arkkitehtuurityyli parantaa mahdollisuutta järjestelmän uudelleenkäytölle.

Fieldingin (1996) tutkimuksen mukaan kerroksittaisella tyylillä ei suorituskyvyn, turvallisuuden, yhteentoimivuuden ja yksinkertaisuuden suhteen ole positiivista tai negatiivista vaikutusta. Edellä mainitut arkkitehtuuriset ominaisuudet eivät myöskään tulleet esille malli-näkymä-ohjain -tyyliä käsittelevästä aineistosta.

Taulukosta 2 voidaan nähdä, että kummaltakin verrattavalta tyyliltä löytyi etsittyjä ominaisuuksia, ja että tyylit näiden ominaisuuksien suhteen eroavat vain laajennettavuuden, skaalautuvuuden ja uudelleenkäytettävyyden suhteen. Malli-näkymä-ohjain -tyylin käyttö tuo mukanaan laajennettavuuden ja uudelleenkäytettävyyden, kun taas kerroksittainen tyyli on skaalautuvampi.

Kahdeksan ominaisuutta yhdestätoista on samoja kummassakin tyylissä, joten voidaan kai sanoa, että ainakin näiden ominaisuuksien valossa tyylit ovat hyvin lähellä toisiaan, vaikka tyylit kuvausten määritelmien suhteen vaikuttavat hyvinkin erilaisilta. Tämän perusteella taas voidaan sanoa, että jos halutaan järjestelmä, joka on muunneltava, siirrettävä ja testattava, ei tyylin valinnalla ole varsinaisesti merkitystä. Tietenkin silloin, kun malli-näkymä-ohjain -tyyli valitaan, tulee mukana vielä joukko muitakin ominaisuuksia. Voidaan siis olettaa, että kerroksittaisen tyylin käyttö on vähäisempää kuin malli-näkymä-ohjain -tyylin.

5.4 Johtopäätökset

Tämän luvun käsittelemän pienimuotoisen tutkimuksen tavoitteena oli luoda kuva siitä, miten kaksi eri ohjelmistoarkkitehtuurityyliä eroavat toisistaan ohjelmistoarkkitehtuuristen ominaisuuksien suhteen. Vertailussa käytettiin joukkoa niin toiminnallisia kuin ei-toiminnallisiakin ominaisuuksia ja selvitettiin, liittyvätkö ominaisuudet käsiteltäviin tyyleihin.

Tutkimus osoitti sen, että kirjallisuuteen perustuvalla poiminnalla arkkitehtuurityylien väliltä löytyy kyllä eroja, mutta jos otetaan huomioon kaikki tutkielmassa mainitut järjestelmät, olisin odottanut vielä suurempaa eroavaisuutta. Erojen vähäisyys saattaa johtua hieman suppeasta verrattavien ominaisuuksien määrästä, mutta myös siitä, että kirjallisuudesta on todella vaikea löytää tyyleihin liittyviä ominaisuuksia, koska niitä ei yleensä mainita, vaan ne pitäisi ymmärtää ja johtaa eri asiayhteyksistä. Myös ominaisuuksien määrittely tarkasti ja yksiselitteisesti tässä yhteydessä on vaikeaa. Voidaankin todeta, että kattavamman vertailun aikaansaamiseksi tulisi tyyleillä toteutettuja olemas-

sa olevia järjestelmiä tarkastella vain etsittäviin ominaisuuksiin keskittyen.

Kehitettävälle järjestelmälle parhaan arkkitehtuurin valinta on vaikea tehtävä, eikä pelkästään toiminnallisiin ja ei-toiminnallisiin ominaisuuksiin perustuva valinta johda välttämättä parhaaseen mahdolliseen lopputulokseen. On kuitenkin huomioitava, että tämän tutkimuksen kaltainen luettelointi ja vertailu voi valittavien arkkitehtuurityylien määrää rajaamalla helpottaa arkkitehtuurin valintaa.

Suunnittelumalleja ja tietorakenteita on tutkittu paljon, ja niiden käyttö nykypäivänä on melko suoraviivaista. Esimerkiksi Gamma *& al.*:n (1995) suunnittelumallien luettelosta voidaan suunnitteluongelmaan hakea ratkaisua ongelman perusteella. Vastaavaa luettelointia kaipaisivat myös arkkitehtuuriset perusratkaisut. Se voisivatko arkkitehtuuriset ominaisuudet olla arkkitehtuurityylien luetteloinnin perustana, ei tullut tässä tutkimuksessa esille.

6 Yhteenveto

Tässä tutkielmassa on käsitelty yleisellä tasolla ohjelmistoarkkitehtuuria ja sen liittymäkohtia ohjelmistotuotantoon. Tutkielmassa esitetään, miten erilaisia ohjelmistoarkkitehtuureja on luokiteltu ohjelmistoarkkitehtuurityyleihin. Tutkielman pääpaino on kerroksittaisen sekä malli-näkymä-ohjain -ohjelmistoarkkitehtuurityylin teoreettisessa tarkastelussa. Tutkielmassa määritellään, mitä tarkoitetaan kerroksittaisella ja mitä malli-näkymä-ohjain -ohjelmistoarkkitehtuurityylillä, miten niitä voidaan käyttää, ja mitä niiden käytöllä voidaan saavuttaa.

Kerroksittaista ohjelmistoarkkitehtuurityyliä voitaisiin kuvata näin: yksinkertainen, mutta kuitenkin monipuolinen. Kerroksittainen tyyli voi olla mainio valinta, jos uudelleenkäytettävyys ja siirrettävyys ovat tärkeitä vaatimuksia kehitettävälle ohjelmistolle. Se voi kuitenkin johtaa myös tehottomaan ja huonoon lopputulokseen, jos sitä käytetään tilanteessa, johon se ei sovellu. Kerroksittaisen arkkitehtuurityylin historia osoittaa sen, että oikein käytetyn kerroksittaisen tyylin avulla voidaan saavuttaa laadukkaita ohjelmistoja, joista on hyötyä useiksi vuosiksi.

Malli-näkymä-ohjain -ohjelmistoarkkitehtuurityyli on kerroksittaista tyyliä hiukan hankalampi ymmärtää, mutta toisaalta siinä on komponenttien vastuut ja tehtävät hyvin selkeästi määritelty. Malli-näkymä-ohjain -tyyli on käyttövalmiimpi kuin kerroksittainen tyyli, mutta tästä johtuen myös sen käyttömahdollisuudet ovat rajallisemmat. Tyylin pääasiallisena käyttökohteena ovatkin graafista käyttöliittymää hyödyntävät järjestelmät.

Ohjelmistoarkkitehdin kannalta eri ohjelmistoarkkitehtuurityylien tuntemus on tärkeää, koska samaan ongelmaan voi olla useita soveltuvia arkkitehtuuriratkaisuja. Sovittamalla useita eri tyylejä järjestelmän arkkitehtuuriratkaisuksi saadaan aikaan paremmat perustelut sille, miksi arkkitehtuurin kiinnittämisen jälkeen on päädytty juuri valittuun ratkaisuun. Vaikka oma henkilökohtainen kokemukseni ohjelmistotuotannon alalta on vielä vähäistä, olen usein törmännyt siihen, että järjestelmän arkkitehtuuria kritisoidaan perustein, jotka eivät oikeastaan edes liity itse arkkitehtuuriin. On muistettava, että edes täydellinen arkkitehtuurityylin valinta ei itsessään ole laadun tae, myös järjestelmän toteutuksen täytyy olla laadukkaasti tehty.

Tässä tutkielmassa eri arkkitehtuurityylejä käsitellessäni minulle selvisi se, kuinka vaikeaa eri arkkitehtuurityylejä on verrata keskenään. Ohjelmistoarkkitehtuuri on - aina-

kin tällä hetkellä - niin abstrakti käsite, että sen vertailu formaalein metodein on vaikeaa. Ehkä tulevaisuudessa voidaan järjestelmälle paras mahdollinen arkkitehtuurityyli valita järjestelmälle kohdistuvien toiminnallisten ja ei-toiminnallisten vaatimusten mukaan, ja sitä kautta parantaa järjestelmien laatua.

Viitteet

Apple Inci. (2007) *Cocoa Fundamentals Guide* http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaFundamentals.pdf (5.3.2008).

Bachmann, F., Bass, L., Carriere, J., Clements, P., Garlan, D., Ivers, I., Nord, R., Little, R. (2000) *Documentation in Practice: Documenting Architectural Layers*. CMU/SEI-2000-SR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

Bachmann, F., Bass, L., Clements P., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J. (2002) *Documenting Software Architecture: Documenting Interfaces*. CMU/SEI-2002-TN-015, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

Bass, L., Clements, P., Kazman, R. (1998) *Software Architecture in Practice*. Addison Wesley Longman, inc., Reading, Massachusetts.

Bass, L., O'Brien, L., Merson, P. (2005) *Quality Attributes and Service-Oriented Architecture*. CMU/SEI-2005-TN-014, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

Booch, G., Jacobson, I., Rumbaugh, J. (1999) *The Unified Modelling Language User Guide*. Addison Wesley Longman, inc., USA.

Bowman, I. (1998) *Conceptual Architecture of the Linux Kernel* http://www.stillhq.com/pdfdb/000524/data.pdf (4.6.2008).

Bredemeyer, D., Malan, R. (2000) *The Role of the Software Architect* http://www.bredemeyer.com/pdf_files/role.pdf (4.6.2008).

Burbeck, S. (1992) *Applications Programming in Smalltalk-80: How to Use Model-View-Cotroller* http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html (5.3.3008).

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Sommerlad, P., Stal, M. (2001) *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns* John Wiley & Sons, England.

Carrington, D., Hussey A. (1995) Comparing Two User-Interface Architectures: MVC and PAC. *Formal Aspects of the Human Computer Interface, FAHCI'96* 95-33, Springer Verlag, Australia.

Chow, P., Tsang, W., Tong, S. (1996) *The Smalltalk MVC Paradigm with Pluggable Views* `citeseer.ist.psu.edu/597439.html` (7.3.2008).

Coutaz, J., (1987) The Construction of User Interfaces and the Object Paradigm. *European Conference on Object-oriented Programming* (toim. Bézivin, J. & *al*.), Springer-Verlag, London, 121-130.

Dijkstra, E. (1968) The Structure of "THE"-Multiprogramming System. *Communication of the ACM* **11**(5), 341-346.

Fielding, R. (2000) *Architectural Styles and the Design of Network-Based Software*. University of California, Information and Computer Science, California.

Gamma, E., Helm, R, Johnson, R., Vlissides, J., (1995) *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley Professional.

Garlan, D. (2000) Software Architecture: A Roadmap. *ICSE - Future of SE Track* (toim. Finkelstein, A.), ACM Press, New York, 91-101.

Garlan, D., Shaw, M. (1998) *Software Architecture: Perspective on an Emerging Discipline*. Prentice-Hall, Inc, Upper Saddle River, New Jersey.

Garlan, D., Shaw, M. (1994) An Introduction to Software Architecture. *Advances in Software Engineering and Knowledge Engineering* (toim. Ambriola, V., Tortora, G.), World Scientific Publishing Company, Singapore, 1-39.

IEEE (1990) *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* Institute of Electrical and Electronics Engineers. New York, NY.

Krasner, G., Pope, S. (1988) A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *Journal of Object-Oriented Programming* **1**(3), 26-49.

Microsoft Corporation (2008a) *Layered Application* WWW-sivusto `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/ArcLayeredApplication.asp` (21.2.2008).

Microsoft Corporation (2008b) *Model-View-Controller* WWW-sivusto, http://msdn2.microsoft.com/en-us/library/ms978748.aspx (21.2.2008).

Reenskaug, T. (1979) *Models-Views-Controllers* http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf (15.2.2008).

Shaw, M. (1996) *Some Patterns for Software Architectures* Proceedings of the Second Pattern Languages of Program Design Workshop, Addison-Wesley, Pennsylvania.

The Software Engineering Institute (2004) *How Do You Define Software Architecture?* WWW-sivusto, http://www.sei.cmu.edu/architecture/definitions.html (3.12005).

Zimmermann, H. (1980) OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, **28**(4), 425-432.

Liite 1: Kerroksittaisella tyylillä toteutetun vaalijärjestelmän lähdekoodi

Tämä liite sisältää yksinkertaisen kerroksittaisella ohjelmistoarkkitehtuurilla toteutetun vaalijärjestelmän lähdekoodin. Lähdekoodin kääntämiseen tarvitaan perustiedot Java-kielisen ohjelman kääntämisestä ja suorittamisesta sekä asianmukainen ympäristö ohjelmineen.

```
package fi.rapa.gradu.kerroksittainen;

import java.awt.event.ActionListener;
import java.util.List;

interface EsityskerrosR extends ActionListener {
  public void alustaIkkunat();
  public void lisaaPuolueet(List<String> puolueet);
}

package fi.rapa.gradu.kerroksittainen;

import java.awt.Component;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.util.List;
import java.util.Map;
import javax.swing.InputVerifier;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class Esityskerros implements EsityskerrosR {

  private LogiikkakerrosR logiikkakerros = null;
  private JTextArea p_prosentit = null;
  private JFrame p_kehys = null;
  private JFrame kehys = null;
  private JTextArea pylvaat = null;

  public Esityskerros(LogiikkakerrosR logiikkakerros) {
    this.logiikkakerros = logiikkakerros;
  }

  public void alustaProsentit() {
    this.p_kehys = new JFrame("Prosentti");
    p_kehys.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    p_kehys.setLocationByPlatform(true);
    this.p_prosentit = new JTextArea(5, 25);
    p_kehys.getContentPane().add(p_prosentit);
    p_kehys.pack();
    p_kehys.setVisible(true);
  }

  public void piirraProsentit() {
    this.p_prosentit.setText(
    this.logiikkakerros.palautaProsentit());
    this.p_kehys.repaint();
  }
```

```
private void alustaPylvaat() {
  kehys = new JFrame("Pylväsdiagrammi");
  kehys.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  this.pylvaat = new JTextArea(10, 50);
  kehys.getContentPane().add(this.pylvaat);
  kehys.setLocationByPlatform(true);
  piirraPylvaat();
  kehys.pack();
  kehys.setVisible(true);
}

public void piirraPylvaat() {
  this.pylvaat.setText(logiikkakerros.palautaPylvaat());
  this.kehys.repaint();
}

private void alustaSyotto() {

  Map<String, String> puolueetJaAanet =
    this.logiikkakerros.palautaAanimaarat();
  JFrame frame = new JFrame("Syöttö");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JPanel paneeli = new JPanel(
    new GridLayout(puolueetJaAanet.size(), 2));
  for (String puolue : puolueetJaAanet.keySet()) {
    JLabel label = new JLabel(puolue);
    JTextField textField = new JTextField(10);
    textField.setInputVerifier(new InputVerifier() {

      @Override
      public boolean verify(JComponent kentta) {
        try {
          Long.parseLong(((JTextField) kentta).getText().trim());
          return true;
        } catch (Exception ignore) {}
        return false;
      }
    });
    textField.setName(puolue);
    textField.setText(puolueetJaAanet.get(puolue).toString());
    textField.addActionListener(this);
    paneeli.add(label);
    paneeli.add(textField);
  }
  frame.add(paneeli);

  frame.pack();
  frame.setVisible(true);
}

public void alustaIkkunat() {
  alustaSyotto();
  alustaProsentit();
  alustaPylvaat();
}

public void actionPerformed(ActionEvent e) {
  String nimi = ((Component) e.getSource()).getName();
  String arvo = e.getActionCommand();
  this.logiikkakerros.muutaAanimaara(nimi, arvo);
  paivitaNaytot();
}

private void paivitaNaytot() {
  piirraProsentit();
  piirraPylvaat();
}
```

```
  public void lisaaPuolueet(List<String> puolueet) {
    this.logiikkakerros.alustaVaalit(puolueet);
  }
}

package fi.rapa.gradu.kerroksittainen;

import java.util.List;
import java.util.Map;

interface LogiikkakerrosR {

  public void alustaVaalit(List<String> puolueet);
  public void muutaAanimaara(
    String puolue, String aanimaara);
  public Map<String, String> palautaAanimaarat();
  public String palautaPylvaat();
  public String palautaProsentit();
}

package fi.rapa.gradu.kerroksittainen;

import java.util.Hashtable;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class Logiikkakerros implements LogiikkakerrosR {

  private TietokerrosR alempiKerros = null;

  public Logiikkakerros(TietokerrosR alempiKerros) {
    this.alempiKerros = alempiKerros;
  }

  public String palautaProsentit() {
    Map<String, String> aanimaarat = palautaAanimaarat();

    float kokonaisaanimaara = kokonaisaanimaara();
    StringBuilder prosentit = new StringBuilder();
    short tmp = 0;
    for (String puolue : aanimaarat.keySet()) {
      if (kokonaisaanimaara == 0) {
        tmp = 0;
      } else {
        float aanimaara = new Float(aanimaarat.get(puolue));
        tmp = (short) (Math.round(
          (aanimaara / kokonaisaanimaara) * 100));
      }
      prosentit.append(puolue + " " + tmp + "%\n");
    }
    return prosentit.toString();
  }

  public String palautaPylvaat() {
    Set<String> puolueet = alempiKerros.haeKaikkiAvaimet();
    if (puolueet != null && !puolueet.isEmpty()) {
      StringBuilder pylvaat = new StringBuilder();
      Long aanimaara = null;
      for (String puolue : puolueet) {
        aanimaara = alempiKerros.hae(puolue);
        if (aanimaara != null) {
          pylvaat.append(puolue + "\n" +
            muodostaPylvas(aanimaara) + "\n");
        }
      }
      return pylvaat.toString();
    }
```

```
  return "";
}

public void muutaAanimaara(String puolue, String arvo) {
  if (puolue == null || arvo == null) {
    return;
  }
  Long aanet = null;
  try {
    aanet = new Long(arvo);
  } catch (NumberFormatException nfe) {
  }
  if (aanet != null && aanet.longValue() > 0) {
    alempiKerros.tallenna(puolue, aanet);
  }
}

public void alustaVaalit(List<String> puolueet) {
  if (puolueet == null) {
    return;
  }
  for (String puolue : puolueet) {
    if (puolue == null) {
      continue;
    }
    alempiKerros.tallenna(puolue, 0L);
  }
}

public Map<String, String> palautaAanimaarat() {
  Set<String> puolueet = alempiKerros.haeKaikkiAvaimet();
  if (puolueet != null && !puolueet.isEmpty()) {
    Map<String, String> puolueidenAanet =
      new Hashtable<String, String>();
    Long aanimaara = null;
    for (String puolue : puolueet) {
      aanimaara = alempiKerros.hae(puolue);
      if (aanimaara != null) {
        puolueidenAanet.put(puolue, aanimaara.toString());
      }
    }
    return puolueidenAanet;
  }
  return null;
}

private String muodostaPylvas(Long koko) {
  if (koko == null || koko.longValue() < 1) {
    return "";
  }
  StringBuilder pylvas = new StringBuilder();
  long laskuri = koko.longValue();
  while (laskuri-- > 0) {
    pylvas.append("*");
  }
  return pylvas.toString();
}

private float kokonaisaanimaara() {
  float kokonaisaanimaara = 0;
  Set<String> puolueet = alempiKerros.haeKaikkiAvaimet();
  if (puolueet != null && !puolueet.isEmpty()) {
    Map<String, String> puolueidenAanet =
      new Hashtable<String, String>();
    Long aanimaara = null;
    for (String puolue : puolueet) {
      aanimaara = alempiKerros.hae(puolue);
      if (aanimaara != null) {
        kokonaisaanimaara += aanimaara;
```

```
        }
      }
    }
    return kokonaisaanimaara;
  }
}

package fi.rapa.gradu.kerroksittainen;

import java.util.Set;

interface TietokerrosR {
  public Long hae(String avain);
  public Set<String> haeKaikkiAvaimet();
  public boolean tallenna(String avain, Long arvo);
}

package fi.rapa.gradu.kerroksittainen;

import java.util.Hashtable;
import java.util.Set;

public class Tietokerros implements TietokerrosR {

  private Hashtable<String, Long> puolueetJaAanet;

  Tietokerros() {
    this.puolueetJaAanet = new Hashtable<String, Long>();
  }

  public Long hae(String avain) {
    return puolueetJaAanet.get(avain);
  }

  public Set<String> haeKaikkiAvaimet() {
    return puolueetJaAanet.keySet();
  }

  public boolean tallenna(String avain, Long arvo) {
    if (avain == null || arvo == null) {
      return false;
    }
    puolueetJaAanet.put(avain, arvo);
    return true;
  }
}
```

Liite 2: Malli-näkymä-ohjain -tyylillä toteutetun vaalijärjestelmän lähdekoodi

Tämä liite sisältää yksinkertaisen malli-näkymä-ohjain -ohjelmistoarkkitehtuurilla toteutetun vaalijärjestelmän lähdekoodin. Lähdekoodin kääntämiseen tarvitaan perustiedot Java-kielisen ohjelman kääntämisestä ja suorittamisesta sekä asianmukainen ympäristö ohjelmineen.

```
package fi.rapa.gradu.mvc;

public class Malli extends Julkaisija {}


package fi.rapa.gradu.mvc;

public abstract class Nakyma implements Tilaaja {

    protected Malli malli;
    protected Ohjain ohjain;

    protected Nakyma(Malli malli) {
        this.malli = malli;
        this.malli.liita(this);
    }
    abstract public void alusta();
    abstract public void paivita();
}


package fi.rapa.gradu.mvc;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public abstract class Ohjain implements Tilaaja, ActionListener {

    protected Malli malli;
    protected Nakyma nakyma;

    public Ohjain(Nakyma nakyma) {
        this.nakyma = nakyma;
        this.malli = nakyma.malli;
        this.malli.liita(this);
    }
    public abstract void kasitteleTapahtuma(Tapahtuma tapahtuma);

    public void actionPerformed(ActionEvent e) {
        kasitteleTapahtuma(new Tapahtuma(e));
    }
}


package fi.rapa.gradu.mvc;

import java.util.ArrayList;

public class Julkaisija {

    private ArrayList<Tilaaja> tilaajat = new ArrayList<Tilaaja>();

    public boolean liita(Tilaaja tarkkailija) {
```

```
        return this.tilaajat.add(tarkkailija);
    }
    public boolean irroita(Tilaaja tarkkailija) {
        return this.tilaajat.remove(tarkkailija);
    }
    public void ilmoita() {
        for (Tilaaja t : tilaajat) {
            t.paivita();
        }
    }
}


package fi.rapa.gradu.mvc;

interface Tilaaja {
    void paivita();
}


package fi.rapa.gradu.mvc;

import java.awt.Component;
import java.awt.event.ActionEvent;

public class Tapahtuma {
    private String nimi = null;
    private String arvo = null;

    public Tapahtuma(ActionEvent e) {
        this.nimi = ((Component)e.getSource()).getName();
        this.arvo = e.getActionCommand();
    }
    public String palautaNimi() {
        return this.nimi;
    }
    public String palautaArvo() {
        return this.arvo;
    }
}


package fi.rapa.gradu.vaalijarjestelma;

import java.util.ArrayList;
import fi.rapa.gradu.mvc.Malli;
import fi.rapa.gradu.mvc.Nakyma;
import fi.rapa.gradu.vaalijarjestelma.malli.Vaalijarjestelma;
import fi.rapa.gradu.vaalijarjestelma.nakymat.ProsenttiNakyma;
import fi.rapa.gradu.vaalijarjestelma.nakymat.PylvasNakyma;
import fi.rapa.gradu.vaalijarjestelma.nakymat.SyottoNakyma;

public class Main {

    public static void main(String[] args) {
        ArrayList<String> puolueet = new ArrayList<String>(4);
        puolueet.add("Punainen");
        puolueet.add("Sininen");
        puolueet.add("Vihreä");
        Malli vaalijarjestelma = new Vaalijarjestelma(puolueet);

        Nakyma pn = new PylvasNakyma(vaalijarjestelma);
        pn.alusta();

        Nakyma prn = new ProsenttiNakyma(vaalijarjestelma);
        prn.alusta();

        Nakyma sn = new SyottoNakyma(vaalijarjestelma);
        sn.alusta();
```

```
    }
}


package fi.rapa.gradu.vaalijarjestelma.malli;

import java.util.HashMap;
import java.util.List;
import fi.rapa.gradu.mvc.Malli;

public class Vaalijarjestelma extends Malli {

    private long kokonaisAanimaara = 0;
    private HashMap<String, Long> puolueetJaAanet;

    public Vaalijarjestelma(List<String> puolueet) {
        puolueetJaAanet = new HashMap<String, Long>();
        for (String p : puolueet) {
            puolueetJaAanet.put(p, 0L);
        }
    }

    public void muutaAanimaara(String puolue, Long aanet) {
        if (this.puolueetJaAanet.containsKey(puolue) &&
            aanet != null && aanet.longValue() > 0) {
            kokonaisAanimaara = kokonaisAanimaara -
                (this.puolueetJaAanet.get(puolue)) + aanet.longValue();
            this.puolueetJaAanet.put(puolue, aanet);
            ilmoita();
        }
    }

    public HashMap<String, Long> palautaPuolueetJaProsentit() {
        HashMap<String, Long> puolueetJaProsentit =
            new Hash<String, Long>();
        for (String puolue: puolueetJaAanet.keySet()) {
            if (kokonaisAanimaara == 0) {
                puolueetJaProsentit.put(puolue, 0L);
            } else {
                puolueetJaProsentit.put(puolue,
                    (long)( ((float) puolueetJaAanet.get(puolue) /
                                    kokonaisAanimaara) * 100));
            }
        }
        return puolueetJaProsentit;
    }

    public HashMap<String, Long> palautaPuolueetJaAanet() {
        return puolueetJaAanet;
    }
}


package fi.rapa.gradu.vaalijarjestelma.nakymat;

import java.util.HashMap;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import fi.rapa.gradu.mvc.Malli;
import fi.rapa.gradu.mvc.Nakyma;
import fi.rapa.gradu.vaalijarjestelma.malli.Vaalijarjestelma;

public class ProsenttiNakyma extends Nakyma {

    private JFrame kehys = null;
    private JTextArea prosentit = null;

    public ProsenttiNakyma(Malli malli) {
        super(malli);
```

```
    }

    public void paivita() {
        piirra();
    }

    public void alusta() {
        kehys = new JFrame("Prosentti");
        kehys.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
        kehys.setLocationByPlatform(true);
        this.prosentit = new JTextArea(10, 50);
        kehys.getContentPane().add(this.prosentit);
        piirra();
        kehys.pack();
        kehys.setVisible(true);
    }

    public void piirra() {
        if (!this.malli instanceof Vaalijarjestelma) {
            return;
        }
        HashMap<String, Long> puolueetJaProsentit =
         ((Vaalijarjestelma)this.malli).palautaPuolueetJaProsentit();
        StringBuilder pylvaat = new StringBuilder();
        for (String puolue : puolueetJaProsentit.keySet()) {
            pylvaat.append(puolue + " " +
                puolueetJaProsentit.get(puolue) + "%\n");
        }
        this.prosentit.setText(pylvaat.toString());
        this.kehys.repaint();
    }
}


package fi.rapa.gradu.vaalijarjestelma.nakymat;

import java.util.HashMap;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import fi.rapa.gradu.mvc.Malli;
import fi.rapa.gradu.mvc.Nakyma;
import fi.rapa.gradu.vaalijarjestelma.malli.Vaalijarjestelma;

public class PylvasNakyma extends Nakyma {
    private JFrame kehys = null;

    private JTextArea pylvaat = null;

    public PylvasNakyma(Malli malli) {
        super(malli);
    }

    public void paivita() {
        piirra();
    }

    public void alusta() {
        kehys = new JFrame("Pylväsdiagrammi");
        kehys.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        this.pylvaat = new JTextArea(10, 50);
        kehys.getContentPane().add(this.pylvaat);
        kehys.setLocationByPlatform(true);
        piirra();
        kehys.pack();
        kehys.setVisible(true);
    }
```

```
    public void piirra() {
        if (!this.malli instanceof Vaalijarjestelma) {
            return;
        }
        HashMap<String, Long> puolueetJaProsentit =
         ((Vaalijarjestelma)this.malli).palautaPuolueetJaProsentit();
        StringBuilder pylvaat = new StringBuilder();
        for (String puolue : puolueetJaProsentit.keySet()) {
            pylvaat.append(puolue + "\n" +
             muodostaPylvas(puolueetJaProsentit.get(puolue)) + "\n");
        }
        this.pylvaat.setText(pylvaat.toString());
        this.kehys.repaint();
    }

    private String muodostaPylvas(Long koko) {
        if (koko == null || koko.longValue() < 1) return "";
        StringBuilder pylvas = new StringBuilder();
        long laskuri = koko.longValue();
        while (laskuri-- > 0 ) {
            pylvas.append("*");
        }
        return pylvas.toString();
    }
}


package fi.rapa.gradu.vaalijarjestelma.nakymat;

import java.awt.GridLayout;
import java.util.HashMap;
import javax.swing.InputVerifier;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import fi.rapa.gradu.mvc.Malli;
import fi.rapa.gradu.mvc.Nakyma;
import fi.rapa.gradu.vaalijarjestelma.malli.Vaalijarjestelma;
import fi.rapa.gradu.vaalijarjestelma.ohjaimet.SyottoOhjain;

public class SyottoNakyma extends Nakyma {

    public SyottoNakyma(Malli malli) {
        super(malli);
        this.ohjain = new SyottoOhjain(this);
    }

    @Override
    public void alusta() {
        if (this.malli instanceof Vaalijarjestelma) {
            HashMap<String, Long> puolueetJaAanet =
                ((Vaalijarjestelma)this.malli).palautaPuolueetJaAanet();
            JFrame frame = new JFrame("Syöttö");
            frame.setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);
            JPanel paneeli =
                new JPanel(
                 new GridLayout(puolueetJaAanet.size(),2));
            for (String puolue: puolueetJaAanet.keySet()) {
                JLabel label = new JLabel(puolue);
                JTextField textField = new JTextField(20);
                textField.setInputVerifier(new InputVerifier() {
                    @Override
            //Display the window.
            frame.pack();
            frame.setVisible(true);
```

```
        }
    }

    @Override
    public void paivita() {}
}


package fi.rapa.gradu.vaalijarjestelma.ohjaimet;

import fi.rapa.gradu.mvc.Nakyma;
import fi.rapa.gradu.mvc.Ohjain;
import fi.rapa.gradu.mvc.Tapahtuma;

public class ProsenttiOhjain extends Ohjain {

    public ProsenttiOhjain(Nakyma nakyma) {
        super(nakyma);
    }

    public void kasitteleTapahtuma(Tapahtuma tapahtuma) {}

    @Override
    public void paivita() {}

}


package fi.rapa.gradu.vaalijarjestelma.ohjaimet;

import fi.rapa.gradu.mvc.Nakyma;
import fi.rapa.gradu.mvc.Ohjain;
import fi.rapa.gradu.mvc.Tapahtuma;

public class PylvasOhjain extends Ohjain {

    public PylvasOhjain(Nakyma nakyma) {
        super(nakyma);
    }

    @Override
    public void paivita() {}

    @Override
    public void kasitteleTapahtuma(Tapahtuma tapahtuma) {}
}


package fi.rapa.gradu.vaalijarjestelma.ohjaimet;

import fi.rapa.gradu.mvc.Nakyma;
import fi.rapa.gradu.mvc.Ohjain;
import fi.rapa.gradu.mvc.Tapahtuma;
import fi.rapa.gradu.vaalijarjestelma.malli.Vaalijarjestelma;

public class SyottoOhjain extends Ohjain {

    public SyottoOhjain(Nakyma nakyma) {
        super(nakyma);
    }

    @Override
    public void kasitteleTapahtuma(Tapahtuma tapahtuma) {
        ((Vaalijarjestelma) this.malli).muutaAanimaara(
            tapahtuma.palautaNimi(),
            new Long(tapahtuma.palautaArvo()));
    }

    @Override
```

```
    public void paivita() {
    }
}
```