STADIA
HELSINGIN AMMATTIKORKEAKOULU

TEKNIIKAN JA LIIKENTEEN TOIMIALA

Tuotantotalous

INSINÖÖRITYÖ

MVC-ARKKITEHTUURIN TOTEUTTAMINEN SUUNNITTELUMALLEJA KÄYTTÄEN

Työn tekijä: Laura Vesterinen
Työn valvoja: Juha-Pekka Kämäri
Työn ohjaaja: Juha-Pekka Kämäri

Työ hyväksytty: __. __. 2006

Juha-Pekka Kämäri
lehtori

ALKULAUSE

Tämä insinöörityö on tehty Helsingin ammattikorkeakoulussa vuonna 2006. Työn ohjaajana toimi lehtori Juha-Pekka Kämäri, jolle esitän kiitokseni asiantuntevasta ohjauksesta ja avusta työn tekemisessä.

Tahdon myös kiittää molempia vanhempiani perheineen tuesta opintojeni aikana.

Ville, sinä tietäisit sanomattakin, haluan kiittää sinua tuestasi, kärsivällisyydestäsi ja avustasi.

Helsingissä 30.10.2006

Laura Vesterinen

STADIA
HELSINGIN AMMATTIKORKEAKOULU

INSINÖÖRITYÖN TIIVISTELMÄ

Tekijä: Laura Vesterinen

Työn nimi: MVC arkkitehtuuri ja sen toteuttaminen suunnittelumalleja käyttäen

Päivämäärä: 30.10.2006 Sivumäärä: 46 s. + 4 liitettä

Koulutusohjelma: Tuotantotalous

Työn valvoja: lehtori Juha-Pekka Kämäri

Työn ohjaaja: lehtori Juha-Pekka Kämäri

Tässä insinöörityössä tutkittiin alan kirjallisuuden kautta MVC-arkkitehtuuria ja suunnittelumalleja. Työn tavoitteena oli toteuttaa käytännön esimerkki, joka yhdistää MVC-arkkitehtuurin ja suunnittelumallit.

Työssä lähdettiin liikkeelle perehtymällä ohjelmistoarkkitehtuuriin ja sen eri muotoihin. Keskeiseksi näistä nousi kerrosarkkitehtuuri, joka on myös MVC-arkkitehtuurin rakenne. Itse MVC ei ole käsitteenä eikä konseptina uusi. Norjalainen Trygve Reenskaug on julkaissut ensimmäiset tutkielmat siitä jo 1970-luvulla. Kirjallisuutta tutkittaessa selvisikin, että MVC on kuitenkin pysynyt suhteellisen muuttumattomana. Lisäksi se on edelleen käytössä etenkin interaktiivisten sovellusten suunnittelutyökaluna. Myös useimmat webtason sovelluskehykset käyttävät MVC-mallia.

Toinen tutkimuksen kohteena ollut alue oli suunnittelumallit. Ne ovat hyvin abstraktin tason ratkaisuja yleisimpiin suunnitteluongelmiin.

Suunnittelumalleja tutkittaessa huomattiin, että niiden käyttöä suunnittelutyössä rajoittaa niiden rajallinen tuntemus. Lisäksi abstraktin esitystavan niitä voi olla hankala soveltaa. Niiden hyödyntäminen käytännön työskentelyssä vaatiikin hyvin syvällistä ohjelmoinnin ja ohjelmointitekniikoiden tuntemusta.

Tämän työn kirjallisuustutkimusta käytettiin hyväksi painoindeksisovelluksessa, joka toteutettiin MVC-arkkitehtuurin rakenteen mukaan suunnittelumalleja apuna käyttäen. Sovelluksen toteuttamista helpotti huomattavasti MVC-rakenteen yleisyys. Käytetyt suunnittelumallitkin löytyivät suositusten perusteella. Käytettyjen ratkaisujen vuoksi ohjelman rakenteesta tuli selkeä ja se on helposti laajennettavissa.

Läpikäydyn kirjallisuuden ja toteutetun käytännön esimerkin perusteella voidaan todeta, että arkkitehtuurien ja suunnittelumallien käyttämiseen pitäisi pyrkiä. Niiden avulla saavutetaan ylläpidettäviä, muutossietoisia ja laajennettavia sovelluksia.

Avainsanat: MVC, kerrosarkkitehtuuri, suunnittelumallit, Gamma, Trygve Reenskaug

STADIA
HELSINGIN AMMATTIKORKEAKOULU

ABSTRACT

Name: Laura Vesterinen

Title: MVC architecture and its implementation through design patterns

Date: 30.10.2006 Number of pages: 46

Department: Industrial Management

Instructor: Juha-Pekka Kämäri

Supervisor: Juha-Pekka Kämäri

This study examines MVC architecture and design patterns. Purpose of this study was made practical example which joins MVC and design patterns.

At the beginning main objectives and related objectives was examined through literature. Related objectives are such as software architecture and layer architecture. Layer architecture is one form of software architecture and MVC represents it. MVC has been developed in 1970's by norwegian Trygve Reenskaug. It has been static model and it is still used especially design tool of interactive applications. Most of the web tier frameworks uses MVC model in some form.

Another part of this study was design patterns. Those are abstract solutions for common ly known design problems. Gamma and his three collegues has collected these for a book, "Design Patterns: Elements of Reusable Object-Oriented Software". These 23 models that are presented in their book are probably the best known design pattern that are used in software engineering. Most of this field literature refers Gammas book. It was found out that because patterns are abstract those are difficult to apply and use especially with fresh software designers.

The literature survey was used in practical example. Practical example is based on MVC structure. There has been also used design patterns. Because MVC structure is commonly known and there was lots of examples it was relatively easy to carry out. Used design patterns was found out through literature recommendations. Because of MVC architecture and design patterns software structure is well ordered it can be said to be expandable.

Conclusion is that aiming to use of architectures and design patterns should be purpose because these techniques helps to create maintainable, modificabable and expandable applications.

Keywords: MVC, design patterns, layered architecture, Gamma, Trygve Reenskaug

SISÄLLYS

ALKULAUSE

TIIVISTELMÄ

ABSTRACT

1 JOHDANTO

”Pyörää ei joka kerta kannata keksiä uudelleen.”

Ohjelmistotekniikka on tieteenä suhteellisen nuori, kuitenkin se kehittyy huimaavaa vauhtia. Yhteisen käsitteistön, termien ja tekniikoiden löytäminen ja kehittäminen vaatii jatkuvaa työtä ja se onkin osoittautunut yhdeksi haastavimmista tehtävistä.

Erilaisia ratkaisuja on kuitenkin koottu kirjoiksikin asti. Eräs tunnetuimmista omalla sarallaan on Design Patterns: Olio-ohjelmoinnin suunnittelumallit (Gamma et al. 1994), joka esittelee olio-ohjelmoinnin suunnittelumalleja. Malleja, ohjeita, ratkaisuja löytyy ohjelmistotyön jokaiseen vaiheeseen. Jotta näitä voidaan tehokkaasti käyttää on kuitenkin syytä tuntea ohjelmoinnin ja ohjelmistotekniikan perusteet perus- ja rinnakkaiskäsitteet.

Ohjelmistoarkkitehtuuri voidaan määritellä esimerkiksi seuraavasti ohjelmiston tai tietokoneohjelman arkkitehtuuri kuvaa järjestelmän rakenteen tai rakenteet, jotka koostuvat ohjelman komponenteista, niiden ulkoisesti näkyvistä ominaisuuksista sekä näiden välisistä suhteista.

Kerrosarkkitehtuuria voidaan kutsua myös erääksi ohjelmistoarkkitehtuurin arkkitehtuurityyliksi (macro architecture). Arkkitehtuurityylien on tarkoitus asettaa rajoitteita ohjelmiston rakenteelle ja saavuttaa tiettyjä laadullisia tekijöitä. Rakenteiden rajoittamisella tarkoitetaan ohjelmiston hierarkiaa. Laadulliset tekijät taas viittaavat rakenteiden uudelleen käytettävyyteen sekä ohjelmiston muunneltavuuteen ja kehitettävyyteen.

MVC-kerrosarkkitehtuurin alkuperäisen mallin kehitti norjalainen Tryggve Reenskaug vuonna 1979, vuosia ennen kuin suunnittelumalleista oli edes kuultu. MVC-malli lähti liikkeelle yhden miehen ajatuksesta, kun taas suunnittelumallit ovat yleisiä ja tunnettuja malleja, jotka edellä mainittu neljän kopla (Gang of Four, Gamma et al.), joka tunnetaan myös nimellä GoF, koosti yhteen, jotta saataisiin malleille yhteiset termit.

Tässä työssä tutkitaan MVC-arkkitehtuuria ja suunnittelumalleja. Tutkimuksen tavoitteena on saada aikaan käytännön sovellus, jossa ne yhdistyvät. Käytännön toteutuksen tarkoituksena on toimia esimerkkinä, jota voidaan hyödyntää vaikka ohjelmistotekniikankursseilla.

Ohjelmistosuunnittelussa, tietojärjestelmissä ja käytännön ohjelmointityössä on kannattavampaa hyötyä olemassa olevasta tiedosta ja tunnetuista perusmenetelmistä kuin yrittää kehittää uusia. Suunnittelumallit kuten MVC-arkkitehtuurikin edustavat näitä perusmenetelmiä. Tarkoituksena on myös selvittää, kuinka paljon nämä perusmenetelmät auttavat käytännön toteutuksessa.

Tässä työssä lähdetään liikkeelle arkkitehtuureista ja niiden tarkoituksesta sovelluskehitystyössä. Lisäksi kerrotaan arkkitehtuurisuunnittelun pääpiirteistä, ohjelmistoarkkitehtuurista ja kerrosarkkitehtuurista. Kolmannessa luvussa paneudutaan täysin MVC-kerrosarkkitehtuuriin ja siihen, mitä se on. MVC:tä kuvataan myös havainnollisin esimerkein. Näkökulmia MVC:hen tuodaan esille muun muassa Sun Microsystemsin tavasta käsitellä sitä, unohtamatta MVC:n mielenkiintoista ja kauaskantoista historiaa. MVC:stä siirrytään suunnittelumalleihin. Suunnittelumalleista kerrotaan ensin yleisellä tasolla, minkä lisäksi kahdesta valitusta suunnittelumallista kerrotaan esimerkein. Viidennessä luvussa toteutetaan sovellus MVC-kerrosarkkitehtuuriin ja valittuihin suunnittelumalleihin perustuen. Viimeisessä luvussa esitetään johtopäätökset.

2 ARKKITEHTUURI

Arkkitehtuuri, on laajalti käytetty termi, ja hakiessa tietoa siitä, törmää termiviidakkoon, josta löytyy kaikkea sisustusarkkitehtuurista yritysarkkitehtuuriin ja tekniikkaan liittyviin arkkitehtuureihin. Simo Vuorinen Systeemityö-lehden 03/2005 pääkirjoituksessa toteaa että, kootessaan omaa yritys- ja kokonaisarkkitehtuurimenetelmää he törmäsivät arkkitehtuuri-loppuisten termien tulvaan. He päätyivät työssään neljään perustermiin: liiketoiminta-arkkitehtuuriin, informaatioarkkitehtuuriin, järjestelmäarkkitehtuuriin ja teknologia-arkkitehtuuriin. Vuorinen jatkaa, että myös ohjelmistoarkkitehtuurimaailmassa tilanne on sama. Termejä on paljon, ja yhdelle termille voi olla useita, toisistaan poikkeavia määritelmiä. Lisäksi arkkitehtuurisuunnitteluun on useita menetelmiä. (Vuorinen 2005:4.)

Tyypillisesti sovelluksen elinkaaresta yli 70 % on tuotantokäyttö- ja ylläpitovaihetta (Poutanen 2000:18). Tuotantokäyttö- ja ylläpitovaiheen aikana selviää, kuinka ylläpidettävä ja laajennettava sovellus todella on, ja tämä vaikuttaa suoraan sovelluksen kokonaiskustannuksiin. Jotta sovellus on ylläpidettävä ja laajennettava, tarvitaan laadukas arkkitehtuuri. Se muodostaa rungon sovelluksen suunnittelulle ja sen toteuttamiselle. Lisäksi se ohjaa sovelluksen rakenteen kehittämistä koko sen elinkaaren ajan. Selkeästi kuvattu arkkitehtuurisuunnitelma toimii myös keskusteluvälineenä kehittämis- ja ylläpitosidosryhmien (organisaation johto, käyttäjät, suunnittelijat ja toteuttajat) välillä. (Wikipedia 2006b).

Arkkitehtuurien noustessa keskeiseen rooliin sekä tietohallinto-organisaatioiden työlistalla että tietojärjestelmien toteutushankkeissa tunnustetaan sen merkitys onnistuneiden hankkeiden apuvälineenä. (Vuorinen 2005:4; Poutanen 2000:18). Arkkitehtuurityöskentelyyn turvaudutaan tyypillisesti silloin, kun tarvitaan apuvälineitä monimutkaisten ja vaikeasti hallittavien teknisten kokonaisuuksien haltuunottoon. (Vuorinen 2005:4).

Jouko Poutasen mukaan arkkitehtuurikeskeisellä lähestymistavalla voidaan saavuttaa merkittäviä hyötyjä verkkokeskeisten sovellusten kehitysprojekteissa koska niissä teknologiariskit ovat suuria. Iteratiivinen työskentelyprosessi sekä arkkitehtuurin rakentaminen ja testaaminen aikaisessa vaiheessa auttavat kohtaamaan ja eliminoimaan keskeiset riskit vaarantamatta koko projektia. (Poutanen 2000:19.)

2.1 Ohjelmistoarkkitehtuuri

Ohjelmistoarkkitehtuurien tutkimus ja kehitys on monen muun tieteenlajin rinnalla suhteellisen nuorta. Arkkitehtuurien merkitys on alkanut korostua viime vuosina; myös Pekka Kähkipuro viittaa tähän artikkelissaan ”Arkkitehtuurit pelastusko tietotekniikkahaasteisiin?” (2005:5). Ohjelmistot, joissa on huolellisesti mietitty, suunniteltu, dokumentoitu ja evaluoitu arkkitehtuuri, näyttävät menestyvän paremmin, palvelevan tarkoitustaan paremmin ja tulevan pitkällä tähtäimellä edullisemmiksi. IEEE (Institute of Electrical and Electronics Engineers, Inc.) on määritellyt arkkitehtuurien kuvaamista koskevan standardin: se määrittelee ohjelmistoarkkitehtuurin järjestelmän perusorganisaatioksi. (Koskimies 2005:18.)

2.2 Ohjelmiston arkkitehtuurisuunnittelu

Ohjelmiston arkkitehtuurisuunnittelussa lähdetään liikkeelle esitutkimuksesta, jossa lyhyesti sanottuna tutkitaan, onko hankkeen toteutus ylipäätään mahdollista. Mikäli esitutkimuksen perusteella ohjelmisto todetaan toteuttamiskelpoiseksi, seuraa tätä vaatimusmäärittely tai vaatimusanalyysi. Vaatimusanalyysin määrittelee toiminnalliset ja laadulliset vaatimukset. Nämä vaatimukset ovat erittäin merkityksellisiä arkkitehtuurin kannalta. Lisäksi ohjelmistolle voi olla toissijaisia toiminnallisia vaatimuksia, joita ei välttämättä huomioida arkkitehtuurin suunnitteluvaiheessa.

Alustavassa arkkitehtuurisuunnittelussa vähimmäisvaatimuksena on yleensä kuvata toiminnallisista ominaisuuksista kohdealueen rakenneosat, niiden väliset yhteydet ja riippuvuudet, ja niiden ulospäin näkyvät ominaisuudet. (Wikipedia 2006b.) Laadullisiin ominaisuuksiin kuvataan kuinka ohjelmisto vastaa käyttäjän toiveisiin ja odotuksiin kohtuullisissa määrin. Laadullisiksi ominaisuuksiksi voidaan lukea myös uudelleenkäytettävyys, muokattavuus ja ylläpidettävyys. Laatu on aina dynaaminen käsite, ja se on riippuvainen käyttäjästä ja käyttöympäristöstä. (Pohjonen 2002:78; Vuorinen 2002:20.) Lyhyesti sanottuna alustavan arkkitehtuurin tulisi kertoa jotakin sille asetetuista tai esitetyistä vaatimuksista ja ne periaatteet, joilla näihin vaatimuksiin vastataan.

Arkkitehtuurin on myös mahdollistettava ohjelmiston ongelmakohtien löytyminen ennen toteutukseen siirtymistä. Vuorisen mukaan muita hyvän arkkitehtuurin ominaisuuksia on helppo testattavuus toiminnallisten vaatimusten

osalta, helppo mitattavuus laadullisten vaatimusten osalta, ja helppo valvottavuus ja hallittavuus. (Vuorinen 2005a:25.)

Testattavuus, mitattavuus ja valvottavuus

Jotta arkkitehtuuri olisi helposti testattavissa toiminnallisten vaatimusten osalta, olisi jo vaatimusten kirjaamisvaiheessa kiinnittää jokaiseen vaatimukseen testi, jolla voidaan varmistaa vaatimuksen toteutuminen. Vaatimus on kelvollinen vasta sitten, kun voidaan sopia, miten se testataan ja mitataan. On kuitenkin otettava huomioon, että edelle kuvattu on testaussuunnittelun ideaalitilanne. Resurssit, kuten aika ja raha, säätelevät usein käytännön suunnittelutyötä.

Laadullisista ominaisuuksista on saatava mitattavia, jotta niitä voidaan arvioida. Arkkitehtuurivaiheessa onkin sovittava mittaustavat ja -kriteerit. Laadullisten ominaisuuksien mittaustavoissa ja -kriteereissä törmätään usein ristiriitoihin, jotka on ratkaista, jotta voidaan luoda hyväksymiskriteerit. (Vuorinen 2005:26.)

Arkkitehtuuria ja sen toteuttamista on pyrittävä seuraamaan sen koko suunnitteluvaiheen ja sovelluksen toteutusvaiheen ajan. Seuraaminen voidaan toteuttaa valvomalla arkkitehtuurirakenteiden toteutumista, analysoimalla ohjelmakoodia myös koodin ja dokumentaation katselmukset ovat todettu hyödyllisiksi. (Vuorinen 2005: 25–26.)

Arkkitehtuurisuunnittelun roolit

Viljamaan mukaan (2005) arkkitehtuurisuunnittelun rooleja ovat proaktiivinen rooli (forward engineering) ja reaktiivinen rooli (reverse engineering). Proaktiivisella roolilla tarkoitetaan eri ratkaisuvaihtoehtojen arviointia. Proaktiivinen rooli määrittelee myös järjestelmän osituksen sekä moduulien välillä toteutuksessa sallitut riippuvuudet. Reaktiivisella roolilla tarkoitetaan toteutuksen yhdenmukaisuuden tarkistamista arkkitehtuurisuunnitelman kanssa. Lisäksi reaktiiviseen rooliin kuuluu riippuvuusmetriikoiden laskeminen, toteutettujen ratkaisujen vertailu, ongelmakohtien etsiminen ja korjaaminen. Ohjelmistoarkkitehtuuria voidaan pitää yleisrakenteena. Sen tarkoituksena on palvella ohjelmiston ymmärrettävyyttä, ylläpidettävyyttä, laajennettavuutta ja skaalautuvuutta, näistä neljästä muodostuu tuettavuus. (Viljamaa 2005)

Yhteenvetona arkkitehtuurisuunnittelu voidaan kiteyttää Poutasen sanoin:

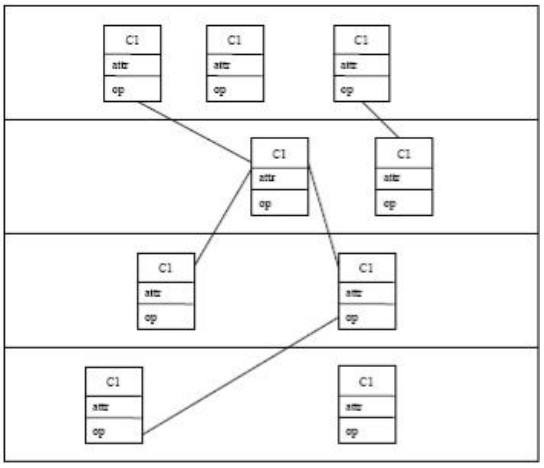
> ”*Arkkitehtuurisuunnittelu auttaa meitä rakentamaan muutossietoisia sovelluksia. Abstraktoinnin, kapseloinnin ja oliokeskeisen suunnittelun avulla voimme luoda arkkitehtuureja, joissa muutosvaikutukset ovat mahdollisimman paikallisia. Modulaarinen osajärjestelmärakenne mahdollistaa tehokkaamman työnjaon projektissa sekä järjestelmäosien testattavuuden parantumisen. Vaikka kehitysprojektissa kustannukset painottuvat alkuvaiheisiin, voidaan projektin aikana saavuttaa merkittäviä säästöjä toimivan perusinfrastruktuurin avulla* (Poutanen 2000:19).”

Ohjelmistoarkkitehtuurin muotoja on useita, kaikkiin niihin kuitenkin pätevät samat yllä esitetyt vaatimukset ja ne kaikki pyrkivät samaan lopputulokseen. On kuitenkin päätettävä, mikä näistä muodoista on omaan tarkoitukseen parhaiten sopiva. MVC-malli rakentuu kerrosarkkitehtuurin muotoon, josta seuraavassa enemmän.

2.3 Kerrosarkkitehtuuri

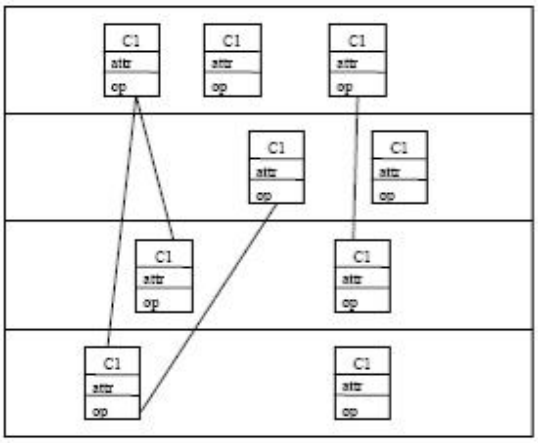
Kerrosarkkitehtuurin ajatuksena on, että toiminta jakautuu eri abstraktiotason operaatioihin, jotka jaetaan loogisesti yhtenäisiin kerroksiin. Kerroksessa ylemmän tason operaatio suoritetaan kerroksissa alemmalla tasolla olevien operaatioiden avulla (Laine 2000a). Toiminta tapahtuu yleensä välillä laitteisto - sovellus. Tietoliikennetekniikasta tuttu OSI-malli on tunnetuin esimerkki kerrosarkkitehtuurista. Muita esimerkkejä ovat TCP/IP sekä virtuaalikoneet, joissa voidaan puhua alemman kerroksen virtuaalisista koneista. Ne eristävät korkeamman kerroksen alemman kerroksen yksityiskohdista tai laitteistomuutoksista että sovellusrajapinnat palveluihin kuten API (application programming interface). (Laine 2000a; Eerola 2004.) Esimerkkinä kannattaa myös mainita nelikerrosarkkitehtuuri, jossa tietokanta, liiketoimintalogiikka, sovelluslogiikka ja käyttöliittymä on erotettu toisistaan kerroksiin. (Järvi ja Alhoniemi 2006:63.)

Kerrosarkkitehtuuri voidaan jakaa niin sanottuun suljettuun kerrosarkkitehtuuriin (opaque layering) ja avoimeen kerrosarkkitehtuuriin (transparent layering). Suljettu kerrosarkkitehtuuri tuntee vain välittömästi alempana olevan kerroksen (kuva 1). Suunnittelun tavoitteissa se tukee ylläpidettävyyttä, siirrettävyyttä, joustavuutta, muokattavuutta ja ymmärrettävyyttä. (Järvi ja Alhoniemi 2006:55.)



Kuva 1. Suljettu kerrosarkkitehtuuri (Järvi ja Alhoniemi 2006: 55)

Avoin kerrosarkkitehtuuri tuntee kaikki alemmat kerrokset ja voi kutsua niiden palveluita (kuva 2). Suunnittelun tavoitteita avoimessa kerrosarkkitehtuurissa on ajonaikainen suoritustehokkuus, nopea toteutettavuus. Ymmärrettävyys, ylläpidettävyys ja siirrettävyys kärsii avoimuudesta. (Järvi ja Alhoniemi 2006:56.)



Kuva 2. Avoin kerrosarkkitehtuuri (Järvi ja Alhoniemi 2006:56)

Kerrosarkkitehtuuri jakaa siis toiminnalliset osiot yhtenäisiin kerroksiin. Seuraavassa osioissa tutkitaan yhtä tunnetuimmista kerrosarkkitehtuurin muodoista, MVC-arkkitehtuuria.

3 MVC-KERROSARKKITEHTUURI

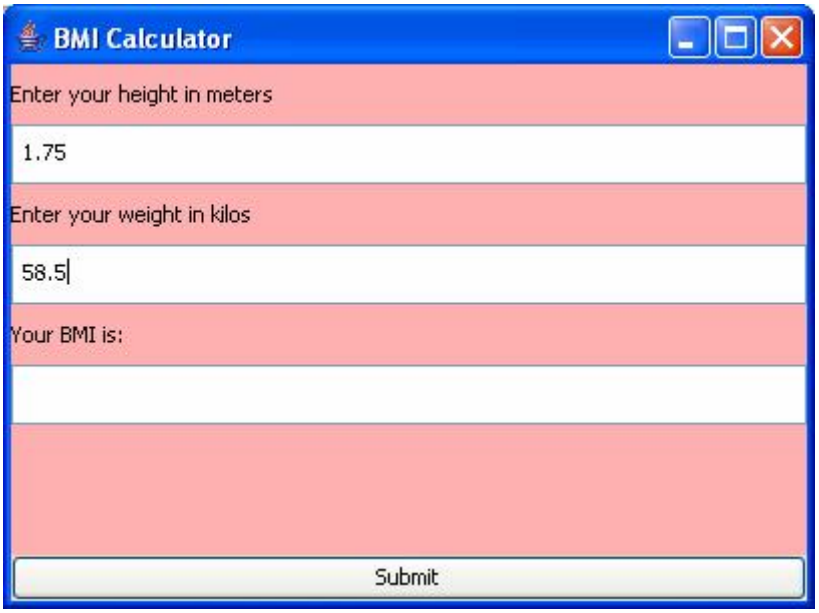
MVC-kerrosarkkitehtuuri (Model - View - Controller) perustuu norjalaisen Trygve Reenskaugin vuonna 1979 kehittämään malliin. MVC:ssä tarkoituksena on erottaa käyttöliittymä sovellustiedosta. Erityisesti MVC-mallia käytetään graafisten käyttöliittymien suunnitteluun ja apuna niiden ohjelmointiin. Sovellusalueena on erityisesti interaktiiviset järjestelmät, jotka on varustettu monimuotoisin käyttöliittymin. Sun Microsystemsin mukaan erityisesti interaktiiviset J2EE-järjestelmät hyötyvät MVC-arkkitehtuurin käyttämisestä. Erityisesti MVC sopii interaktiivisiin web-sovelluksiin, joissa webin käyttäjä on vuorovaikutuksessa www-sivuston kanssa, esimerkiksi useiden edestakaisten pyyntöjen (request) ja vastausten (response) kautta. Useimmat web-tason sovelluskehykset käyttävät jossain muodossa MVC-mallia. (Sun Microsystems 2002b.)

3.1 Mikä on MVC?

MVC muodostuu kolmesta osasta: Model (malli/sisältö), View (näyttö/näkymä) ja Controller (ohjain) . Seuraavassa jokaisesta näistä osasta tarkemmin. Lisäksi jokaisen kerroksen jälkeen on esitetty esimerkki Javalla toteutetusta painoindeksilaskimesta.

Näyttö (view) on sisällön esitystapa joko koko tai osatiedolle, joka on liitetty kyseiseen näyttöön. Näyttö käyttää hyväkseen sisällön kyselypalvelua saadakseen selville sisällön tilan. Vaihtoehtoisesti myös sisältö voi ilmoittaa tilansa muuttumisesta ja näyttö saa mahdollisuuden päivittää itsensä. Tällainen lähestymistapa mahdollistaa useiden näyttöjen liittämisen samaan sisältöön, jotta voidaan muodostaa erilaisia esitystapoja.

Painoindeksilaskin esimerkissä kuvassa 3. käyttäjän on antanut tietonsa tekstikenttiin näytölle painamalla Submit –nappulaa, jolloin tieto siirtyy tekstikentistä ohjaimelle.



Kuva 3. Painoindeksilaskin, käyttäjä syöttänyt arvot tekstikenttiin

Ohjaimen (controller) tehtävänä on määritellä, kuinka käyttöliittymä reagoi käyttäjän antamiin syötteisiin. Ohjain voi vaikuttaa sekä malliin että näkymään tai vain toiseen niistä. Esimerkkinä käyttäjä klikkaa hiirtä ohjain ottaa tämän syötteen vastaan ja muuntaa sisällön tilaan vaikuttavaksi operaatioksi.

Esimerkissä ohjain välittää tekstikentistä poimitut tiedot muodossa double mallille, joka suorittaa painoindeksin laskennan. Malli palauttaa tuloksen ohjaimelle, joka edelleen välittää tiedon näytölle.

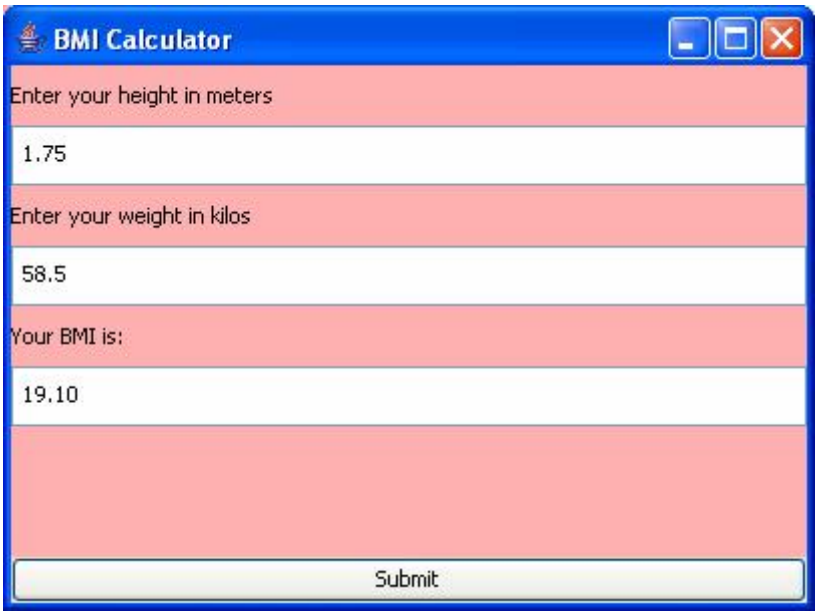
```
if (e.getActionCommand().equals("Submit"))
 {
double height = Double.parseDouble(view.getHeigth());
double weight = Double.parseDouble(view.getWeight());
double bmi = model.processData(weight, height);
Double d1 = new Double(bmi);
String BMIresult = new String(d1.toString());
view.showResult(BMIresult);
 }
```

Malli (model) on sovellusolio, toisin sanoen se pitää sisällään esitettävän tiedon ja niiden muuttamiseen tarvittavat operaatiot. Mallin ei tarvitse tietää, eikä se tiedä mitään käyttöliittymästä, tietojen esittämistavasta eikä tiedolle käyttöliittymässä tehtävistä operaatiosta. Koska näkymä ja malli on erotettu toisistaan, niiden välille on perustettu tilaa/ilmoita-protokolla.

Mallissa suoritetaan painoindeksin laskenta ja tulos palautetaan ohjaimelle.

```
public static double processData(double weight, double height)
```

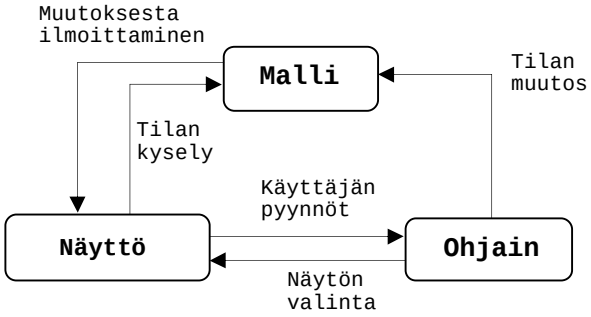
```
{
  double BMIresult = weight/(height*height);
  return BMIresult;
}
```



Kuva 4. Laskenta suoritettu, arvo palautettu käyttäjälle

Kuvassa 4. käyttäjälle on palautettu laskennan tulos. Liitteessä 1 on koodi laskimen toteuttamiseen.

Johtuen tästä kerroksiin erottelusta useat näytöt ja ohjaimet voivat käyttää samaa sisältöä rajapintojen kautta. Jopa uudet näytöt ja ohjaimet voivat käyttää samaa mallia rajapintojen kautta, ilman että mallia tarvitsee muuttaa. Esimerkiksi yllä esitetystä painoindeksilaskimesta voitaisiin tuottaa matkapuhelinsovellus lisäämällä rajapinta ja näyttö. Lisäksi koko ohjelmisto on helpommin muunnettavissa tai sen uudelleen käyttäminen on helpompaa, koska nämä kolme on erotettu toisistaan. Kuvassa 5 esitetään MVC-prosessi yleisellä tasolla.



Kuva 5.MVC

Yllä esitetyssä mallissa tapahtumien siirtyessä näytöltä ohjaimelle, ohjain muuttaa mallia, näyttöä tai molempia. Ohjaimen muuttaessa tietoa tai ominaisuuksia mallissa, malli ilmoittaa muutoksestaan näytöille, jotka päivittävät itsensä. Vastaavasti ohjain voi muuttaa näyttöä esimerkiksi paljastaen alueen, joka oli aiemmin piilossa (vierityspalkit).

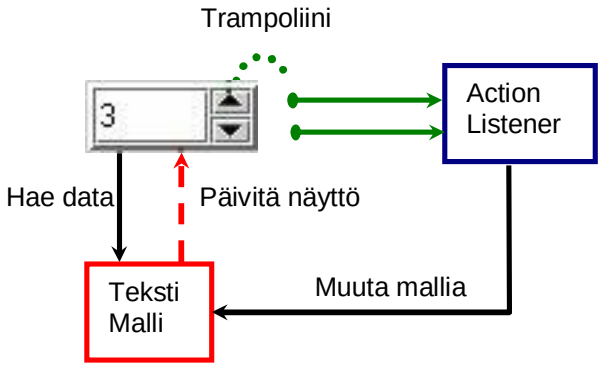
Painoindeksilaskin esimerkin kannalta yllä esitetty kuva voidaan selittää seuraavasti: Käyttäjä syöttää tekstikenttiin pituutensa ja painonsa, painaa Submit-nappulaa, missä seurauksena tiedot (pituus ja paino) lähetetään ohjaimelle. Koska tiedot ovat näytöllä tekstimuotoisina ohjain muokkaa tiedon mallia varten numeeriseksi, jotta malli voi suorittaa laskennan. Laskennan suoritettuaan malli palauttaa tuloksen ohjaimelle, joka kertoo näytölle päivitystarpeesta.

Myös Gamma et al. (1994:5) on ottanut kantaa MVC-malliin: ”MVC-mallissa voidaan muuttaa tapaa, jolla näkymä reagoi käyttäjän syötteeseen muuttamatta näkymän visuaalista esitystä. Halutaan esimerkiksi muuttaa tapaa, jolla näkymä reagoi näppäimistöön tai vaihtaa komentonäppäinten käyttö alasvetovalikkoon. MVC kapseloi reagointitavan ohjain-olioon. Ohjaimet muodostavat luokkahierarkian, joten on helppo muodostaa uusi ohjain varioimalla/muuttamalla jotain olemassa olevaa ohjainta” (Gamma et al. 1994:5).

”Näkymä käyttää ohjain-aliluokan ilmentymää tietyn reagointitavan toteuttamiseen; toisenlainen tapa toteuttaa yksinkertaisesti vaihtamalla ilmentymä toisenlaiseen kontrolleriin. On jopa mahdollista vaihtaa näkymän kontrolleria ajoaikana, jolloin näkymä voi muuttaa tapaansa reagoida käyttäjän syötteeseen. Esimerkiksi yksinkertainen tapa estää näkymää toimimasta niin, ettei se reagoi syötteeseen, on antaa sille ohjain, joka ei käsittele syöttötapahtumia” (Gamma et al. 1994:6).

3.2 Esimerkki MVC:stä spinner–komponentissa

Aiemmin esitettiin esimerkki painoindeksilaskimesta; seuraavassa esitetään MVC:n toimintaa spinner-komponentissa. Spinner-komponentin muodostavat tekstikenttä ja kaksi nuolinäppäintä. Nuolinäppäimistä painamalla voidaan lisätä tai vähentää tekstikentässä esitettävää numeerista arvoa. (eNode, Inc. 2002.)



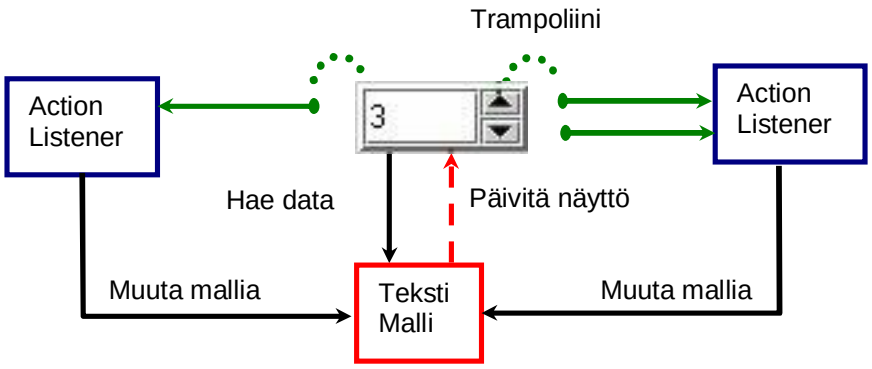
Kuva 6. Spinner komponentti MVC:ssä (eNode, Inc. 2002)

Yllä olevan spinner-komponentin data säilytetään mallissa, joka on jaettu tekstikentän kanssa. Tekstikenttä toimii näyttönä spinner-komponentin arvolle. Komponentin nuolinäppäimet laukaisevat tapahtuman (action event) joka kerta niitä painettaessa. Nuolinäppäimet voivat olla kytkettyjä trampoliiniin (trampoline), joka ottaa vastaan tapahtumia (action event) ja ohjaa niitä tapahtumakuuntelijoille (action listener), jotka käsittelevät tapahtuman. Uudelleenkutsussa trampoliinille on määritelty oma tapahtumakuuntelija , joka ohjaa tapahtuman käsittelyn toiselle kuuntelijalle. (eNode, Inc. 2002.)

Tapahtuman lähteestä riippuen tapahtumakuuntelija (ultimate action listener) joko kasvattaa tai pienentää arvoa, joka on mallissa. Tapahtumakuuntelija toimii tässä esimerkkinä ohjaimesta. Trampoliinit, jotka alkuaan saavat tapahtumat nuolinäppäimistä, ovat myös ohjaimia. Sen sijaan, että ne muokkaisivat spinner-komponenttia suoraan, ne delegoivat tehtävän erilliselle ohjaimelle (action listenerille). (eNode, Inc. 2002.)

MVC-malli antaa myös useampien ohjaimien muokata samaa mallia. Käyttäen edellä mainittua esimerkkiä voidaan siihen lisätä kolmas tapahtuma: Syötettäessä arvo suoraan tekstikenttään ja painettaessa Enter-painiketta, se laukaisee tapahtuman. Tämä tapahtuma hoidetaan eri tapahtumakuunte-

lijan kautta kuin nuolinäppäimiä painettaessa. Tästä esimerkki kuva alla. (eNode, Inc. 2002.)

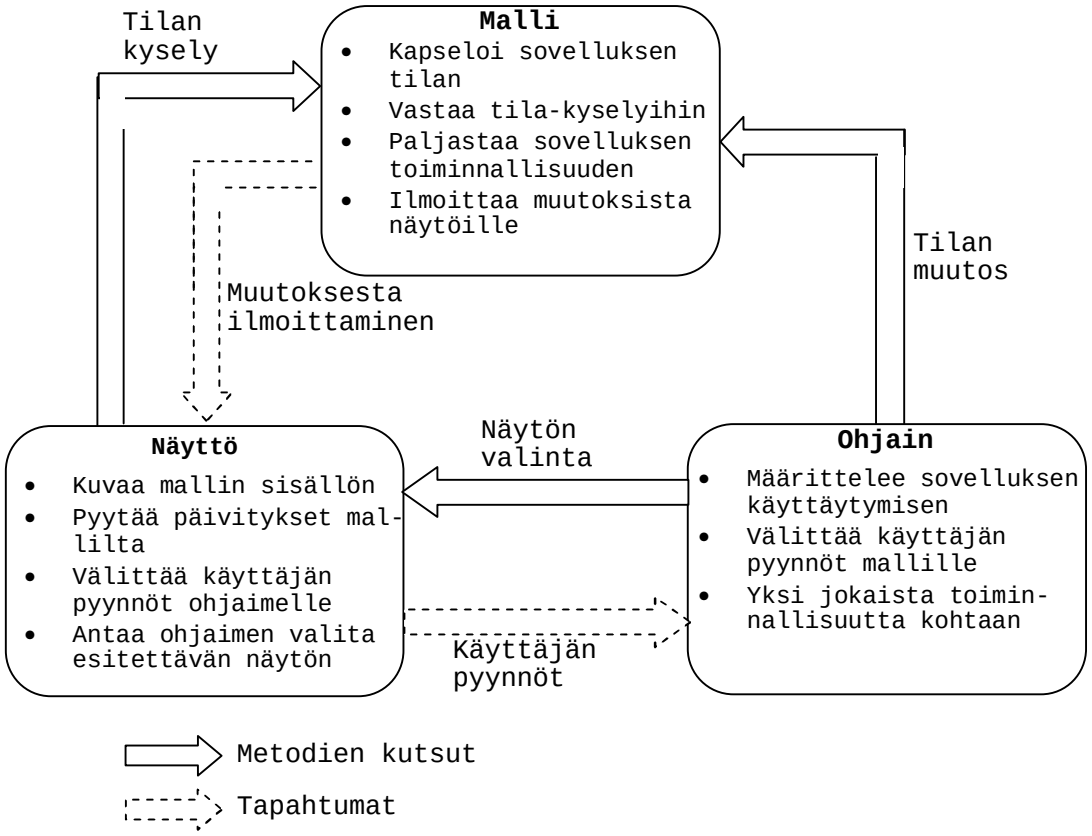


Kuva 7. Usean ohjaimen MVC malli (eNode, Inc. 2002)

Edellä mainitun esimerkin tarkoitus on selventää, MVC:n toimintaa spinner-tyyppisessä komponentissa. Seuraavassa perehdytään MVC-malliin Sun Microsystemsin mukaan. Sun Microsystems on tunnettu muun muassa Java ohjelmointikielen kehittämisestä.

3.3 MVC Sun Microsystemsin mukaan

MVC-arkkitehtuuri on laajalti käytetty arkkitehtoninen lähestymistapa interaktiivisiin sovelluksiin. Alun perin mallia sovellettiin GUI-malliin (graphical user interaction model), jossa se käsitteli perinteisiä käyttäjän antamia syötteitä (input), joista saatiin tulosteita (output). Jokainen näistä kolmesta malli, näyttö ja ohjain, erottaa kullekin kuuluvan vastuualueen. Jokainen kerros siis hoitaa niille ominaisia tehtäviä, ja niillä on niille ominaiset vastuualueet muille alueille. (Sun Microsystems 2002a.)



Kuva 8. Malli-, näyttö- ja ohjain-kerrosten väliset riippuvuudet MVC-sovelluksessa (Sun Microsystems 2002a)

Kuva 8 esittää Sun Microsystemsin mukaan mallin, näytön ja ohjaimen toiminnot sovellettuna monitasoisiin web-pohjaisiin yrityssovelluksiin.

Mallin (model) tarkoituksena on esittää yrityksen tietoa ja toiminnan säännöt (business rules) jotka kattavat pääsyn ja ylläpito-oikeuden tähän tietoon. Usein malli tarjoaa ohjelmallisen arvion todellisen maailman prosesseista. Joten yksinkertainen todellisen maailman mallinnustekniikoiden soveltaminen on paikallaan, kun määritellään mallia (ks. 3.5). Malli ilmoittaa näytölle, kun se muuttuu ja tarjoaa näytölle mahdollisuuden kysyä mallilta sen tilasta. Se tarjoaa myös ohjaimelle pääsyn sovelluksen toimintoihin, jotka malli on kapseloinut.

Näyttö (view) esittää mallin sisältöä. Näytöllä on pääsy yrityksen tietoon mallin kautta, ja se määrittelee tiedon esitystavan. Näytön vastuualueena on pitää yllä yhtenäinen esitystapa kun malli muuttuu. Tämä saavutetaan käyttä-

mällä työntömallia (push model), jossa näyttö rekisteröi itsensä mallin kanssa, jotta se saa tiedon muutoksista, tai vetomalli (pull model), jossa näyttö on vastuussa mallin kutsumisessa, kun se tarvitsee kaikkein uusinta tietoa.

Ohjain (controller) kääntää vuorovaikutuksen näytön kanssa toiminnaksi, jonka malli suorittaa. Yksittäisen GUI-asiakkaan (graphical user interface client) käyttäjän toiminta voi olla näppäimen painaminen tai valikkotoiminnon käyttäminen, web-sovelluksessa nämä esiintyvät GET ja POST HTTP-pyyntöinä. Ohjaimen suorittamiin toimintoihin kuuluu liiketoimintaprosessien käynnistäminen tai mallin tilan muuttaminen. Perustuen käyttäjän toimintoihin ja mallin toiminnan seurauksiin ohjain vastaa valitsemalla sopivan näytön. Sovelluksella on usein yksi ohjain jokaista saman sukuista toiminnallisuusjoukkoa kohtaan. Jotkut sovellukset voivat käyttää erillisiä ohjaimia jokaiselle palvelimen asiakkaalle, koska näytön valinta ja toiminnot vaihtelevat asiakkaan mukaan.

Erottamalla toiminnot malli-, näyttö- ja ohjainolioiden kesken vähennetään koodin toistoa ja tehdään järjestelmästä helpommin ylläpidettävä. Tämä tekee myös tiedon käsittelystä helpompaa lisättäessä sitten uusia tiedon lähteitä tai muuttamalla tiedon esitystapaa, koska toimintalogiikka on erotettu tiedosta. On myös helpompaa tukea uusia asiakastyyppejä, koska ei ole tarpeen muuttaa toimintalogiikkaa jokaisen uuden asiakkaan mukaan.

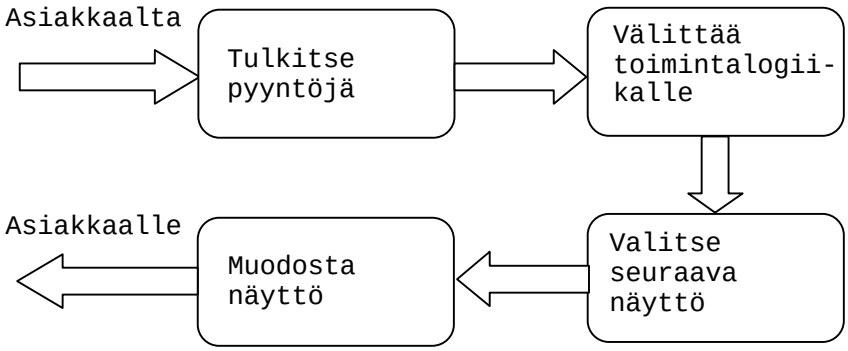
3.4 MVC-malli syvemmin Sun Microsystemsin mukaan

MVC-malli tarjoaa pohjan suunnittelulle. Se erottelee suunnitteluongelmat kuten tiedon jatkuvuus ja käyttäytyminen, esittäminen, ja kontrollointi, vähentää koodin kopiointia, keskittää kontrolloinnin ja tekee sovelluksesta helpommin muunnettavan. MVC auttaa eritasoisia kehittäjiä asettamaan painopisteen omaan pääosaamisalueeseen ja tekemään yhteistyötä muiden kanssa selkeästi määriteltyjen rajapintojen kautta. Esimerkiksi J2EE-sovellusprojekteissa voi olla useita eri sovelluskehittäjiä näytöille, sovelluslogiikoille, tietokantojen toiminnallisuudelle ja verkonhallintaan. MVC-mallin avulla voidaan sovelluksen toiminnot, kuten tietoturva (security), kirjautuminen (logging) ja näytön vierittäminen (screen flow) keskittää. Uusia tiedon lähteitä on helppo lisätä MVC-sovellukseen luomalla koodia, joka omaksuu uuden tiedon API-näytön (application programming interface) kautta. Vastaavasti uuden tyyppiset asiakkaat on helppo lisätä sovittamalla uusi

asiakas toimimaan kuten MVC:n näyttö. MVC määrittelee selkeästi osallistuville luokille vastuut, tehden virheistä (bug) helpompia jäljittää ja korjata. (Sun Microsystems 2002b.)

3.4.1 MVC web-sovelluksen suunnittelussa

Tässä luvussa kuvataan Sun Microsystemsin mukaan, kuinka käyttää MVC:tä organisoimaan J2EE (Java 2 Enterprise Edition) web-sovelluksen suunnittelua käyttäen web-sovelluskehysmallia esimerkkinä. J2EE-sovelluksen web-taso käsittelee HTTP-pyyntöjä (HyperText Transform Protocol). Korkeimmalla tasolla web-taso (Web tier) tekee neljä perusasiaa tietyssä järjestyksessä (kuva 9): tulkitsee asiakkaan pyyntöjä, lähettää pyynnöt toimintalogiikalle, valitsee seuraavan esitettävän näytön ja muodostaa ja tuottaa seuraavan näytön.



Kuva 9. Web-tason palvelun kierto (Sun Microsystems 2002b)

Web-tason ohjain saa jokaisen tulevan HTTP-pyynnön ja välittää pyydetyn sovelluslogiikkaan kohdistuvan operaation mallille. Perustuen operaatioon ja mallin tilaan ohjain valitsee näytön ja seuraavan esitettävän näytön. Lopulta ohjain luo valitun näytön ja välittää sen esitettäväksi asiakkaalle.

Kuvassa 9 esitetty rakenne harhaanjohtavan yksinkertainen. Yrityssovelluksen web-tasolla on yleensä seuraavia vaatimuksia.

- Sovellussuunnitelmassa on oltava strategia nykyisille ja uusille asiakastyypeille.
- Web-tason ohjaimen täytyy olla ylläpidettävä ja laajennettava. Sen tehtäviin kuuluu pyyntöjen sijoittaminen sovelluksen mallin operaati-

oiksi, näyttöjen valinta ja kokoaminen ja näytön vierittämisen hallinta. Hyvä rakenne voi minimoida koodi kompleksisuutta.

- Sovellusmallin API:n suunnittelulla ja tekniikan valinnalla on suuri vaikutus sovelluksen monimutkaisuuteen, skaalattavuuteen ja ohjelmiston laatuun.

- Valittaessa sopiva teknologia dynaamisen sisällön luomiseen tuetaan ja parannetaan kehitystyötä sekä ylläpidon tehokkuutta.

(Sun Microsystems 2002b.)

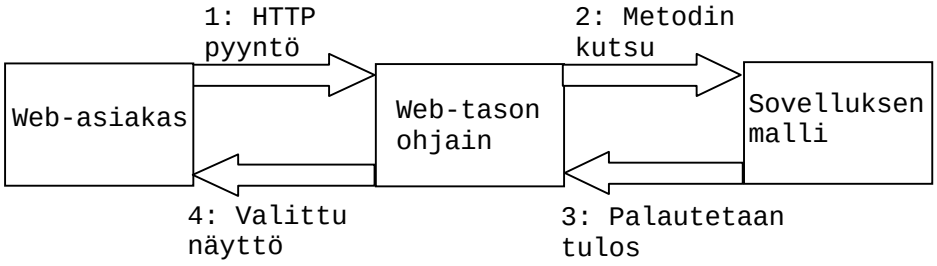
3.4.2 J2EE sovellus web-tasolla

Seuraavassa Sun microsystems (2002b) kuvaa yleisen mallin J2EE-sovellukselle web-tasolla. Lisäksi seuraava auttaa ymmärtämään ja käyttämään web-sovelluskehystä, ja käytettäessä omaa web-tason arkkitehtuurikoodia se auttaa ymmärtämään, kuinka käyttää kyseistä teknologiaa.

Web-sovellukset voivat tukea jokaista asiakasta omalla protokollalla, turvallisuusmenetelmällä ja esityslogiikalla. Web-asiakkaisiin voi kuulua useita versioita useista eri selaimista, MIDP-asiakkaita (Mobile Information Device Profile), ja niin kutsuttuja raskaita asiakkaita ("rich" clients). Pitkäikäisten sovellusten täytyy voida käsitellä myös uudentyyppisiä web-asiakkaita.

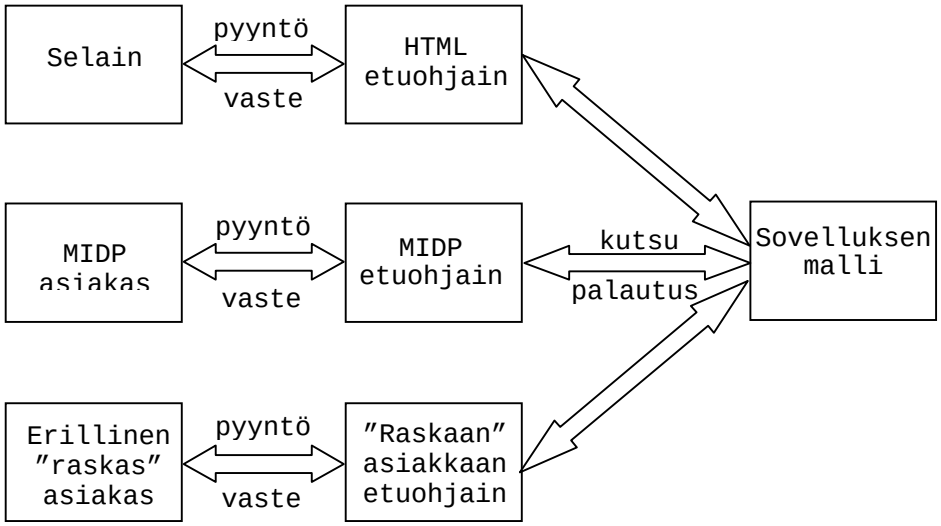
Jokainen asiakastyyppi tarvitsee oman ohjaimen, joka toimii asiakastyypin vaatiman protokollan mukaan. Tietyntyyppiset asiakkaat saattavat tarvita myös erilaisia esityskomponentteja muotoilutekijöiden tai muiden syiden takia (kuva 11).

Seuraavassa joitain mahdollisuuksia, kuinka palvella pyyntöjä asiakkailta, jotka käyttävät eri sovellustason protokollia. Web-tason asiakkaat käyttävä HTTP:tä kuljettamiseen. Jokainen seuraavista kuvissa (kuvat 10, 11 ja 12) esitetyistä vaihtoehdoista on laajennuksia kuvasta 9, ja jokaisessa joustavuus ja kompleksisuus lisääntyy.



Kuva 10. Etuohjaimen käyttämien selaimen vuorovaikutuksen käsittelyssä (Sun Microsystems 2002b)

Sovellukset, joilla on vain yksi asiakastyyppi, voivat käyttää yksinkertaista etuohjainta (front controller). Esimerkiksi vain selainkäyttöinen sovellus on esitetty kuvassa 10. Sen ainut etuohjain, servletti, saa HTTP-pyyntöjä selaimelta, kääntää näiden pyyntöjen sisällön operaatioiksi sovelluksen mallille ja palvelee näyttöjä, jotka esittävät tulokset HTML:nä (HyperText Markup Language) tai XML:nä (Extensible Markup Language). Lisäksi ohjaimet voivat tukea uusia asiakastyyppejä kuten kuvassa 11 on esitetty.

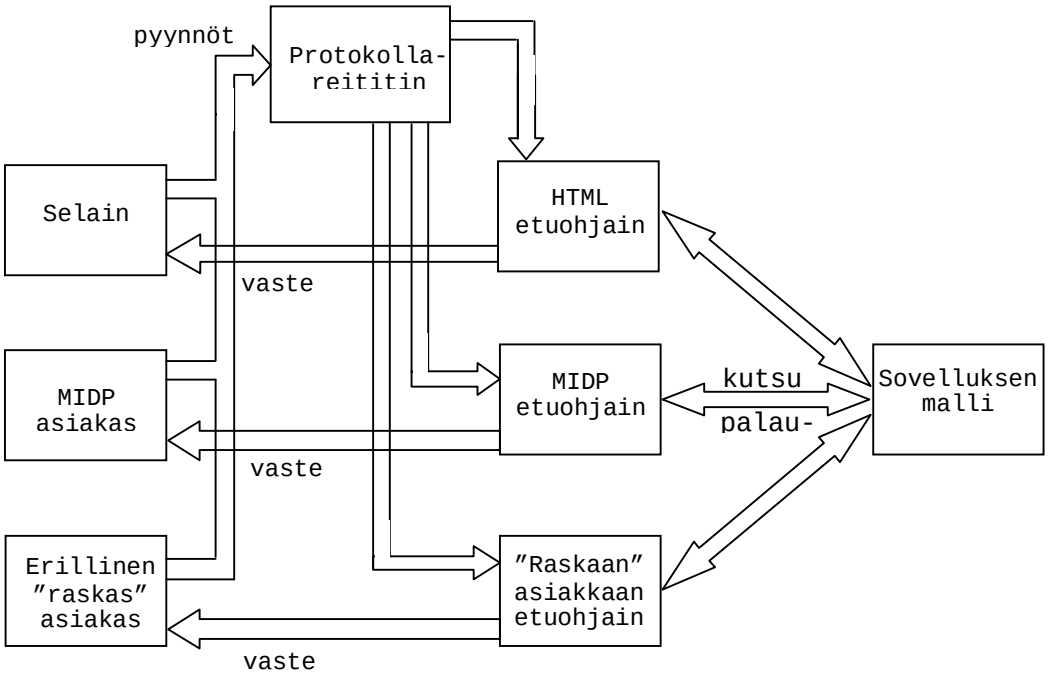


Kuva 11. Useiden asiakastyyppien tukeminen useilla ohjaimilla (Sun Microsystems 2002b)

Moninkertaisen ohjaimen lähestymistapa tarjoaa (kuva 11) laajennettavuutta mille tahansa tulevaisuuden web-asiakastyypeille mukaan lukien myös ne joita ei vielä tunneta. Koska servlettejä ei ole rajoitettu pelkästään HTTP:hen, tämä arkkitehtuuri voi tukea myös muita kuin web-asiakkaita. Jokainen ohjain voi toteuttaa työmäärän, esityslogiikan, ja turvallisuusrajoitteet, jotka ovat yksilöllisiä jokaiselle asiakastyypille. Huomattakoon

myös, että koodi, joka on toteutettu sovelluksen malliin, on jaettu kaikille ohjaimille. Tämä erottelu malliin ja ohjaimiin varmistaa identtisen sovelluskäyttäytymisen riippumatta asiakastyypistä, ja lisäksi se helpottaa ylläpitoa, huoltoa ja testausta.

Jotkut sovelluksen toiminnot, etenkin turvallisuus, voivat olla helpommin hallittavissa vain yhdestä kohtaa, siis paikallisesti. Liittämällä protokollareitittimen (protocol router), joka on näytetty kuvassa 12, voi tarjota aiemmin mainitun yhden pisteen hallittavuuden kaikille web-asiakkaille, jättäen kuitenkin jokaiselle oman ohjaimen.



Kuva 12. Protokollareitittimen käyttäminen keskitettyyn kontrolliin (Sun Microsystems 2002b)

Protokollareititin (protocol router), jota on käytetty kuvassa 12, on joko servletti tai servlettisuodatin (servlet filter), nämä määrittelevät asiakastyypin ja välittävät pyynnön soveltuvalle ohjaimelle. Reititin käyttää tyypillisesti käyttäjän edustajan HTTP-otsikkoa (HTTP header user-agent) määrittääkseen, minkä tyyppinen asiakas on lähettämässä pyyntöä. Protokollareititin voidaan toteuttaa sovelluksenlaajuisiin toimintoihin, kuten turvallisuus ja kirjautuminen. Asiakkaille ominaiset ohjaimet voidaan toteuttaa jokaisen asiakkaan protokollaan sopiviksi asiakkaan käyttäytymisen mukaan.

Etuohjaimet voivat olla servlettejä, ja mikäli näin on, protokollareititin lähettää pyynnöt käyttäen `RequestDispatcher.forward` -komentoja. Mutta jos taas protokollareititin on servletti etuohjaimet voivat olla kerroksen yksinkertaisia määräasemia, joille reititin delegoin pyyntöjen prosessoinnin.

Ohjain-vaihtoehdot jotka on esitetty yllä olevissa kuvissa, voidaan toteuttaa inkrementaalisesti, eli jokainen lähestymistapa voidaan rakentaa aiemman päälle. Sunin Blue Prints suosittaakin valitsemaan mahdollisimman soveltuvan sen hetkiseen tilanteeseen ja lisäämään toimintoja tarvittaessa. (Sun Microsystems 2002b.)

Web-tason ohjaimen malli MVC:ssä

Web-tasolla MVC:n ohjain käsittelee ja välittää tulevat toimintopyynnöt sovelluksen mallille ja valitsee näytöt perustuen malliin ja istunnon tilaan. Web-tason ohjaimilla on paljon velvollisuuksia, joten ne vaativat huolellista suunnittelua, jotta ne voivat hoitaa sovelluksen kompleksisuuden. Koska useimmat yrityssovellukset kasvavat ajan myötä, laajennettavuus on merkittävä vaatimus. (Sun Microsystems 2002b.)

Web-tason näytön malli MVC:ssä

MVC:ssä näytöt esittävät tiedon, jonka malli on tuottanut. Näytön komponentit (tunnetaan myös esityskomponentteina, presentation components) web-tasolla ovat usein JSP-sivuja (Java Server Pages) ja servlettejä, mukana on myös staattisia resursseja kuten HTML-sivuja, PDF-tiedostoja, grafiikkaa vain muutamia mainitakseni. JSP-sivut on yleensä parhaita jos halutaan luoda tekstipohjaista sisältöä, usein HTML:ää tai XML:ää. Servletit taas soveltuvat parhaiten binaarisen sisällön luomisessa tai sisällön, jolla on vaihtuva rakenne. HTML-selaimet ovat hyvin kevyitä asiakkaita, joten web-taso luo dynaamisen sisällön selaimille. Raskaille asiakkaille voidaan tuottaa suhteellisesti enemmän toiminnallisuutta asiakastasolla (Client tier) ja vähemmän webtasolla. Edellä mainittuihin asiakkaisiin kuuluvat yksittäiset raskaat asiakkaat, sovellusasiakkaat ja asiakkaat, jotka käyttävät erityisiä sisältä formaatteja kuten MacroMedia Flash tai Adobe Portable Document Formaattia (PDF). (Sun Microsystems 2002b.)

Web-tason komponentteja ei ole rajoitettu palvelemaan HTML-yli-HTTP-web-selaimia. Web-taso voi palvella myös mobiiliasiakkaita (MIDP) käyttä-

mällä soveltuvia protokollia, raskaita asiakkaita (rich client) käyttäen XML:ää, tai webin vertaispalveluja (Web service peers) pyytäen palveluita ebXML (Electronic Business XML) käyttäen tai SOAP (Simple Object Access Protocol) viestejä käyttäen. Jokainen näistä esimerkeistä käyttää erisovellustason protokollaa, käyttäen HTTP:tä kuljetukseen. Hyvin suunniteltu web-taso yhdistää pääsyn sovelluksen toimintoihin miltä tahansa asiakkaalta. Web-taso tarjoaa virtuaalisen istunnon hallinnan joillekin asiakastyypeille.

Web-tason mallin malli MVC:ssä

MVC-sovelluksen malli esittää yrityksen tietoa ja toteuttaa toimintalogiikan (business logic). Monet J2EE-sovellukset implementoivat malliinsa Enterprise bean:eja, jotka tarjoavat skaalattavuutta, moniajettavuutta, kuormituksen tasapainoa, automaattista resurssien hallintaa ja muita etuja Yksinkertaisemmat J2EE-sovellukset voivat toteuttaa mallin kokoelmana web-tason JavaBeans-komponenteista joita käytetään suoraan JSP-sivujen tai servlettien kautta. JavaBeans-komponentit tarjoavat nopean pääsyn dataan, kun taas enterprise beans -komponentit tarjoavat pääsen jaettuun toimintalogiikkaan ja tietoon.

Huomaa, että kuvassa 10 esitetty sovelluksen malli on yleinen: se ei sovella tiettyä teknologiaa tai tasoa. Sovelluksen malli on yksinkertainen ohjelmallinen rajapinta sovelluksen toimintalogiikkaan. Malli API:n suunnittelu ja mallin käyttämä teknologia ovat molemmat merkittäviä suunnitteluvaiheen päätöksiä. (Sun Microsystems 2002b.)

Esimerkkien ja johdatuksen MVC:hen jälkeen siirrymme ajassa taaksepäin 70-luvun loppuun, jolloin norjalainen tohtori Trygve Reenskaug kehitteli ajatuksen MVC-mallista.

3.5 MVC:n historiikki

MVC:n historia juontaa juurensa 1970-luvun alkuun, jolloin olio-ohjelmoinnista ei vielä edes haaveiltu. Tutkijat olivat vasta alkaneet tutkia käsitteitä hajautettu (distributed) ja kommunikaatiokomponentit (communication components). MVC:n historiikissa kuvataan historiaa Reenskaugin ensimmäisen artikkelin mukaan (1979) sekä Ruppin artikkelin (2003) mukaan.

Seuraavassa palataan ajassa vuoden 1973 Osloon, Norjaan. Trygve Reenskaugin muistiinpanot tuolta ajalta muistuttavat hyvin siitä, että ongelmat joita kohtaamme nykyään hajautetussa tietojenkäsittelyssä, ovat luonteeltaan samanlaisia, joita tutkijat kohtasivat kolme vuosikymmentä sitten. Lisäksi tästä voidaan päätellä, kuinka paljon on vielä opittavaa monimutkaisten järjestelmien yksinkertaistamisessa.

Elokuussa vuonna 1973 Tohtori Reenskaug kirjoitti tutkielman ”Administrative Control in the Shipyard” eli vapaasti suomennettuna Telakan hallinnollinen ohjaaminen, jonka hän esitteli kansainvälisessä konferenssissa, joka koski tietojenkäsittelysovelluksia laivanrakennuksessa.

Tutkielmassaan Reenskaug analysoi modernin telakan informaatiojärjestelmää tavoitteenaan vähentää kokonaisvaltaista monimutkaisuutta, jotta telakka voitaisiin helpommin mallintaa tietojenkäsittelysovellukseen. Hänen strategianaan oli, että suuri ja monimutkainen järjestelmä ositettaisiin useisiin modulaarisiin osatekijöihin. Nykyään tämä menetelmä tunnetaan toimintojen osituksena (Separation of Concerns, SoC). Seuraamalla tätä strategiaa Reenskaug kuvaa yleisen rungon, jonka tarkoituksena on mahdollistaa uuden informaatiojärjestelmän muovaaminen, joka soveltuu olemassa oleviin menetelmiin organisoida asioita. Tuloksena oli saavuttaa järjestelmä, joka on riittävän joustava sopeutumaan muutoksiin organisaatiossa. Hän myös tuo esille useita päävaatimuksia järjestelmän kehittämiselle yleisen rungon mukaan. Alla on listattuna toiminnallisia ominaisuuksia, joita tulisi hakea jotta runko olisi mahdollinen:

1. Haluamme rakentaa ihminen-kone järjestelmän, jossa automaattisesti tuotettuja tuloksia voidaan muokata manuaalisesti ja järjestelmä on kykenevä muokkaamaan tulosta manuaalisesti syötetyn tiedon mukaan.

2. Vastuut, päätösvalta ja pätevyys jakaantuu eri ryhmien kesken telakalla. Koko tietojenkäsittelyjärjestelmä jaetaan osajärjestelmiin. Osajärjestelmillä on jokaisella oma siihen liitetty vastuualue. Näitä osajärjestelmiä hallinnoivat kyseiset ryhmät, jotka myös jatkokehittävät sitä.

3. Alijärjestelmän täytyy olla läpinäkyvä siitä vastuussa olevalle ryhmällä, jotta sen omistajat voivat täysin ymmärtää järjestelmän käyttäytymisen ja kehittää sitä.

4. Alijärjestelmien täytyy olla myös avorakenteisia, jotta ne voidaan liittää toisiin alijärjestelmiin, jotka kuuluvat samalle tai toiselle vastuuryhmälle.

5. Koko järjestelmän täytyy pystyä jatkuvaan kasvuun. Siihen täytyy voida liittää uusia alijärjestelmiä, eikä olemassa olevien alijärjestelmien muutos saa vaikuttaa koko järjestelmään. Sen täytyy olla liitettävissä muihin järjestelmiin, ilman että se järisyttää kokonaisjärjestelmää. Vaikeaa siirtymää sukupolvien välillä ei voida enää sallia.

Huolimatta siitä, että Trygve Reenskaug loi perustuksen MVC:lle, sen toteuttamista viivästytti olio-ohjelmointikielen puuttuminen. Vuosia myöhemmin Trygve Reenskaug pääsi käsiksi maailman ensimmäiseen mikrotietokoneeseen Palo Altossa.

Pääsy Xeroxin Palo Alton Tutkimuskeskukseen (PARC, Palo Alto Research Center) vuonna 1978 auttoi Reenskaugia samaistumaan hajautetun järjestelmän loppukäyttäjään. Yhdessä kollegoidensa kanssa tohtori Reenskaug suunnitteli MVC:n ratkaisuna suurten ja monimutkaisten tiedostojen hallinnointiin. Heidän tehtävänsä oli samankaltainen kuin Reenskaugin vuonna 1973 määrittelemä, vähentää järjestelmälle rajapinnoissa esitettävää sisäistä monimutkaisuutta. Tähän aikaan alijärjestelmänä ymmärrettiin käyttöliittymä.

Jotta tämä tehtävä saataisiin täytettyä, täytyi kuvata uusi runko. Trygve Reenskaug sanoo, että vaikeinta oli löytää hyvät nimet eri arkkitehtuurikomponenteille. Ensimmäiset nimet olivat Malli-Näyttö-Muotoilija (Model-View-Editor), josta tuli Asia-Malli-Näyttö-Muotoilija (Thing-Model-View-Editor) ja sittemmin vuoden 1979 jälkeen vakiintunut Malli-Näyttö-Ohjain (Model-View-Controller). Termit asia, näyttö, malli ja muotoilija voidaan määritellä seuraavasti:

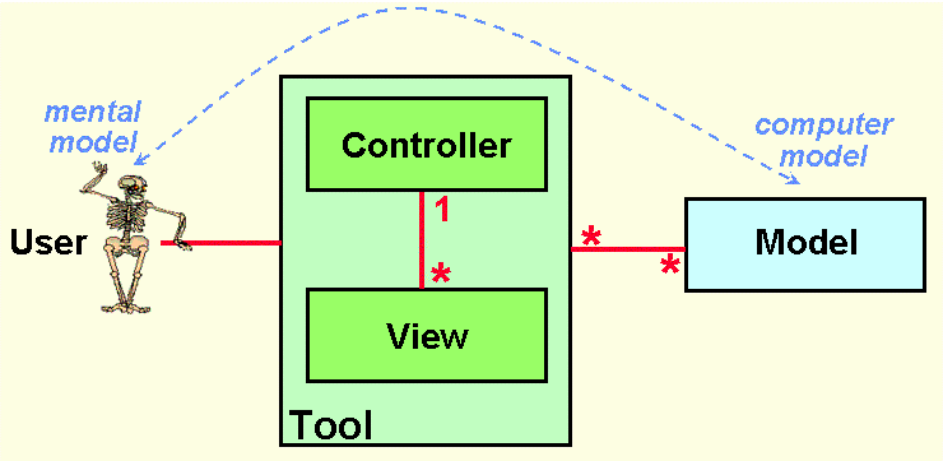
1. Asia (thing) on reaalimaailmankäsite, joka on käyttäjän mielenkiinnon kohde.

2. Malli (model) on abstraktion dynaaminen tiedon esitysmuoto tietojenkäsittelyjärjestelmässä. Abstraktiolla tarkoitetaan tässä asiaa.

3. Näyttö (view) on kuvallinen esitystapa mallista.

4. Muotoilija (editor) on rajapinta käyttäjän ja näytön välillä, myöhemmin tunnettu ohjaimena (controller).

Ensisijainen tarkoitus MVC:llä oli toimia siltana inhimillisen käyttäjän mentaalisen mallin ja digitaalisen mallin, joka esiintyy tietokoneessa välillä. Ideaali MVC ratkaisu tukee käyttäjän illuusiota nähdä ja manipuloida kohdealueen tietoa suoraan. Rakenne on hyvin käytännöllinen, jos käyttäjän tarvitsee nähdä sama mallialkio samanaikaisesti eri viitekehyksessä ja/tai eri näkökulmasta. Alla oleva kuva havainnollistaa tätä (kuva 13). Kuvassa esitetty Tool on ohjaimen ja näytön kokoonpano, jossa ohjain käsittelee tähän kokoonpanoon kohdistuvat pyynnöt.



Kuva 13. Käyttäjän mentaalinen malli vs. tietokoneen malli (Reenskaug 1979)

Näistä varhaisista prototyypeistä huolimatta MVC:n kolminaisuus nousi esiin SmallTalk-80 ohjelmointikielessä. Kuten jo yllä mainittiin malli oli abstraktio reaalimaailman käsitteestä, näyttö visuaalinen esitystapa mallista ja ohjaimen välityksellä käyttäjä saattoi kommunikoida mallin kanssa. Kaikki kolminaisuuden osat ovat vuorovaikutuksessa toistensa kanssa, joten tässä vaiheessa kerrostuneisuutta ei ollut.

Trygve Reenskaugin lähdettyä Xerox PARC:sta hänen työtään jäivät jatkamaan Jim Althoff ja Dan Ingalls. Tämän varhaisen työn vuoksi käyttöliittymien kehitys Smalltalk-80 oli hyvin pelkistettyä. Kuitenkin MVC-malli on tänäkin päivänä tehokas ja soveltuva ratkaisu käyttöliittymäkomponenttien suunnitteluun.

Myöhemmin Trygve Reenskaug on sanonut: ”MVC-paradigmalla on enemmän merkityksiä kuin ymmärsinkään vuonna 1979. Työskentelen mallikielen

(pattern language) parissa selvittääkseni eri näkökulmia, tämä on vasta ensimmäinen hahmotelma. Tätä mallia pitäisi kehittää useamman tutkijan toimesta, ei vain nykyisen yksittäisen" (Reenskaug 1979).

3.6 Yhteenveto MVC:stä

MVC on siis laajalti käytetty arkkitehtoninen ratkaisu, joka on helposti omaksuttavissa myös aloittelevalle sovelluskehittäjälle. Vaikka alkuperäinen ajatus MVC:stä juontaa juurensa jo 70-luvun alkuun, eivät uudet ohjelmointikielet ja -tekniikat ole suuresti vaikuttaneet sen rakenteeseen. Onkin hämmästyttävää huomata, kuinka jo noin varhaisessa vaiheessa on kyetty tuottamaan hyvin täsmällisiä spesifikaatioita siitä kuinka, järjestelmän tulisi toimia. Alla vielä Trygve Reenskaugin listattu tiivistelmä järjestelmän toiminnallisista vaatimuksista vuodelta 1973:

1. yksinkertaistettu manuaalinen ohjelman ohittaminen
2. jaollinen modulaarisiin alijärjestelmiin
3. alijärjestelmien toiminnallinen läpinäkyvyys
4. alijärjestelmien moduulien yhdistettävyys
5. kokonaisjärjestelmän kyky jatkuvaan kasvuun.

Ensimmäistä kohtaa lukuun ottamatta, koska se on nykyään oletuksena kaikkialla, kaikkien jäljelle jäävien neljän kohdan voidaan todeta edelleen olevan ajankohtaisia. Myös suunnittelumallien käyttö tähtää kohtiin neljä ja etenkin viisi. Suunnittelumallit auttavat kehittämään järjestelmän toimivista kokonaisuuksista ja vaikuttivat tämän kautta järjestelmän kykyyn olla kehitettävissä. Seuraavaksi tutustutaan lähemmin suunnittelumalleihin.

4 SUUNNITELUMALLIT

Christopher Alexander et al. määritelmää malleille, rakennusten ja kaupunkien suunnitteluun vuodelta 1977 lainataan hyvin usein ohjelmistotekniikan

suunnittelumalleista puhuttaessa, joten onkin hyvä palauttaa mieleen alkuperäinen määritelmä.

> ”*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it same way twice” (Christopher Alexander et al. 1977 p.x).*

Niin ikään Gamma et al. mukaan tämä on yhtä hyvin sovellettavissa oliopohjaisiin suunnittelumalleihin.

Suunnittelumallit ovat yleensä melko korkean abstraktiotason kuvauksia. Ne parantavat olemassa olevien järjestelmien dokumentaatiota ja ylläpidettävyyttä. Ne tarjoavat yksiselitteisen määritelmän luokkien ja olioiden suhteista ja käyttötarkoituksesta. Selkeästi määriteltyinä ne pystytään uudelleen käyttämään muodostaen niistä hyväksi havaittuja ratkaisuja ja arkkitehtuureja. (Gamma et al. 1994: 2.) Tosin tarkastelun näkökulma vaikuttaa siihen, mitä pidetään mallina. Esimerkkejä malleista, jotka eivät ole suunnittelumalleja, on muun muassa hajautustaulu (hash table) ja linkitetty lista (linked list). Suunnittelumalleiksi ei myöskään huomioida kokonaisia sovelluksia tai osajärjestelmiä koskevia suunnitteluratkaisuja. (Gamma et al. 1994: 3.)

Gamma et al. (1994: 3) määrittelee kirjassaan Design Patterns: Olioohjelmoinnin suunnittelumallit, suunnittelumallin neljä keskeistä osaa: nimi, ongelma, ratkaisu ja seuraukset.

1. Mallin nimi

   Mallin nimi on lyhyt sanallinen kuvaus ongelmasta, ongelman ratkaisusta tai ratkaisun seuraamuksista. Nimellä kartutetaan sanastoa, ja se auttaa suunnittelua korkeammalla abstraktiotasolla. Yhteisen sanaston luominen helpottaa mallien käyttämistä.

2. Ongelma

   Ongelman ja sen ympäristön avulla kuvataan mallin soveltamiskohteita. Sen kautta voidaan kuvata suunnitteluongelmia, joka voi olla esimerkiksi

kuinka muotoillaan algoritmista olio. ”Se voi tuoda esiin luokkia ja oliorakenteita, jotka ovat oireita joustamattomasta suunnittelusta” (Gamma et al. 1994: 3). Jotta mallia voitaisiin soveltaa, ongelman alle voi olla koottuna ehtoja, joiden on täyttyessä mallin soveltaminen on järkevää.

3. Ratkaisu

   Kuvaus elementeistä, joista suunnitteluratkaisu muodostuu. Se sisältää myös elementtien vastuualueet, niiden väliset suhteet ja niiden keskinäisen yhteistyön. Elementeillä tarkoitetaan luokkien ja olioiden yleisrakennetta. Ratkaisuna annettava malli toimii runkona, jota voidaan soveltaa. Se ei siis ole konkreettinen toteutus. Tarkoituksena on antaa kuvaus ongelmasta ja siitä, millaisilla elementeillä se on mahdollista ratkaista.

4. Seuraukset

   Auttavat arvioimaan mallin hyviä ja huonoja puolia sekä vaihtoehtoisia ratkaisuja. Seurauksien voidaan yleensä olettaa liittyvän ohjelmointikieleen ja toteutukseen sekä muistitilaan ja suoritusaikaan. Seurauksissa selostetaan mallin vaikutus ratkaisun muunneltavuuteen, laajennettavuuteen ja siirrettävyyteen.

Suunnittelumallit voidaan Gamma et al. mukaan luokitella kahden kriteerin mukaan: tarkoituksen, joka kuvaa mallin toimintaa, ja kohteen, joka kertoo liittyykö malli ensisijaisesti luokkiin vai olioihin. Ensimmäisen kriteerin mukaiset mallit kohdistuvat joko luontiin (creational pattern), rakenteeseen (structural pattern) tai käyttäytymiseen (behavioral pattern). Nimensä mukaisesti luontimallit käsittelevät olioiden luontiprosessia. Rakennemallien avulla käsitellään luokkien ja olioiden koosteita. Luokkien ja olioiden vuorovaikutuksia ja niiden vastuiden jakoa käsitellään käyttäytymismallien kautta. Toisen kriteerin mukaan luokkamallit käsittelevät luokkien ja niiden aliluokkien välisiä suhteita. Nämä suhteet kiinnitetään periytymisen kautta, joten ne ovat staattisia eli käännösaikaisia. Oliomallit käsittelevät olioiden välisiä suhteita, joita voidaan muuttaa ajonaikaisesti ja jotka ovat dynaamisempia. (Gamma et al. 1994: 10–11.)

Luontimallit (Creational pattern)

Luontimallien tarkoituksena on auttaa ratkaisemaan ongelmia, kun luodaan oliota ympäristössä, jossa liittymänä on joukko abstrakteja luokkia tai kun luokalla on vain yksi ilmentymä, jonka täytyisi olla muiden luokkien saatavilla. (Immonen 2002: 6) Luontimallien kaksi yhteistä piirrettä ovat järjestelmän käyttämien todellisten luokkien kapselointi ja luokkien ilmentymien todellisen syntymekanismin piilottaminen. Ohjelmisto tietää vain olioista, ei niiden takana olevista luokista. (Wikman 2001: 12.)

Rakennemallit (Structural pattern)

Rakennemallien kuvataan ratkaisuja, joiden avulla luokkia ja olioita yhdistelemällä muodostetaan suurempia rakenteita ja yhdistämällä niitä saadaan aikaan uusia toimintoja. Rakennemallit auttavat esimerkiksi muistinkäytön vähentämisessä, ja niiden avulla voidaan ratkaista tehokkuuden lisäämiseen liittyviä ongelmia, etenkin luotaessa useita olioita tai sovitettaessa eri järjestelmiä yhteen. (Gamma et al. 1994: 137; Immonen 2002: 6–7.)

Käyttäytymismallit (Behavioral pattern)

Käyttäytymismallit edustavat näkökulmaa, jonka mukaan suunnittelutyössä ei pidä keskittyä siihen, miten kontrollin kulkua hallitaan vaan miten oliot liitetään toisiinsa (Gamma et al. 1994: 221). Mikäli ohjelmassa on toimintoja, jotka muuttuvat usein löytyy käyttäytymismalleista keinot kapseloida toiminnot yhteen olioon. Ne myös käsittelevät algoritmeja ja olioiden välisten vastuiden määrittelyä. Vastuiden lisäksi käyttäytymismallit tarjoavat suunnitteluapua olioiden mallintamisen lisäksi myös niiden väliseen kommunikaatioon. (Gamma et al. 1994:226.) Tällä kuten suunnittelulla yleensäkin pyritään eli muutosten helpompaan toteutettavuuteen ja hallittavuuteen.

Alla esitellään kaksi suunnittelumallia yleisellä tasolla. Esiteltävät mallit on valittu ajatellen MVC:n toteuttamista niiden avulla.

4.1 Rakennemalli: Rekursiokooste (Composite)

Rekursiokooste-suunnittelumallia voidaan käyttää hierarkkisten rakenteiden esittämiseen. Hierarkkinen rakenne koostuu komponenteista, joista kukin voi olla joko yksittäinen komponentti eli lehti tai hierarkkinen rakenne itsessään eli solmu. Tällaisen rakenteen läpikäyminen voi olla hankalaa, varsinkin jos rakenteessa on paljon erilaisia komponentteja. Usein lehti- ja solmukom-

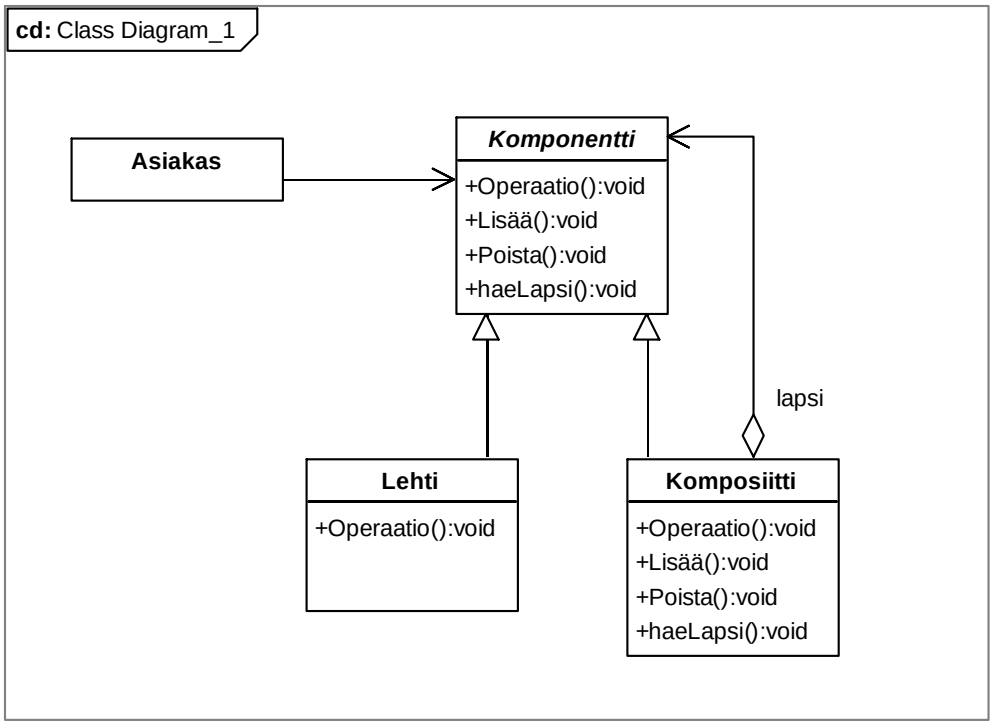
ponenteilla on yhteisiä operaatioita, jotka olisi hyvä toteuttaa vain yhdessä paikassa eikä erikseen jokaisessa komponentissa. (Immonen 2002: 61.)

Kooste-suunnittelumalli ratkaisee edellä esitetyn ongelman määrittelemällä jokaiselle rakenteessa olevalle komponentille yhteisen abstraktin yliluokan, joka toteuttaa yhteiset operaatiot. Lisäksi jokaisessa aliluokassa voidaan luonnollisesti toteuttaa vain kyseiselle aliluokalle tarpeelliset operaatiot. Mallin avulla voidaan siis esittää olio hierarkkisesti toisista olioista koostuvana siten, että koosteolioita ja niiden osaolioita voidaan käsitellä samalla tavalla. (Immonen 2002: 62.)

Rekursiokoosteen soveltuvuus

Kooste-malli soveltuu muun muassa seuraaviin tilanteisiin, joissa halutaan esittää hierarkkista rakennetta olioista, jotka koostuvat osaolioista. Malli soveltuu hyvin tapauksiin, jossa asiakasohjelman kannalta on tärkeää olla erottelematta olioiden koosteita ja yksittäisiä olioita. (Immonen 2002: 62.)

Rekursiokoosteen toteutus



Kuva 14. Rekursiokooste-mallin rakenne (Gamma et al. 1994: 164)

Kooste-malli rakentuu tietyn avainabstraktion ympärille. Abstraktio (kuva 14) määrittelee rajapinnan kullekin komponentille ominaisille toiminnoille sekä ne operaatiot, jotka ovat yhteisiä kaikille alikomponenteille. Kullekin alikomponentille on oma aliluokkansa (*Leaf*), jotka määrittelevät itselleen ominaiset toimintonsa.

Kooste-malli muodostuu seuraavista osallistujista: Komponentti määrittelee rajapinnan mallissa mukana oleville olioille ja lapsikomponenteille, niiden käsittelemiseksi. Komponentti myös toteuttaa kaikkien luokkien yhteiset operaatiot. Lehti (leaf) määrittelee jokaisen primitiiviolion toiminnan. Lehdellä ei voi olla lapsikomponenttia. Komposiitti määrittelee vanhempien toiminnan ja ylläpitää rakennetta johon on tallennettu lapsikomponentit. Sen tehtävänä on myös määritellä lapsikomponentteihin liittyvät operaatiot, joihin kuuluu muun muassa lapsiolion luominen. Kooste-mallissa asiakas (client) käsittelee siinä mukana olevia olioita komponentti-luokan määrittelemän rajapinnan mukaan. (Immonen 2002: 63.)

Komponentti-luokassa määritellään rajapinta, jota asiakasohjelmat käyttävät keskustellakseen komposiitti-rakenteen olioiden kanssa. Keskustellakseen olion kanssa asiakkaan ei tarvitse tietää minkä tyyppisen olion kanssa se on tekemisissä. Jos asiakas keskustelee lehden kanssa, hoitaa lehti itse asiak-

kaalta saamansa pyynnöt. Mikäli pyynnöt kohdistuvat komposiitti-rakenteeseen, se välittää ne edelleen lapsikomponenteille. Komposiitti suorittaa itse pyynnön toteuttamiseen tarvittavat ennen tai jälkeen pyynnön välitystä. (Immonen 2002: 63.)

Komposiitti-luokan rakenne mahdollistaa yhtenäisen käsittelyn olioille. Lisäksi yhteinen rajapinta komponentti ja komposiitti-luokilla antaa komposiitti-luokalle mahdollisuuden luoda rekursiivisesti uusia komponentti-olioita. Näin saadaan aikaan puurakenne, jossa solmuina on komponentti-olioita ja lehtinä lehti-olioita. (Immonen 2002: 63.)

Kooste-mallia toteutettaessa on otettava seuraavanlaisia asioita: oliopuun läpikäynti, luodaanko jokaiselle primitiivioliolle täysin oma olio, myös lapsikomponentteihin liittyvät toimenpiteet täytyy ratkaista. Lisäksi täytyy löytää paras tietorakenne lapsikomponenttien tallentamiseen. (Immonen 2002: 63.)

Kooste ominaisuuksia

Kooste-suunnittelumalli määrittelee luokkahierarkian, joka koostuu sekä primitiivioliosta että primitiivioliosta rekursiivisesti muodostetuista koosteista. Asiakasohjelma ei näe eroa kooste- ja primitiivioliosien välillä. Samalla asiakasohjelman toteuttaminen yksinkertaistuu ja helpottuu, kun sama osa koodia suorittaa kaikki pyynnöt kohdeolion tyypistä riippumatta. Uusien ominaisuuksien lisääminen järjestelmään on helppoa, koska lisäys ei vaikuta järjestelmään muuten kuin lisäämällä luokkarakenteeseen uuden primitiiviluokan. Uudet primitiiviluokat toimivat luokkarakenteessa sellaisenaan. (Immonen 2002: 67)

4.2 Käyttäytymismalli: Tarkkailija (Observer)

”Tarkkailija-suunnittelumallissa ajatellaan, että maailma koostuu kahdenlaisista olioista: *subjekteista*, joita tarkkaillaan, ja *tarkkailijoista,* jotka tarkkailevat. Jokaisella subjektilla voi olla useita tarkkailijoita, mutta subjekti ei tunne niiden laatua. Kun tarkkailija haluaa ryhtyä tarkkailemaan tiettyä subjektia, se ilmoittaa tälle siitä. Aina, kun subjektin tila muuttuu, se ilmoittaa tilamuutoksesta kaikille tarkkailijoilleen. tarkkailijat reagoivat muutokseen kukin omalla tavallaan. Usein systeemeissä aiheuttaa ongelmia tilanne, jossa tieto muuttuu yhdessä osassa ja muut osat ovat riippuvaisia tästä tiedosta. Juuri

tämän ongelman ratkaisemiseksi tarkkailija-suunnittelumalli on kehitetty. Se auttaa pitämään yhdessä toimivien osien tilat tasapainossa.” (Immonen 2002: 15–16.)

Tarkkailijan soveltuvuus

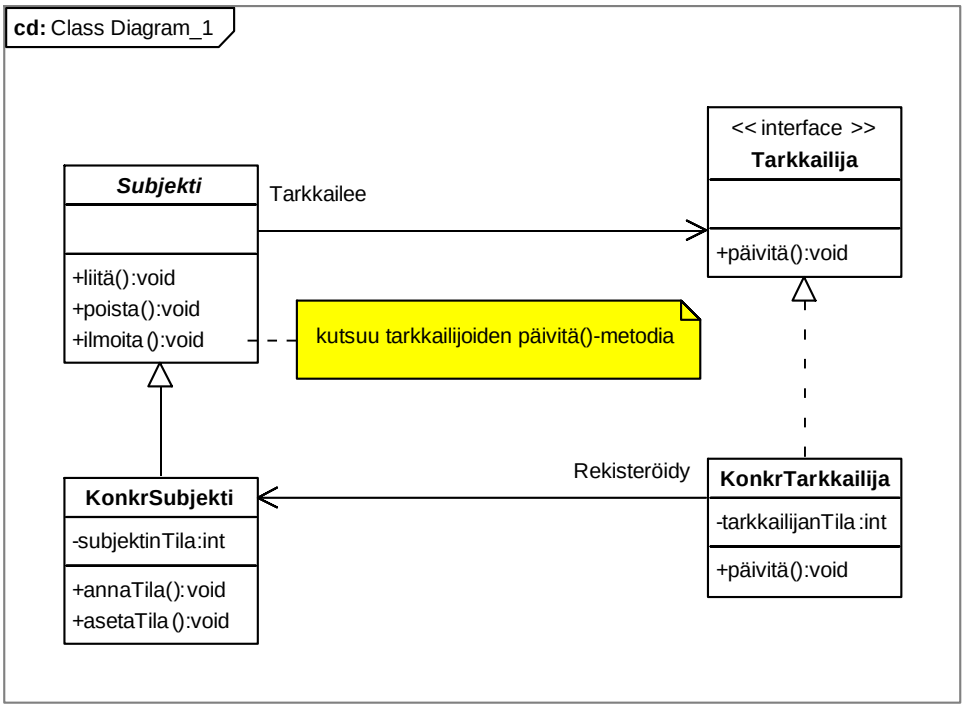
Tarkkailija-malli soveltuu erityisesti tilainteisiin, joissa oliot, joilla on keskeinen riippuvuus halutaan toteuttaa mahdollisimman riippumattomasti. Immonen (2002:16) suosittelee tarkkailijaa käytettävän seuraavanlaisissa tapauksissa:

- Kun käsiteltävällä abstraktiolla on kaksi kokonaisuutta, jotka riippuvat toisistaan. Kapseloimalla kumpikin osa omaksi oliokseen, niiden vaihtaminen ja uudelleenkäyttö itsenäisinä kokonaisuuksina tulee myös mahdolliseksi.

- Kun tietyn olion muutoksesta aiheutuu muutostarve myös muihin olioihin. Ei myöskään tiedetä niiden olioiden määrää, joihin muutostarve kohdistuu.

- Kun olion tulee ilmoittaa tilastaan muille olioille tietämättä tarkasti, mitä nämä oliot ovat.

Tarkkailijan toteuttaminen



Kuva 15. Tarkkailija-suunnittelumallin rakenne (mukaillen Gamma et al. 1994: 294)

Tarkkailija-malliin osallistuu: subjekti, joka on abstraktiluokka sisältäen tarkkailija-olioiden perusoperaatiot. Subjektilla on tiedossa kaikki sitä tarkkailevat tarkkailijat, ja se tarjoaa rajapinnan tarkkailija-olioiden liitä- ja poista-operaatioille. Tarkkailija on tarkkailijoiden rajapinnan kuvaava abstrakti luokka, joka sisältää päivitä-operaation. Se myös määrittelee rajapinnan niille olioille, joille tiedotetaan subjektin tilan muuttumisesta. Konkreettinen subjektin tehtävänä on tiedottaa tarkkailijoilleen tilansa muutoksesta. Se on subjektin konkreettisen aliluokan ilmentymä. Myös konkreettinen tarkkailija on konkreettisen aliluokan ilmentymä. Konkreettisella tarkkailijalla on oma päivitä-operaati,o ja sen tehtävänä on ylläpitää viittausta konkreettiseen subjektiolioon. Jos konkreettisen tarkkailijan tilan täytyy pysyä yhtäläisenä subjektin kanssa, se tallentaa tämän tilan.

”Tarkkailija-suunnittelumallia voidaan käyttää hyväksi esimerkiksi käyttöliittymäarkkitehtuurissa. Tällöin subjektina on itse sovellus tai sen osa ja tarkkailijana toimii tämän näkymä näytöllä. Kun sovelluksen tila muuttuu, kaikille sen näkymille ilmoitetaan muutoksesta. Vaikka eri elementit riippuvatkin toisistaan, niiden ei kuitenkaan tarvitse tuntea toisiaan. Sovelluksen ei puolestaan tarvitse tuntea näyttöjen tarkempaa laatua eikä toteutustapaa. Tarvitaan ainoastaan tarkkailijarajapinta (operaatio), jolla muutoksesta ilmoitetaan

käyttöliittymäelementeille, jotta ne voisivat päivittää näkymänsä.” (Immonen 2002: 17–18)

Tarkkailija-malli siis ratkaisee ongelman, jossa oliojoukon täytyy reagoida jos yksikin olio muuttaa tilaansa. Ongelma yleisyyden takia Javassa tarkkailija-suunnittelumallin tarjoama ratkaisu on otettu osaksi sen luokkakirjastoa. Observable-luokan tehtävänä on ylläpitää tietoa niistä elementeistä, jotka saavat tiedon muutoksen tapahduttua. Tarkkailtava objekti kutsuu Observable-luokan notifyObservers()-metodia, joka ilmoittaa tarkkailijoille tilansa muuttumisesta ja Observer-rajapintaluokan update()-metodi päivittää tarkkailijoiden tilan.

”Observable-luokassa on lippukohta, joka osoittaa onko muutosta tapahtunut. Suurimman osan työstä tekee notifyObservers(), joka ei kuitenkaan tee mitään, jos lippukohdan muutosta ei ole asetettu. Pelkkä yksinkertainen Observable-olion käyttäminen ei kuitenkaan riitä hoitamaan muutoksia, vaan sen luokan on perittävä Observable-luokka ja jossakin kohtaa tässä periväs-sä luokassa on kutsuttava Observable-luokan setChanged()-metodia. Tämä jäsenfunktio asettaa muutoksen lippukohtaan, mikä tarkoittaa sitä, että kun kutsutaan notifyObservers()-metodia, kaikki tarkkailijat saavat tiedon muutoksesta.” (Immonen 2002: 18)

Tarkkailijan ominaisuuksia

Immonen (2002: 19–20) selittää tarkkailija-mallin ominaisuuksia erikoistyös-sään: ”Tarkkailija-suunnittelumalli sallii subjektien ja tarkkailijoiden riippumattoman vaihtelun. Subjekteja voidaan uudelleen käyttää ilman, että niiden tarkkailijoita tarvitsisi käyttää. Sama pätee myös toisinpäin. Malli sallii myös lisätä tarkkailijoita ilman, että subjektia tai muita tarkkailijoita tarvitsisi muuttaa. Muita ominaisuuksia:

- Subjektin ja tarkkailijan välillä on abstrakti kytkentä. Tämä tarkoittaa sitä, että subjekti tietää vain sen, että sillä on tarkkailijoita, joista jokainen käyttää tarkkailija-luokan yksinkertaista rajapintaa. Subjekti ei sen sijaan tiedä yhdenkään tarkkailijan luokkaa. Kytkennän tulee olla myös mahdollisimman vähäistä. Kun kytkentä ei ole tiukkaa, subjekti ja tarkkailija voivat kuulua systeemin eri abstraktiokerroksiin, eli alemman tason subjekti voi esimerkiksi informoida ylemmän tason tarkkailijaa.

- Subjektin lähettämälle tiedolle ei tarvitse määritellä saajaa. Tieto lähetetään automaattisesti kaikille olioille, joille sillä on merkitystä. Koska subjektille ei ole merkitystä kuinka paljon sillä on tarkkailijoita, tarkkailijoiden määrää voidaan vaihdella missä vaiheessa tahansa.

- Koska tarkkailijoilla ei ole lainkaan tietoa toistensa olemassaolosta, ne eivät ymmärrä miten suuri kokonaisvaikutus subjektin muuttamisesta voi aiheutua. Huonosti määritellyt ja ylläpidetyt riippuvuudet voivat myös aiheuttaa ongelmia päivityksissä. Nämä päivitykset voivat olla hyvin vaikeasti jäljitettävissä, koska tavallinen päivitysmenetelmä ei sisällä tietoa siitä, mikä subjektissa on muuttunut.

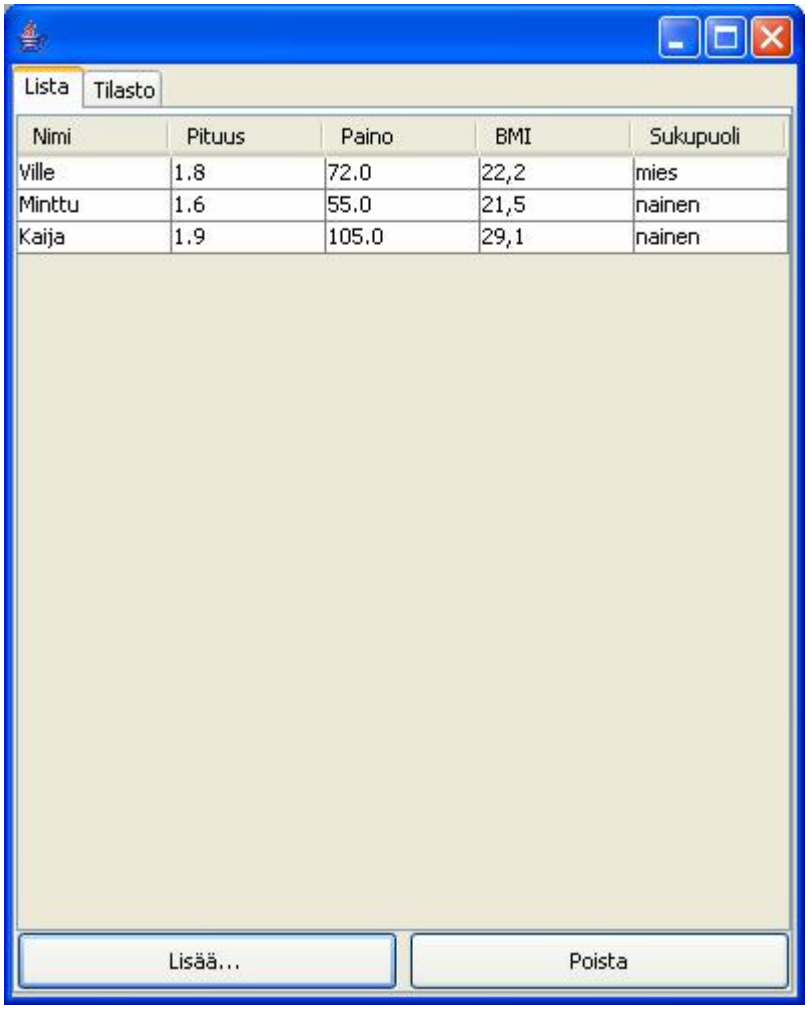
- Malli voi olla kaksisuuntainen.”

5 PAINOINDEKSISOVELLUS

Käytännön toteutukseksi tähän työhön on valittu sovellus, jossa sovelletaan aiemmin läpikäytyä asioita. Tarkoituksena on tutkia, kuinka MVC-kerrosarkkitehtuuri ja siihen liittyvät suunnittelumallit toimivat käytännössä. Painoindeksisovellus on toteutettu käyttäen Borlandin JBuilder 2006 Enterprise Edition -ohjelmointityökalua. JBuilder-työkalun etuna on muun muassa Javan API-dokumenttien automaattinen generointi. Tässä työssä ne löytyvät liitteestä 4. UML-kaaviot, on toteuttettu Poseidon mallinnustyökalun standard edition trial -versiolla. Koko sovelluksen UML-kaavio löytyy liitteestä 3. Varsinaisen ohjelmoinnin tukena on käytetty Mirja Immosen erikoistyötä ”Suunnitelumallit” (Immonen 2002) , jossa on esitetty Java-kielellä toteutettuja esimerkkejä suunnittelumalleista. Aiemmin toteutettu painoindeksilaskin on toiminut tukena tässä toteutuksessa. Toteutukseksi on valittu yksinkertainen painoindeksilaskin. Koska kyseessä on yksinkertainen sovellus, on siitä jätetty pois ominaisuuksia, jotka oikeassa sovelluksessa olisi pakollisia esimerkiksi käyttäjien tietojen tallennus tietokantaan tai tiedostoon.

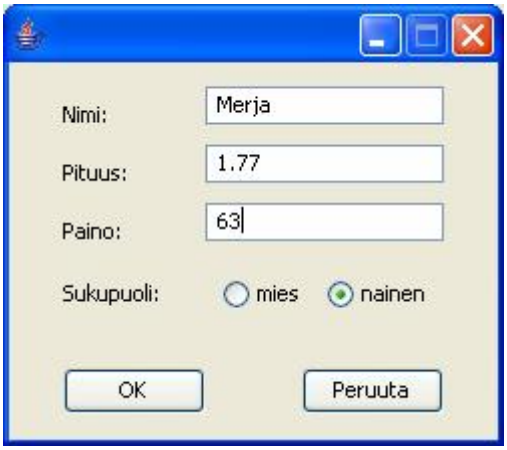
5.1 Käyttöliittymä

Painoindeksisovelluksessa on pääikkuna, jossa on kaksi välilehteä. Ensimmäisellä välilehdellä, nimeltään listanäkymä (kuva 16), voidaan esittää käyttäjien tietoja taulukossa. Lisäksi listanäkymä-näytöllä voidaan lisätä ja poistaa tietoja.



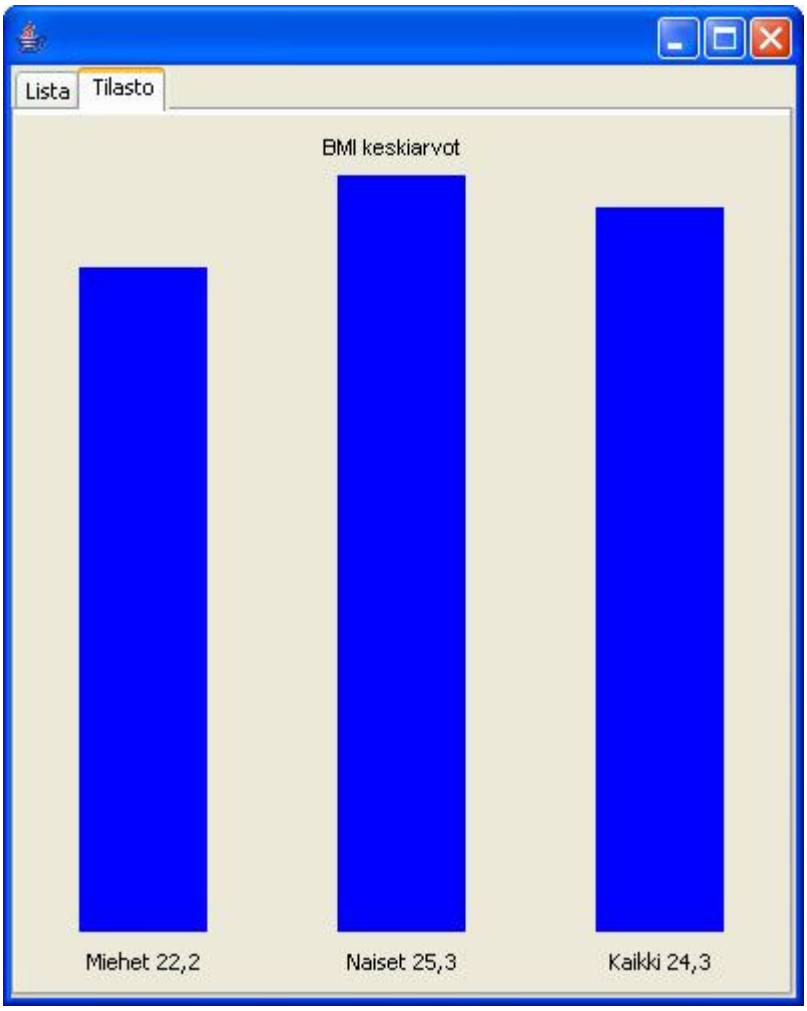
Kuva 16. Painoindeksisovelluksen listanäkymä

Lisäys listanäkymään tapahtuu Lisää-painikkeen kautta, joka tuo esiin uuden ikkunan (kuva 17).



Kuva 17. Listanäkymän lisää-ikkuna

Toisella välilehdellä, jota kutsutaan tilastonäkymäksi (kuva 18), esitetään pylväsdiagrammi, jossa esitetään mies-, nais- ja kaikkien käyttäjien painoindeksien keskiarvot. Toisen välilehden tarkoituksena on toteuttaa MVC:n ajatus, jossa mallissa oleva tieto voidaan esittää eri tavoilla.



Kuva 18. Painoindeksisovelluksen tilastonäkymä

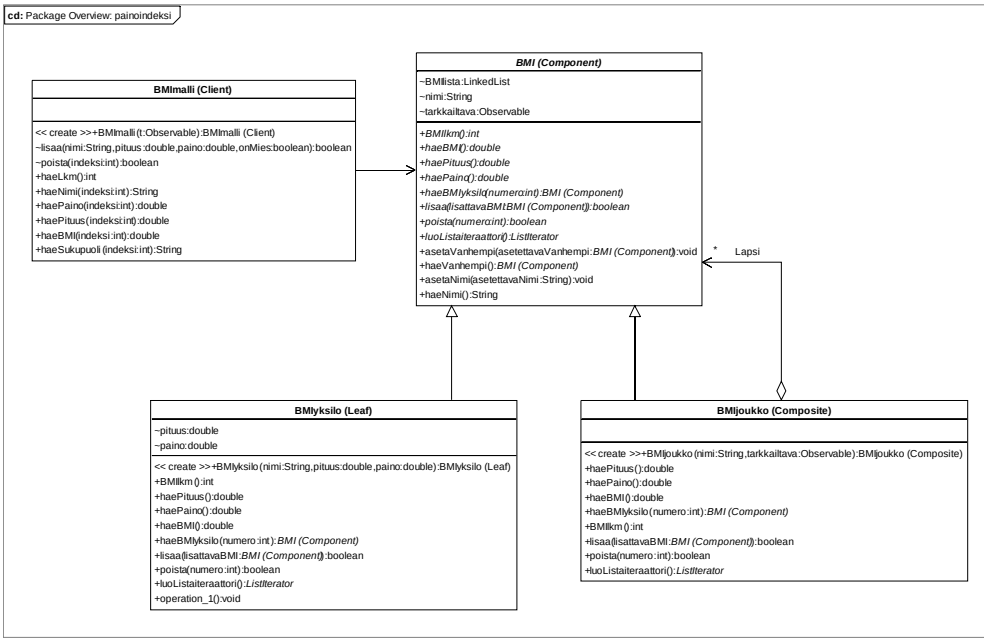
Tämän työn kannalta ei ole oleellista paneutua erityisesti sovelluksen ulkonäköön liittyviin seikkoihin, eikä asioiden pikkutarkkaan hiomiseen. Oleellista sen sijaan on esittää toimiva MVC-arkkitehtuuriin nojautuva sovellus, jossa on voitu hyödyntää suunnittelumalleja.

5.2 Arkkitehtuuri

Tässä työssä on aloitettu sovellussuunnittelun perusteista, päätyen eri vaiheiden kautta näihin viimeisiin vaiheisiin, joissa aiemmin tutkittua käytetään hyväksi varsinaisessa sovelluksessa. MVC-kerrosarkkitehtuurin toteutus painoindeksisovelluksessa esitetään liitteessä 3. Sovellus luo mallin, ohjaimen ja näkymän.

5.3 Malli

Painoindeksisovelluksessa käytetään MVC-arkkitehtuurin mallin luomiseen rekursiokoostetta (kuva 19).



Kuva 19. Malli toteutettuna rekursiokooste-suunnittelumalli avulla

BMImalli (Client)**:** Sovellus luo BMImalli-luokan ilmentymän. Konstruktori luo BMIjoukko-luokan ilmentymät kaikki, miehet ja naiset.

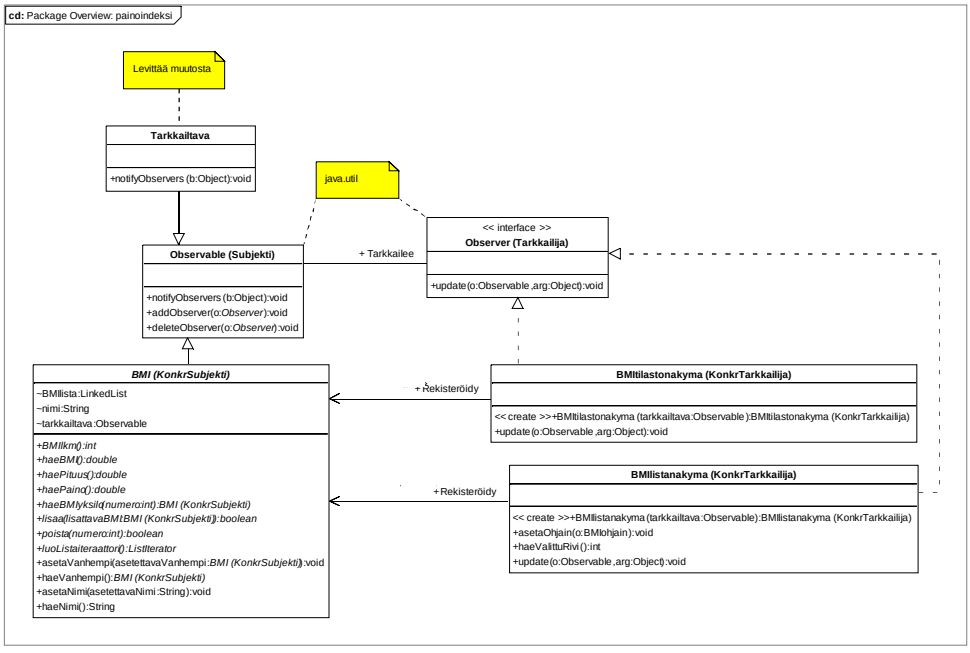
BMI (Component)**:** Määrittelee rajapinnan kooste-mallissa mukana oleville olioille. Toteuttaa kaikille luokille yhteiset operaatiot, joita painoindeksisovelluksessa ovat *asetaNimi(), asetaVanhempi()*, *haeNimi()* ja *haeVanhempi()*. Määrittelee myös rajapinnan lapsikomponenttien käsittelyyn ja hallintaan. Käytännössä tämä tarkoittaa, että BMI-luokka määrää aliluokille toteutettavaksi metodit *BMIlkm(), haeBMI(), haeBMIyksilo(), haePaino(), haePituus(), lisaa(), luoListaiteraattori()* ja *poista()* metodit.

BMIyksilo: (Leaf) Sisältää yhden henkilön tiedot, joita ovat nimi, pituus ja paino. HaeBMI()-metodi palauttaa henkilön painoindeksin, joka lasketaan pituus- ja paino-kenttien avulla. Henkilön sukupuoli määräytyy joukon mukaan, johon yksilö kuuluu. Ei voi sisältää lapsikomponenttia.

BMIjoukko (Composite)**:** Solmu sisältää painoindeksisovellukseen kuuluvien joukkojen, joita ovat *miehet*, *naiset* ja *kaikki*, tiedot. Joukko *kaikki* sisältää alijoukot *miehet* ja *naiset.* Yksilöt eli yksittäiset henkilöt kuuluvat joukkoihin *miehet* ja *naiset.* Tietoihin kuuluu joukkoon kuuluvien yksilöiden pituuksien, painojen ja painoindeksien keskiarvot. BMIjoukko määrittelee lapsikomponentteihin eli joukkoon tai yksilöön liittyvät operaatiot, kuten lapsiolion luonnin ja poiston.

5.4 Näkymät

Näyttöjen ulkoasut ovat esitelty luvussa 5.1 Käyttöliittymä. Näkymät-otsikon alla esitellään tarkkailija-suunnittelumallin toteutus MVC-arkkitehtuurissa se muodostaa linkin näkymien ja mallin välille (kuva 20).



Kuva 20. Tarkkailija-suunnittelumalli painoindeksisovelluksessa

BMInakyma: Sovellus luo BMInakyma-luokan ilmentymän. Luokan konstruktorissa luodaan BMIlistanakyma-, BMItilastonakyma ja BMIlisaa-luokkien ilmentymät (liite 3).

BMItilastonakyma (KonkrTarkkailija): Luo käyttöliittymään tilastovälilehden. Tilastonäkymässä esitetään miesten, naisten ja kaikkien henkilöiden painoindeksien keskiarvot pylväsdiagrammina. On tarkkailija-mallin konkreettinen tarkkailija. Se toteuttaa java.util-kirjaston Observer-rajapinnan.

BMIlistanakyma (KonkrTarkkailija): Luo käyttöliittymään listavälilehden. Listanäkymässä esitetään yksittäisten henkilöiden tiedot: nimi, pituus, paino, sukupuoli ja painoindeksi. Listanäkymän kautta käyttäjä voi lisätä ja poistaa henkilöitä. BMIlistanakyma on tarkkailija-mallin toinen konkreettinen tarkkailija, joka myös toteuttaa Observer-rajapinnan.

BMIlisaa: Lisää ikkunasta käyttäjä voi lisätä BMIyksilo-luokan ilmentymiä eli henkilöitä (ks. liite 3).

Observer (Tarkkailija): Osa java.util-kirjastoa. On rajapintaluokka, jolla on yksi jäsenfunktio update().

Observable (Subjekti): Abstrakti Observable-luokka on myös osa java.util-kirjastoa. Sisältää perusoperaatiot, kuten notifyObservers(), addObserver() ja deleteObserver(). NotifyObservers()-metodia kutsutaan muutoksen tapahduttua, ilmoittaa Observer-luokalle päivitystarpeesta. AddObserver()- ja deleteObserver()-metodit nimiensä mukaisesti lisäävät ja poistavat tarkkailijoita.

Tarkkailija: Perii Observable-luokan ja toteuttaa notifyObservers()-metodin, joka asettaa tarkkailtavien tilan muuttuneeksi ja siten levittää muutosta. Muutosilmoituksen jälkeen koodin suoritus jatkuu tarkkailijoiden update()-metodissa.

5.5 Ohjain

Painoindeksisovellus luo ohjaimen, joka sisältää painikkeiden tapahtumakuuntelijat. Kun käyttäjä painaa käyttöliittymässä lisää- tai poista-painiketta, ohjain välittää tiedon mallille, joka puolestaan lisää tai poistaa kyseisen henkilön tiedot malliin/mallista. Ohjain ei muokkaa käyttöliittymää, vaan käyttöliittymä päivittyy automaattisesti mallin tilan muuttuessa tarkkailija-suunnittelumallin mukaisesti.

6 JOHTOPÄÄTÖKSET

Arkkitehtuurien ja suunnittelumallien tarkoituksena on luoda rakenteita, jotka ennen toteuttamista tuovat esille mahdollisia puutteita ja ongelmaksi muodostuvia kohtia. Niiden tarkoituksena on myös tuottaa dokumentaatiota ohjelmistosta ja sen rakenteesta. Dokumentaatiota käytetään kommunikoinnin välineenä eri sidosryhmien välillä, sen avulla rakennetaan myös ohjelmiston toteutus.

MVC-kerrosarkkitehtuuri on looginen ja suhteellisen helposti ymmärrettävä ja toteutettava ohjelmiston rakenne. Sen avulla ohjelmistotekniikka vain vähän tunteva voi perehtyä esimerkiksi käyttöliittymien luomiseen käyttäen jäsenneltyä rakennetta.

Suunnittelumallien ymmärtäminen vaatiikin jo sitten hieman enemmän ohjelmointikokemusta. Niiden avulla ja ne hyvin tunteva suunnittelija voi helposti rakentaa hallittavan ja muutossietoisen sovelluksen. Oikotietä onneen näistä malleista ja ohjeista ei aloittelijalle yleensä ole. Suunnittelumalleihin perehdyttäessä onkin ehkä syytä hankkia tässäkin työssä laajalti apuna käytetty Gamma et al. Design Patterns (1994) -teos. Sillä tutkittaessa asiaa esimerkiksi verkosta päädytään ongelmaan; suunnittelumalleja alkaa olla paljon ja nimitystä suunnittelumalli saatetaan käyttää kovinkin ohuin perustein. Tästä myös Vuorinen (2005a) kirjoittaa artikkelissaan.

Alan terminologian kehittäminen koetaankin olevan yksi haasteellisimmista tehtävistä, tästä johtuen sanasto on värikästä ja terminologia laajaa. Aloittelijan onkin usein vaikea pysyä kärryillä kun samaa tarkoittavia termejä on useita tai termit eri kontekstissa tarkoittavatkin muuta.

Työssä on toteutettu kaksi sovellusta, painoindeksilaskin, joka toimii esimerkkinä MVC-arkkitehtuurista ja painoindeksisovellus, joka on painoindeksilaskinta laajempi sovellus. Se nojaa MVC-arkkitehtuuriin, ja sen malli on toteutettu rekursiokooste-suunnittelumallia käyttäen. Näyttöjen ja mallin välille on toteutettu linkki tarkkailija-mallia käyttäen.

Painoindeksisovelluksen toteuttamisessa törmäsin moniin ongelmiin, esimerkiksi suunnittelumallien käyttäminen osoittautui erittäin haasteelliseksi. Onneksi Gamma et al. (1994) kirjoittavat myös MVC-mallista ja sen toteuttamisesta suunnittelumallien avulla. Siitä ja Immosen (2002) Javalla toteute-

tuista käytännön esimerkeistä oli huomattava apu käytännön työssä. Vaikeuksista huolimatta käytännön toteutus kuitenkin valmistui. Lisäksi siinä täyttyy ainakin kaksi hyvän sovelluksen vaatimusta: muutossietoisuus ja laajennettavuus. Esimerkiksi jos halutaan poistaa näkymiä, ei malliin tarvitse koskea, eli sovellus on muutossietoinen. Haluttaessa taas lisätä näkymiä voidaan vain luoda uusi näkymä, jälleen koskematta malliin tämä tekee sovelluksesta niin ikään muutossietoisen, mutta myös laajennettavan.

Arkkitehtuureja ja suunnittelumalleja sovelletaan tapauskohtaisesti riippuen käyttötarkoituksesta. Ideaalisesti niiden avulla voidaan luoda puhtaita helposti muokattavia, laajennettavia ja ylläpidettäviä sovelluksia. Harvoin niitä kuitenkin tavataan täysin puhtaana. Arkkitehtuurien ja suunnittelumallien voidaan sanoa olevan laajalti sovellettavissa. Ei myöskään voida todeta ongelmiin olevan yhtä ainoaa oikeaa ratkaisua. Ohjelmistojen ja järjestelmien kehitystyötä tuntuu kuitenkin vaivaavan krooninen kiire. Tämä kiire vaikuttaa siihen, ettei henkilöillä ole aikaa perehtyä rakenteellisiin ratkaisuihin ja usein toteutukset nojaavatkin totuttuihin tapoihin, joiden jopa tiedetään olevan pitkällä tähtäimellä kestämättömiä. Asiaa ei kuitenkaan saa nähdä yksistään negatiivisena, toki löytyy yrityksiä ja henkilöitä, jotka ovat perehtyneet näihin rakenteisiin ja kykenevät tuottamaan laajennettavia ja ylläpidettäviä ohjelmistoja. Eli voidaan todeta, että ohjelmiston yleisten rakenteellisten ominaisuuksien ja suunnittelumallien ymmärtäminen antaa merkittävän edun ohjelmistotyöntekijöille.

Ironista kyllä ongelmat, joita kohtaamme tämän päivän hajautetuissa järjestelmissä, ovat samankaltaisia kuin kolme vuosikymmentä sitten. Eli ohjelmistotekniikan kannalta ajatellen ”pyörää” ei ehkä vielä olekaan keksitty tai ehkä se ei ole löytänyt vielä lopullista muotoaan.

VIITELUETTELO

Alexander, C. et al. 1977. *A Pattern Language - Towns - Buildings - Contruction*. New York: Oxford University Press. 1171 s. ISBN 0-19-501919-9

Eerola, Anne. 2004. *Käyttöliittymän toteutus - Arkkitehtuurit*. [verkkodokumentti]. [viitattu 18.8.2006]. Saatavilla http://www.cs.uku.fi/kurssit/KAT/extdoc/KAT-Arkkitehtuuri-AE.pdf

eNode, Inc. 2002. *Model-View-Controller Pattern*. [verkkodokumentti]. [viitattu 4.8.2006]. Saatavilla http://www.enode.com/x/markup/tutorial/mvc.html

Ericsson. 2005. *Ericssonin Mobile Softswitch Solution*. [verkkodokumentti]. [viitattu 1.8.2006]. Saatavilla http://swbusiness.fi/portal/news/news_from_software_business_cluster/product_news/?id=4581

Gamma, E. et al. 2001. *Design Patterns: Olio-ohjelmoinnin suunnittelumallit.* Helsinki: IT Press. 395 s. ISBN 951-826-428-7

Haikala, Ilkka & Märijärvi, Jukka. 2000. *Ohjelmistotuotanto*. Pieksämäki: Rt-print Oy. 414 s. ISBN 951-762-769-6

Immonen, Mirja. 2002. *Suunnittelumallit.*[verkkodokumentti] Erikoistyö. Kuopion yliopisto. Tietojenkäsittelytieteen ja sov.matem. laitos. [viitattu 13.8.2006] Saatavilla http://opetus.stadia.fi/savolainen/OhjelmoinninJatkokurssi/Immonen-2002.pdf

Järvi, Antero & Alhoniemi, Esa. 2006. *Ohjelmiston ositus ja arkkitehtuurityylit*. [verkkodokumentti].

[viitattu 18.8.2006]. Saatavilla

http://staff.cs.utu.fi/kurssit/ohjelmistotuotanto/06/OsitusJaArkkitehtuurityylit.pdf

Karvonen, Tuomas. 2005. *Ericssonilta eväitä ip-pohjaiseen viestintään*. ITviikko [verkkolehti]. 24.2.2005 [viitattu 1.8.2006]. Saatavilla http://www.itviikko.fi/page.php?page_id=15&news_id=2005993

Koskimies, Kai. 2000. *Oliokirja*. Jyväskylä: Gummerus. 422 s. ISBN 951-762-720-3.

Koskimies, Kai & Mikkonen, Tommi. 2005. *Ohjelmistoarkkitehtuurit.* Helsinki: Talentum. 245 s. ISBN 952-14-0862-6

Kähkipuro, Pekka. 2005. *Arkkitehtuurit - pelastusko tietotekniikka haasteisiin?.* Systeemityö 05:3. s. 5-7 [viitattu 1.8.2006] Saatavilla: http://www.pcuf.fi/sytyke/lehti/kirj/st20053/ST053-05A.pdf

Lahtela, Marko & Perosuo Tomas & Syväluoma Juuso. *Elokuvatietokannan toteuttaminen J2EE-tekniikalla.* [verkkodokumentti]. Seminaarityö. Helia. [viitattu 12.10.2006] Saatavilla http://koti.mbnet.fi/lahtelam/liitteet/j2ee_elokuvatietokanta.pdf

Laine, Harri. 2000. *Ohjelmistoarkkitehtuuri, Model-View-Controller.* [verkkodokumentti]. Helsingin yliopisto [viitattu 12.7.2006]. Saatavilla http://www.cs.helsinki.fi/u/laine/arkki/k00/jarkki5.pdf

Laine, Harri. 2000a. *Ohjelmistoarkkitehtuuri, kerrosarkkitehtuuri.* [verkkodokumentti]. Helsingin yliopisto [viitattu 12.7.2006]. Saatavilla http://www.cs.helsinki.fi/u/laine/arkki/k01/ark01_2.pdf

Pohjonen, Risto. 2002. *Tietojärjestelmien kehittäminen*. Jyväskylä: Docendo. 178 s. ISBN 951-846-146-5

Poutanen, Jouko. 2000. *Arkkitehtuurikeskeisyys sovelluskehitysprosessissa.* Systeemityö [verkkolehti], 00:4. s.18-19. [viitattu 12.7.2006]. Saatavilla http://www.pcuf.fi/sytyke/lehti/kirj/st20004/18-19.pdf

Reenskaug Trygve. 1979. *MVC XEROX PARC 1978 - 1979.* [verkkodokumentti]. Oslon yliopisto [viitattu 28.6.2006]. Saatavilla http://folk.uio.no/trygver/themes/mvc/mvc-index.html

Reenskaug Trygve. 1979a. *Thing - Model - View - Editor*. [verkkodokumentti] [viitattu 28.6.2006]. Saatavilla http://folk.uio.no/trygver/1979/mvc-1/1979-05-MVC.pdf

Rupp, N.A. 2003. *Beyond MVC: A New Look at the Servlet Infrastructure.* [verkkodokumentti]. [viitattu 16.9.2006]. Saatavissa http://today.java.net/pub/a/today/2003/12/11/mvc.html

Sun Microsystems, Inc. 2002a. *Model - View - Controller*. [verkkodokumentti]. [viitattu 6.8.2006]. http://java.sun.com/blueprints/patterns/MVC-detailed.html

Sun Microsystems, Inc. 2002b. *Designing enterprise applications.* [verkkodokumentti]. [viitattu 13.8.2006]. Saatavilla http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/DEA2eIX.html

Viljamaa, Jukka. 2005. *Ohjelmistotekniikan menetelmät.* [verkkodokumentti]. [viitattu 12.7.2006]. Saatavilla http://www.cs.helsinki.fi/u/jviljama/otm/s05/OTM-Osa7-Arkkitehtuurisuunnittelu.pdf

Vuorinen, Simo. 2002. *J2EE-arkkitehdin työkalupakki*.[verkkolehti] Systeemityö 02:3. s.2 - 5. [viitattu 17.72006]. Saatavilla www.pcuf.fi/sytyke/lehti/kirj/st20023/st023.pdf

Vuorinen, Simo. 2005. *Terveisiä arkkitehtuuriviidakosta!.* Systeemityö 05:3. s. 4. [viitattu 1.8.2006]. ISSN1237-0523

Vuorinen, Simo. 2005a. *Ohjelmistoarkkitehtuurista puoliketterästi.* Systeemityö 05:3. s. 24-28. [viitattu 1.8.2006]. ISSN1237-0523

Wikipedia. 2006a. *MVC-arkkitehtuuri*. [verkkodokumentti][viitattu 11.7.2006]. Saatavilla http://fi.wikipedia.org/wiki/MVC-arkkitehtuuri

Wikipedia. 2006b. *Tietojärjestelmäarkkitehtuuri*. [verkkodokumentti]. [viitattu 17.8.2006]. Saatavilla http://fi.wikipedia.org/wiki/Tietoj%C3%A4rjestelm%C3%A4arkkitehtuuri

Wikipedia 2006c. *API.* [verkkodokumentti][viitattu 5.10.2006]. Saatavilla http://fi.wikipedia.org/wiki/API

Wikipedia 2006d. *ebXML.* [verkkodokumentti][viitattu 5.10.2006]. Saatavilla http://fi.wikipedia.org/wiki/EbXML

Wikipedia 2006e. *GUI.*[verkkodokumentti][viitattu 5.10.2006]. Saatavilla http://fi.wikipedia.org/wiki/GUI

Wikipedia 2006f. *HTML.* [verkkodokumentti][viitattu 5.10.2006]. Saatavilla http://fi.wikipedia.org/wiki/HTML

Wikipedia 2006g. *HTTP.* [verkkodokumentti][viitattu 5.10.2006]. Saatavilla http://fi.wikipedia.org/wiki/HTTP

Wikman, Kimm. 2001. *Olioarkkitehtuurit ja suunnitelumallit*. [verkkodokumentti]. Tutkielma. Helsingin Yliopisto. [viitattu 1.11.2006]. Saatavilla www.cs.helsinki.fi/u/ksavikma/seminaarit/tiki/tutkielma.doc

LIITTEET

Liite 1. Painoindeksilaskimen ohjelmakoodi

Liite 2. Painoindeksisovelluksen ohjelmakoodi

Liite 3. Painoindeksisovelluksen UML-kaavio

Liite 4. Painoindeksisovelluksen Java-dokumentaatio

PAINOINDEKSILASKIMEN OHJELMAKOODI

//SOVELLUS

```
package bmiproject;
import java.awt.*;
import javax.swing.*;

public class BMIapplication
{
    boolean packFrame = false;
    /**
     * Construct and show the application.
     */
    public BMIapplication()
    {
        BMImain frame = new BMImain();
        BMImodel model = new BMImodel();
        BMIcontroller controller = new BMIcontroller(frame, model);
        frame.setController(controller); // link is constructed with
view

        // Validate frames that have preset sizes
        // Pack frames that have useful preferred size info, e.g.
from their layout
        if (packFrame)
        {
            frame.pack();
        }
        else
        {
            frame.validate();
        }

        // Center the window
        Dimension screenSize =
Toolkit.getDefaultToolkit().getScreenSize();
        Dimension frameSize = frame.getSize();
        if (frameSize.height > screenSize.height)
        {
            frameSize.height = screenSize.height;
        }
        if (frameSize.width > screenSize.width)
        {
            frameSize.width = screenSize.width;
        }
        frame.setLocation((screenSize.width - frameSize.width) / 2,
                          (screenSize.height - frameSize.height) /
2);
        frame.setVisible(true);
    }

    /* Application entry point.     */
    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(new Runnable()
        {
            public void run()
            {
                try
                {
```

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
                }
                catch (Exception exception)
                {
                    exception.printStackTrace();
                }

                new BMIapplication();
            }
        });
    }
}
```

//VIEW

```
package bmiproject;
import java.awt.*;
import javax.swing.*;

public class BMImain extends JFrame
{
    JPanel contentPane;
    BorderLayout borderLayout1 = new BorderLayout();

    JPanel inputPane = new JPanel(); // Panel for labels and input
fields

    JLabel height = new JLabel("Enter your height in meters");
    JTextField inputHeight = new JTextField(10);

    JLabel weight = new JLabel("Enter your weight in kilos");
    JTextField inputWeight = new JTextField(10);
    JButton button = new JButton("Submit");

    JLabel BMI = new JLabel("Your BMI is: ");
    JTextField BMIresult = new JTextField(10);

    BMIcontroller controller;

    public BMImain()
    {
        try
        {
            setDefaultCloseOperation(EXIT_ON_CLOSE);
            jbInit();
        }
        catch (Exception exception)
        {
            exception.printStackTrace();
        }
    }

    private void jbInit() throws Exception
    {
        contentPane = (JPanel) getContentPane();
        contentPane.setLayout(borderLayout1);

        inputPane.setLayout(new GridLayout(8,2));

        inputPane.add(height);
        inputPane.add(inputHeight);
```

```
        contentPane.add(inputPane, BorderLayout.CENTER);

        inputPane.add(weight);
        inputPane.add(inputWeight);
        contentPane.add(inputPane, BorderLayout.CENTER);

        inputPane.setBackground(Color.pink);
        contentPane.add(button, BorderLayout.SOUTH);

        inputPane.add(BMI);
        inputPane.add(BMIresult);

        setSize(new Dimension(400, 300));
        setTitle("BMI Calculator");
    }

    public void setController(BMIcontroller c) //täytyy olla!!
    {
        controller = c; // link is constructed
        button.addActionListener(controller);
    }

    public String getHeigth() //HUOM BMI calculator!! From here goes
to controller
    {
        return inputHeight.getText();
    }

    public String getWeight()
    {
     return inputWeight.getText();
    }

    public void showResult(String Str)// (double Str)
    {
        BMIresult.setText(Str); // .setText("" + Str)
    }
}
//CONTROLLER
package bmiproject;
import java.awt.event.*;
import java.awt.event.ActionListener;

public class BMIcontroller implements ActionListener
{
        BMImain view;
        BMImodel model;

        public BMIcontroller(BMImain v, BMImodel m) // link to
control.. and model
        {
            view = v;
            model = m;
        }

        public void actionPerformed(ActionEvent e)
        {
            if (e.getActionCommand().equals("Submit")) // data is
ready in the view
            {
                double height = Double.parseDouble(view.getHeigth());
```

```
                double weight = Double.parseDouble(view.getWeight());
                double bmi = model.processData(weight, height);
                Double d1 = new Double(bmi);
                String BMIresult = new String(d1.toString());
                view.showResult(BMIresult);
            }
        }
    }
```

//MODEL

```
package bmiproject;
public class BMImodel
{
    public BMImodel()
    {    }

    public static double processData(double weight, double height)
    {
        double BMIresult = weight/(height*height);
        System.out.println(BMIresult);
        return BMIresult;
    }
}
```

Painoindeksisovellus

```
//BMIsovellus
package painoindeksi;

import java.awt.*;
import javax.swing.*;
import java.util.Observable;

/**
 * Tarkkailija -luokka.
 */
class Tarkkailtava extends Observable {
  public void notifyObservers(Object b) {
    // Muuten se ei levitä muutosta
    setChanged();
    super.notifyObservers(b);
  }
}

/**
 * Sovelluksen ylätaso.
 */
public class BMIsovellus
{
    Observable tarkkailtava;
    BMImalli malli;
    BMInakyma nakyma;
    BMIohjain ohjain;

    /**
     * Muodostetaan ja luodaan malli, näkymä ja ohjain. Luodaan myös
tarkkailija-
     * olio mallin ja näkymän välille.
     */
    public BMIsovellus()
    {
        tarkkailtava = new Tarkkailtava();
        malli = new BMImalli(tarkkailtava);
        nakyma = new BMInakyma(tarkkailtava);
        ohjain = new BMIohjain(nakyma, malli);
        nakyma.asetaOhjain(ohjain);
        nakyma.setVisible(true);
        nakyma.validate();

        // Asettaa avautuvan ikkunan keskelle näyttöä
        Dimension naytonKoko = Toolkit.getDefaultToolkit().getScreenSize();
        Dimension ikkunanKoko = nakyma.getSize();
        if (ikkunanKoko.height > naytonKoko.height)
        {
            ikkunanKoko.height = naytonKoko.height;
        }
        if (ikkunanKoko.width > naytonKoko.width)
        {
            ikkunanKoko.width = naytonKoko.width;
        }
        nakyma.setLocation((naytonKoko.width - ikkunanKoko.width) / 2,
                           (naytonKoko.height - ikkunanKoko.height) / 2);
        nakyma.setVisible(true);
    }

    /**
     * Sovelluksen suorittaminen alkaa tästä.
     *
     * @param args String[]
     */
    public static void main(String[] args)
```

```
    {
        SwingUtilities.invokeLater(new Runnable()
        {
            public void run()
            {
                try
                {

UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
                }
                catch (Exception exception)
                {
                    exception.printStackTrace();
                }
                // Kutsutaan sovelluksen konstruktoria
                new BMIsovellus();
            }
        });
    }
}
```

Painoindeksisovelluksen - malli

```
package painoindeksi;

import java.util.Observable;

public class BMImalli {

  BMI kaikki;
  BMI miehet;
  BMI naiset;

  public BMImalli(Observable t) {
    kaikki = new BMIjoukko("Kaikki", t);
    miehet = new BMIjoukko("Miehet", t);
    naiset = new BMIjoukko("Naiset", t);
    kaikki.lisaa(miehet);
    kaikki.lisaa(naiset);
  }

  /**
     * Lisää henkilön malliin.
     * @param nimi henkilön nimi
     * @param pituus henkilön pituus
     * @param paino henkilön paino
     * @param onMies true, jos henkilö on mies
     * @return true, jos onnistuu
     * @return false, jos lisäys ei onnistu
   */
  boolean lisaa(String nimi, double pituus, double paino, boolean onMies) {
    if(onMies) {
      return miehet.lisaa(new BMIyksilo(nimi, pituus, paino));
    }
    else {
      return naiset.lisaa(new BMIyksilo(nimi, pituus, paino));
    }
  }

  /**
   * Poistaa järjestynumeron mukaisen henkilöm.
   * @param indeksi poistettavan henkilön järjestysnumero.
   * @return true, jos onnistuu
   * @return false, jos ei onnistu poistamisessa.
   */
  boolean poista(int indeksi) {
```

```
    if(indeksi>=kaikki.BMIlkm()) {
      return false;
    }

    if (kaikki.haeBMIyksilo(indeksi).haeVanhempi().haeNimi() == "Miehet") {
      return miehet.poista(indeksi);
    }
    else {
      return naiset.poista(indeksi - miehet.BMIlkm());
    }
  }

  /**
   * Palauttaa mallin yksilöiden (henkilöt) lukumäärän.
   * @return yksilöiden lukumäärä
   */
  public int haeLkm() {
    return kaikki.BMIlkm();
  }

  /**
   * Palauttaa mallin indeksiä vastaavan yksilön nimen.
   * @param indeksi yksilön järjestysnumero
   * @return nimi
   */
  public String haeNimi(int indeksi) {
      return kaikki.haeBMIyksilo(indeksi).haeNimi();
  }

  /**
   * Palauttaa mallin indeksiä vastaavan yksilön painon
   * @param indeksi yksilön järjestysnumero
   * @return paino
   */
  public double haePaino(int indeksi) {
      return kaikki.haeBMIyksilo(indeksi).haePaino();
  }

  /**
   * Palauttaa mallin indeksiä vastaavan yksilön pituuden.
   * @param indeksi yksilön järjestysnumero
   * @return pituus
   */
  public double haePituus(int indeksi) {
      return kaikki.haeBMIyksilo(indeksi).haePituus();
  }

  /**
   * Palauttaa mallin indeksiä vastaavan yksilön painoindeksin.
   * @param indeksi yksilön järjestysnumero
   * @return painoindeksi
   */
  public double haeBMI(int indeksi) {
      return kaikki.haeBMIyksilo(indeksi).haeBMI();
  }

  /**
   * Palauttaa mallin indeksiä vastaavan yksilön sukupuolen.
   * @param indeksi yksilön järjestysnumero
   * @return "mies" tai "nainen"
   */
  public String haeSukupuoli(int indeksi) {
      if(kaikki.haeBMIyksilo(indeksi).haeVanhempi().haeNimi()=="Miehet")
        return "mies";
      else
        return "nainen";
  }
}
```

BMI-luokka

```
package painoindeksi;

import java.util.LinkedList;
import java.util.ListIterator;
import java.util.Observable;

/**
 * Rekursiokooste-suunnittelumallin abstrakti koosteluokka.
 */

public abstract class BMI extends Observable{

  LinkedList BMIlista;
  BMI vanhempi;
  String nimi;
  Observable tarkkailtava;
  /**
   * Palauttaa yksilöiden lukumäärän.
   * @return yksilöiden lukumäärä
   */
  public abstract int BMIlkm();

  /**
   * Palauttaa painoindeksin.
   * @return painoindeksi
   */
  public abstract double haeBMI();

  /**
   * Palauttaa pituuden.
   * @return pituus
   */
  public abstract double haePituus();

  /**
   * Palauttaa painon.
   * @return paino
   */
  public abstract double haePaino();

  /**
   * Palauttaa yksilön järjestysnumeron perusteella.
   * @param numero yksilön järjestysnumero
   * @return yksilö
   */
  public abstract BMI haeBMIyksilo(int numero);

  /**
   * Lisää joukkoon alijoukon (solmu) tai yksilön (lehti).
   * @param lisattavaBMI lisättävä joukko tai yksilö
   * @return true, jos onnistuu
   * @return false, jos lisäys ei onnistu
   */
  public abstract boolean lisaa(BMI lisattavaBMI);

  /**
   * Poistaa järjestynumeron mukaisen yksilön/joukon listasta.
   * @param numero poistettavan yksilön/joukon järjestysnumero.
   * @return true, jos onnistuu
   * @return false, jos ei onnistu poistamisessa.
   */
  public abstract boolean poista(int numero);

  /**
   * Luodaan listan iteroija.
   * @return listaiteraattori
   */
```

```
  public abstract ListIterator luoListaiteraattori();

  /**
   * Asettaa yksilölle/joukolle linkin yläjoukkoon.
   * @param asetettavaVanhempi yläjoukko
   */

  public void asetaVanhempi(BMI asetettavaVanhempi) {
    vanhempi = asetettavaVanhempi;
  }

  /**
   * Palauttaa yksilön/joukon yläjoukon.
   * @return yläjoukko
   */
  public BMI haeVanhempi() {
    return vanhempi;
  }

  /**
   * Asettaa yksilön/joukon nimen.
   * @param asetettavaNimi nimi
   */
  public void asetaNimi(String asetettavaNimi) {
    nimi = asetettavaNimi;
  }

  /**
    * Palauttaa yksilön/joukon nimen.
    * @return nimi
   */
  public String haeNimi() {
    return nimi;
  }
}
```

BMIjoukko-luokka

```
package painoindeksi;

import java.util.ListIterator;
/**
 * Rekursiokooste-suunnittelumallin lehtiluokka.
 * Sovelluksen yksittäiset henkilöt ovat tämän luokan ilmentymiä.
 */
public class BMIyksilo extends BMI { // Abstraktin BMI-luokan aliluokka

  double pituus;
  double paino;
  /**
    * Luo uuden yksilön.
    * @param nimi henkilön nimi
    * @param pituus henkilön pituus
    * @param paino henkilön paino
   */
  public BMIyksilo(String nimi, double pituus, double paino) {
    this.asetaNimi(nimi);
    this.pituus=pituus;
    this.paino=paino;
  }

  /**
   * Palauttaa henkilöiden lukumäärän.
   * @return 1
   */
  public int BMIlkm() {
    return 1;
  }
```

```
/**
 * Palauttaa henkilön pituuden.
 * @return pituus
 */
public double haePituus() {
  return pituus;
}

/**
 * Palauttaa henkilön painon.
 * @return paino
 */
public double haePaino() {
  return paino;
}

/**
 * Palauttaa henkilön painoindeksin.
 * @return painoindeksi
 */
public double haeBMI() {
  return paino/(pituus*pituus);
}

/**
 * Palauttaa järjestysnumeron mukaisen henkilön.
 * @param numero järjestysnumero
 * @return itsensä, jos numero on 0
 * @return null, jos numero on erisuuri kuin 0
 */
public BMI haeBMIyksilo(int numero) {
  if(numero==0)
    return this;
  else
    return null;
}

/**
 * Dummy-metodi.
 * Toteuttaa abstraktin luokan vaatiman metodin.
 * Yksilö ei voi sisältää aliyksilöitä.
 * @param lisattavaBMI lisättävä yksilö/joukko
 * @return false
 */
public boolean lisaa(BMI lisattavaBMI) {
  return false;
}

/**
 * Dummy-metodi.
 * Toteuttaa abstraktin luokan vaatiman metodin.
 * Yksilö ei voi sisältää aliyksilöitä, joten niiden poistaminenkaan
 * ei ole mahdollista.
 * @param numero poistettavan yksilön/joukon järjestysnumero
 * @return false
 */
public boolean poista(int numero) {
  return false;
}

/**
 * Dummy-metodi.
 * Toteuttaa abstraktin luokan vaatiman metodin.
 * Yksilö ei voi sisältää linkitettyä listaa.
 * @return null
 */
public ListIterator luoListaiteraattori() {
  return null;
```

```
  }

}
```

Painoindeksisovelluksen näkymät

BMInakyma-luokka

```
package painoindeksi;

import javax.swing.*;
import java.awt.*;
import java.util.Observable;

/**
 * MVC-arkkitehtuurin nakyma (view).
 * Luo pääikkunan, asettaa siihen lista- ja tilastonäkymät.
 */
public class BMInakyma extends JFrame{

  JTabbedPane valilehdet;
  BMIlistanakyma listaNakyma;
  BMItilastonakyma tilastoNakyma;
  BMIlisaa lisaaIkkuna;

  /**
   * Luo lista- ja tilastonäkymän.
   * Tekee niistä tarkkailijoita.
   * Kutsuu jbInit funktiota.
   * @param t tarkkailija
   */

  public BMInakyma(Observable t) {

    valilehdet = new JTabbedPane();
    listaNakyma = new BMIlistanakyma(t);
    tilastoNakyma = new BMItilastonakyma(t);
    lisaaIkkuna = new BMIlisaa();

    try {
      jbInit();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }
  /**
   * Lisää lista- ja tilastonäkymän välilehdille, asettaa ikkunan koon ja
   * perusominaisuuksia. JBuilder Designerin luoma funktio, joten sen
   * nimeä ei kannata muuttaa.
   * @throws Exception
   */
  private void jbInit() throws Exception {
    this.getContentPane().add(valilehdet);
    valilehdet.add(listaNakyma, "Lista");
    valilehdet.add(tilastoNakyma, "Tilasto");
    this.setSize(400,500);
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
  /**
   * Palauttaa henkilöiden lisäysikkunan.
   * @return lisaysikkuna
   */
  public BMIlisaa haeLisaaIkkuna() {
      return lisaaIkkuna;
  }
  /**
   * Palauttaa listanäkymässä valitun rivin.
   * @return valittu rivi
```

```
   */
  public int haeValittuRivi() {
    return listaNakyma.haeValittuRivi();
  }
  /**
   * Asettaa ohjaimen kuuntelemaan listanäkymän ja lisäysikkunan
painikkeita.
   * @param o asetettava ohjain
   */
  public void asetaOhjain(BMIohjain o) {
    // link is constructed
    listaNakyma.asetaOhjain(o);
    lisaaIkkuna.asetaOhjain(o);
  }
}
```

BMItilastonakyma-luokka

```
package painoindeksi;

import java.awt.*;
import javax.swing.*;
import java.util.*;
import java.text.DecimalFormat;

/**
 * Luo tilastonäkymän, joka sisältää pylväsdiagrammin joukkojen
painoindeksien keskiarvoista.
 * Tarkkailee mallin muutoksia.
 */
public class BMItilastonakyma extends JPanel implements Observer {
  // Tilastonäkymä
  JPanel pylvaat = new JPanel();
  JPanel nimet = new JPanel();
  JLabel Lmiehet = new JLabel("Miehet 0,0");
  JLabel Lnaiset = new JLabel("Naiset 0,0");
  JLabel Lkaikki = new JLabel("Kaikki 0,0");
  Pylvas Pmiehet = new Pylvas();
  Pylvas Pkaikki = new Pylvas();
  Pylvas Pnaiset = new Pylvas();
  JPanel ylatila = new JPanel();

  /**
   * Asetetaan tilastonäkymän komponentit paikalleen.
   * @param tarkkailtava malli, jota tarkkaillaan
   */
  public BMItilastonakyma(Observable tarkkailtava) {
    tarkkailtava.addObserver(this);
    this.setLayout(new BorderLayout());
    nimet.setMinimumSize(new Dimension(30, 30));
    nimet.setPreferredSize(new Dimension(30, 30));
    nimet.setLayout(new GridLayout(1,3));
    pylvaat.setLayout(new GridLayout(1,3));
    Lmiehet.setHorizontalAlignment(SwingConstants.CENTER);
    Lnaiset.setHorizontalAlignment(SwingConstants.CENTER);
    Lkaikki.setHorizontalAlignment(SwingConstants.CENTER);
    ylatila.setMinimumSize(new Dimension(30, 30));
    ylatila.setPreferredSize(new Dimension(30, 30));
    ylatila.add(new Label("BMI keskiarvot", SwingConstants.CENTER));
    this.add(pylvaat, java.awt.BorderLayout.CENTER);
    pylvaat.add(Pmiehet);
    pylvaat.add(Pnaiset);
    pylvaat.add(Pkaikki);
    this.add(nimet, java.awt.BorderLayout.SOUTH);
    nimet.add(Lmiehet);
    nimet.add(Lnaiset);
    nimet.add(Lkaikki);
    this.add(ylatila, java.awt.BorderLayout.NORTH);
```

```
  }
  /**
   * Päivittää diagrammin pylväiden pituudet ja tekstit, kun malli on
muuttunut.
   * @param o tarkkailtava
   * @param arg objekti, joka lisättiin/poistettiin
   */

  public void update(Observable o, Object arg) { //Päivitysmetodi

    BMI apu = (BMI)arg;
    DecimalFormat df = new DecimalFormat("0.0");
    int korkeusK, korkeusM, korkeusN;

    if(apu.haeVanhempi().haeNimi() == "Miehet") {
      Lmiehet.setText(apu.haeVanhempi().haeNimi() + " " +
df.format(apu.haeVanhempi().haeBMI()));
      Pmiehet.asetaKorkeus( (int) apu.haeVanhempi().haeBMI());
      Lkaikki.setText(apu.haeVanhempi().haeVanhempi().haeNimi() + " " +
df.format(apu.haeVanhempi().haeVanhempi().haeBMI()));
      Pkaikki.asetaKorkeus( (int) apu.haeVanhempi().haeVanhempi().haeBMI());
    }
    else if(apu.haeVanhempi().haeNimi() == "Naiset") {
      Lnaiset.setText(apu.haeVanhempi().haeNimi() + " " +
df.format(apu.haeVanhempi().haeBMI()));
      Pnaiset.asetaKorkeus( (int) apu.haeVanhempi().haeBMI());
      Lkaikki.setText(apu.haeVanhempi().haeVanhempi().haeNimi() + " " +
df.format(apu.haeVanhempi().haeVanhempi().haeBMI()));
      Pkaikki.asetaKorkeus( (int) apu.haeVanhempi().haeVanhempi().haeBMI());
    }

    // Asetetaan pylväiden korkeudet
    repaint(); //Päivitetään piirros
  }
}

class Pylvas
    extends JPanel {
  int korkeus;
  static int korkein;

  public Pylvas() {
    this.korkeus = 0;
    korkein = 0;
  }

  public void asetaKorkeus(int korkeus) {
    this.korkeus = korkeus;
    if (korkeus > korkein)
      korkein = korkeus;
  }

  public void paintComponent(Graphics g) { //Piirretään kuvio
    int apu;

    super.paintComponent(g);
    Dimension s = getSize();

    if(korkein > 0)
      apu = s.height * this.korkeus / korkein;
    else
      apu = 0;

    g.setColor(Color.blue);
    g.fillRect(s.width / 4, s.height - apu, s.width / 2, apu);
  }
}
```

BMIlistanakyma-luokka

```
package painoindeksi;

import javax.swing.*;
import java.awt.*;
import java.util.*;
import javax.swing.table.AbstractTableModel;
import java.text.DecimalFormat;

/**
 * Luo listanäkymän, joka sisältää listan henkilöistä, sekä lisää- ja
poista-painikkeet.
 * Tarkkailee mallin muutoksia.
 */
public class BMIlistanakyma extends JPanel implements Observer {
  //Lista ja buttonit: lisää, poista
  JTable taulukko;
  JButton Blisaa = new JButton("Lisää...");
  JButton Bpoista = new JButton("Poista");
  BMIohjain ohjain;

  //taulukkomalli, joka kytkee mallin yksilöt JTable-komponenttiin
  class taulukkoMalli extends AbstractTableModel {
    private String[] otsikot = {
        "Nimi",
        "Pituus",
        "Paino",
        "BMI",
        "Sukupuoli"};

    public int getColumnCount() {
      return otsikot.length;
    }

    public int getRowCount() {
      try {
        return ohjain.haeLkm();
      }
      catch (NullPointerException e) {
        return 0;
      }
    }

    public String getColumnName(int col) {
      return otsikot[col];
    }

    public Object getValueAt(int row, int col) {

      DecimalFormat df = new DecimalFormat("0.0");

      switch (col) {
        case 0:
          return ohjain.haeNimi(row);
        case 1:
          return ohjain.haePituus(row);
        case 2:
          return ohjain.haePaino(row);
        case 3:
          return df.format(ohjain.haeBMI(row));
        case 4:
          return ohjain.haeSukupuoli(row);
        default:
          return 0;
      }
    }
  }
```

```
  /**
   * Asettelee listan, painikkeet paikoilleen
   * @param tarkkailtava malli, jota tarkkaillaan
   */
  public BMIlistanakyma(Observable tarkkailtava) { //konstruktori
    tarkkailtava.addObserver(this);
    JPanel painikepaneeli = new JPanel();
    painikepaneeli.setLayout(new GridLayout(1, 2, 5, 5));
    painikepaneeli.setMinimumSize(new Dimension(30, 30));
    painikepaneeli.setPreferredSize(new Dimension(30, 30));
    painikepaneeli.add(Blisaa);
    painikepaneeli.add(Bpoista);
    this.setLayout(new BorderLayout());

    taulukko = new JTable(new taulukkoMalli());
    taulukko.setPreferredScrollableViewportSize(new Dimension(500, 70));

    //Create the scroll pane and add the table to it.
    JScrollPane scrollPane = new JScrollPane(taulukko);

    this.add(scrollPane, java.awt.BorderLayout.CENTER);
    this.add(painikepaneeli, java.awt.BorderLayout.SOUTH);
  }
  /**
   * Asettaa painikkeiden tapahtumakuuntelijaksi ohjaimen.
   * @param o ohjain
   */
  public void asetaOhjain(BMIohjain o) {
    ohjain = o; // link is constructed
    Blisaa.addActionListener(ohjain);
    Bpoista.addActionListener(ohjain);
  }
  /**
   * Palauttaa taulukosta valitun rivin numeron
   * @return rivinumero
   */
  public int haeValittuRivi() {
    return taulukko.getSelectedRow();
  }
  /**
   * Päivittää taulukon, kun malli on muuttunut
   * @param o tarkkailtava
   * @param arg objekti, joka lisättiin/poistettiin
   */
  public void update(Observable o, Object arg) { //Päivitysmetodi
    taulukko.updateUI();
  }
}
```

BMIlisaa-luokka

```
package painoindeksi;

import javax.swing.*;
import java.awt.*;

/**
 * Luo lisäysikkunan, jonka avulla saadaan luotua uusia yksilöitä.
 */
public class BMIlisaa extends JFrame{

  JLabel Lnimi = new JLabel();
  JLabel Lsukupuoli = new JLabel();
  JLabel Lpaino = new JLabel();
  JLabel Lpituus = new JLabel();
  JTextField Tnimi = new JTextField();
  JTextField Tpituus = new JTextField();
  JTextField Tpaino = new JTextField();
  ButtonGroup Rryhma = new ButtonGroup();
```

```
JPanel paneeli = new JPanel();
JRadioButton Rmies = new JRadioButton();
JRadioButton Rnainen = new JRadioButton();
JButton Bok = new JButton();
JButton Bperuuta = new JButton();

 /**
  * Käynnistää jbInit-funktion.
  */
 public BMIlisaa() {

  try {
    jbInit();
  }
  catch (Exception ex) {
    ex.printStackTrace();
  }

}

/**
 * Asettaa painikkeiden tapahtumakuuntelijaksi ohjaimen.
 * @param o ohjain
 */
public void asetaOhjain(BMIohjain o) {
  Bok.addActionListener(o);
  Bperuuta.addActionListener(o);
}
/**
 * Palauttaa tekstikenttään syötetyn nimen.
 * @return nimi
 */
public String haeNimi() {
  return Tnimi.getText();
}

/**
 * Palauttaa tekstikenttään syötetyn pituuden.
 * @return pituus
 */
public double haePituus() {
  return Double.parseDouble(Tpituus.getText());
}


/**
 * Palauttaa tekstikenttään syötetyn painon.
 * @return paino
 */
public double haePaino() {
  return Double.parseDouble(Tpaino.getText());
}
/**
 *
 * @return true, jos kyseessä on mies
 * @return false, jos kyseessä on nainen
 */
public boolean onMies() {
  if(Rmies.isSelected())
    return true;
  else
    return false;
}
/**
 * Asettaa lisäysikkunan perusominaisuudet.
 * @throws Exception
 */
```

```
  private void jbInit() throws Exception {
    this.getContentPane().setLayout(new GridBagLayout());
    Lnimi.setText("Nimi:");
    Lsukupuoli.setText("Sukupuoli:");
    Lpaino.setText("Paino:");
    Lpituus.setText("Pituus:");
    this.setResizable(false);
    Tnimi.setSelectedTextColor(Color.white);
    Tnimi.setText("Ville");
    Tpituus.setText("1.8");
    Tpaino.setText("72");
    Rmies.setSelected(true);
    Rmies.setText("mies");
    Bok.setMaximumSize(new Dimension(71, 23));
    Bok.setMinimumSize(new Dimension(71, 23));
    Bok.setPreferredSize(new Dimension(71, 23));
    Bok.setActionMap(null);
    Bok.setText("OK");
    Bperuuta.setText("Peruuta");
    this.getContentPane().add(paneeli,
                              new GridBagConstraints(1, 3, 1, 1, 0.0, 0.0
        , GridBagConstraints.CENTER, GridBagConstraints.NONE,
        new Insets(0, 0, 0, 0), 0, 0));
    Rnainen.setText("nainen");
    paneeli.add(Rmies);
    paneeli.add(Rnainen);
    this.getContentPane().add(Tnimi,
                              new GridBagConstraints(1, 0, 1, 1, 0.0, 0.0
        , GridBagConstraints.CENTER, GridBagConstraints.HORIZONTAL,
        new Insets(0, 0, 10, 0), 0, 0));
    this.getContentPane().add(Tpituus,
                              new GridBagConstraints(1, 1, 1, 1, 0.0, 0.0
        , GridBagConstraints.CENTER, GridBagConstraints.HORIZONTAL,
        new Insets(0, 0, 10, 0), 0, 0));
    this.getContentPane().add(Tpaino,
                              new GridBagConstraints(1, 2, 1, 1, 0.0, 0.0
        , GridBagConstraints.CENTER, GridBagConstraints.HORIZONTAL,
        new Insets(0, 0, 10, 0), 0, 0));
    this.getContentPane().add(Lpituus,
                              new GridBagConstraints(0, 1, 1, 1, 0.0, 0.0
        , GridBagConstraints.WEST, GridBagConstraints.HORIZONTAL,
        new Insets(0, 0, 0, 0), 0, 0));
    this.getContentPane().add(Lnimi,
                              new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0
        , GridBagConstraints.CENTER, GridBagConstraints.HORIZONTAL,
        new Insets(0, 0, 0, 0), 0, 0));
    this.getContentPane().add(Lpaino,
                              new GridBagConstraints(0, 2, 1, 1, 0.0, 0.0
        , GridBagConstraints.EAST, GridBagConstraints.HORIZONTAL,
        new Insets(0, 0, 0, 0), 0, 0));
    this.getContentPane().add(Lsukupuoli,
                              new GridBagConstraints(0, 3, 1, 1, 0.0, 0.0
        , GridBagConstraints.CENTER, GridBagConstraints.HORIZONTAL,
        new Insets(0, 0, 0, 0), 0, 0));
    Rryhma.add(Rmies);
    Rryhma.add(Rnainen);
    this.getContentPane().add(Bok, new GridBagConstraints(0, 4, 1, 1, 0.0,
0.0
        , GridBagConstraints.SOUTHWEST, GridBagConstraints.NONE,
        new Insets(20, 0, 0, 0), 0, 0));
    this.getContentPane().add(Bperuuta,
                              new GridBagConstraints(1, 4, 1, 1, 0.0, 0.0
        , GridBagConstraints.SOUTHEAST, GridBagConstraints.NONE,
        new Insets(20, 0, 0, 0), 0, 0));

    this.setSize(250,220);
    // Asettaa avautuvan ikkunan keskelle näyttöä
    Dimension naytonKoko = Toolkit.getDefaultToolkit().getScreenSize();
```

```
    Dimension ikkunanKoko = this.getSize();
    if (ikkunanKoko.height > naytonKoko.height) {
      ikkunanKoko.height = naytonKoko.height;
    }
    if (ikkunanKoko.width > naytonKoko.width) {
      ikkunanKoko.width = naytonKoko.width;
    }
    this.setLocation( (naytonKoko.width - ikkunanKoko.width) / 2,
                      (naytonKoko.height - ikkunanKoko.height) / 2);
  }
}
```

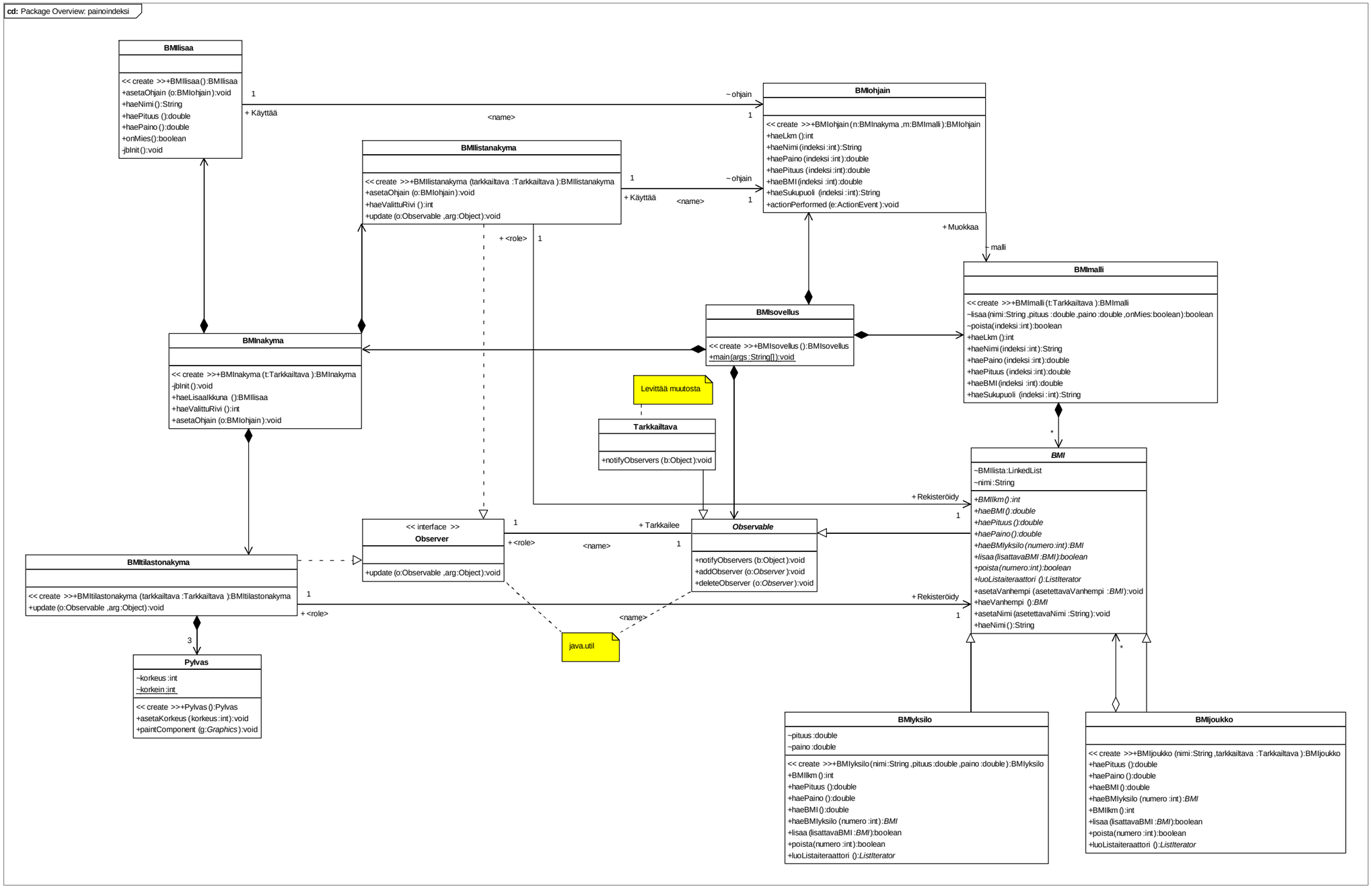
Painoindeksisovelluksen ohjain

BMIohjain

```
package painoindeksi;

import java.awt.event.*;

/**
 * Luodaan MVC-arkkitehtuurin mukainen ohjain (controller).
 */
public class BMIohjain implements ActionListener {

  BMInakyma nakyma;
  BMImalli malli;

  /**
   * Luo mallille alijoukot, miehet ja naiset.
   * @param n nakyma
   * @param m malli
   * @param t tarkkailtava
   */
  public BMIohjain(BMInakyma n, BMImalli m) {
    nakyma = n;
    malli = m;
  }
  /**
   * Palauttaa mallin yksilöiden (henkilöt) lukumäärän.
   * @return yksilöiden lukumäärä
   */
  public int haeLkm() {
    return malli.haeLkm();
  }

  /**
   * Palauttaa mallin indeksiä vastaavan yksilön nimen.
   * @param indeksi yksilön järjestysnumero
   * @return nimi
   */
  public String haeNimi(int indeksi) {
      return malli.haeNimi(indeksi);
  }

  /**
   * Palauttaa mallin indeksiä vastaavan yksilön painon
   * @param indeksi yksilön järjestysnumero
   * @return paino
   */
  public double haePaino(int indeksi) {
      return malli.haePaino(indeksi);
  }

  /**
   * Palauttaa mallin indeksiä vastaavan yksilön pituuden.
   * @param indeksi yksilön järjestysnumero
   * @return pituus
   */
```

```
  public double haePituus(int indeksi) {
      return malli.haePituus(indeksi);
  }

  /**
   * Palauttaa mallin indeksiä vastaavan yksilön painoindeksin.
   * @param indeksi yksilön järjestysnumero
   * @return painoindeksi
   */
  public double haeBMI(int indeksi) {
      return malli.haeBMI(indeksi);
  }
  /**
   * Palauttaa mallin indeksiä vastaavan yksilön sukupuolen.
   * @param indeksi yksilön järjestysnumero
   * @return "mies" tai "nainen"
   */
  public String haeSukupuoli(int indeksi) {
      return malli.haeSukupuoli(indeksi);
  }
  /**
   * Painettaessa listanäkymän lisää-nappulaa, avaa lisäysikkunan.
   * Painettaessa listanäkymän poista-nappulaa, poistaa valitun yksilön.
   * Painettaessa lisäysikkunassa ok-nappulaa, lisää malliin yksilön ja
sulkee lisäysikkunan.
   * Painettaessa lisäysikkunassa peruuta-nappulaa, sulkee lisäysikkunan.
   * @param e ActionEvent
   */
  public void actionPerformed(ActionEvent e) {


    if (e.getActionCommand().equals("Lisää...")) { // data is ready in the
view
      nakyma.haeLisaaIkkuna().setVisible(true);
    }
    else if (e.getActionCommand().equals("Poista")) { // data is ready in
the view
      if (nakyma.haeValittuRivi() >= 0) {
        malli.poista(nakyma.haeValittuRivi());
      }
    }

    else if (e.getActionCommand().equals("OK")) { // data is ready in the
view

      BMIlisaa lisaaIkkuna = nakyma.haeLisaaIkkuna();
      malli.lisaa(lisaaIkkuna.haeNimi(),
                  lisaaIkkuna.haePituus(),
                  lisaaIkkuna.haePaino(),
                  lisaaIkkuna.onMies());

      lisaaIkkuna.setVisible(false);
    }
    else if (e.getActionCommand().equals("Peruuta")) {
      nakyma.haeLisaaIkkuna().setVisible(false);
    }
  }
}
```

LIITE 3. Painoindeksisovelluksen UML-kaavio


cd: Package Overview: painoindeksi
BMIlisaa
<< create >>+BMIlisaa():BMIlisaa
+asetaOhjain (o:BMIohjain):void
+haeNimi():String
+haePituus ():double
+haePaino ():double
+onMies():boolean
-jbInit():void
1
+ Käyttää
<name>
~ ohjain
1
BMIohjain
<< create >>+BMIohjain(n:BMInakyma ,m:BMImalli):BMIohjain
+haeLkm ():int
+haeNimi (indeksi :int):String
+haePaino (indeksi :int):double
+haePituus (indeksi :int):double
+haeBMI (indeksi :int):double
+haeSukupuoli (indeksi :int):String
+actionPerformed (e:ActionEvent):void
+ Muokkaa
~ malli
BMIlistanakyma
<< create >>+BMIlistanakyma (tarkkailtava :Tarkkailtava):BMIlistanakyma
+asetaOhjain (o:BMIohjain):void
+haeValittuRivi ():int
+update (o:Observable ,arg:Object):void
1
+ Käyttää
<name>
~ ohjain
1
+ <role>
1
BMImalli
<< create >>+BMImalli(t:Tarkkailtava):BMImalli
~lisaa(nimi:String ,pituus :double ,paino :double ,onMies:boolean):boolean
~poista(indeksi :int):boolean
+haeLkm ():int
+haeNimi (indeksi :int):String
+haePaino (indeksi :int):double
+haePituus (indeksi :int):double
+haeBMI (indeksi :int):double
+haeSukupuoli (indeksi :int):String
BMIsovellus
<< create >>+BMIsovellus():BMIsovellus
+main(args :String[]):void
BMInakyma
<< create >>+BMInakyma(t:Tarkkailtava):BMInakyma
-jbInit():void
+haeLisaalkkuna ():BMIlisaa
+haeValittuRivi ():int
+asetaOhjain (o:BMIohjain):void
Levittää muutosta
Tarkkailtava
+notifyObservers (b:Object):void
*
BMI
~BMIlista:LinkedList
~nimi:String
+BMIlkm():int
+haeBMI():double
+haePituus ():double
+haePaino ():double
+haeBMIyksilo (numero:int):BMI
+lisaa(lisattavaBMI :BMI):boolean
+poista(numero:int):boolean
+luoListaiteraattori ():ListIterator
+asetaVanhempi (asetettavaVanhempi :BMI):void
+haeVanhempi ():BMI
+asetaNimi (asetettavaNimi :String):void
+haeNimi():String
+ Rekisteröidy
1
+ Rekisteröidy
1
<< interface >>
Observer
+update (o:Observable ,arg:Object):void
1
+ <role>
+ Tarkkailee
<name>
1
Observable
+notifyObservers (b:Object):void
+addObserver (o:Observer):void
+deleteObserver (o:Observer):void
BMItilastonakyma
<< create >>+BMItilastonakyma (tarkkailtava :Tarkkailtava):BMItilastonakyma
+update(o:Observable ,arg:Object):void
1
+ <role>
<name>
3
Pylvas
~korkeus :int
~korkein :int
<< create >>+Pylvas():Pylvas
+asetaKorkeus (korkeus:int):void
+paintComponent (g:Graphics):void
java.util
BMIyksilo
~pituus :double
~paino :double
<< create >>+BMIyksilo(nimi:String ,pituus:double ,paino :double):BMIyksilo
+BMIlkm():int
+haePituus ():double
+haePaino ():double
+haeBMI ():double
+haeBMIyksilo (numero :int):BMI
+lisaa (lisattavaBMI :BMI):boolean
+poista(numero :int):boolean
+luoListaiteraattori ():ListIterator
*
BMIjoukko
<< create >>+BMIjoukko (nimi:String ,tarkkailtava :Tarkkailtava):BMIjoukko
+haePituus ():double
+haePaino ():double
+haeBMI ():double
+haeBMIyksilo (numero :int):BMI
+BMIlkm ():int
+lisaa (lisattavaBMI :BMI):boolean
+poista(numero :int):boolean
+luoListaiteraattori ():ListIterator

painoindeksi

Class BMIsovellus

public class **BMIsovellus**

Sovelluksen ylätaso.

Constructor Detail

BMIsovellus

```
public BMIsovellus()
```

Muodostetaan ja luodaan malli, näkymä ja ohjain. Luodaan myös tarkkailija-olio mallin ja näkymän välille.

Method Detail

main

```
public static void main(java.lang.String[] args)
```

Sovelluksen suorittaminen alkaa tästä.

Parameters:

`args` - String[]

Class BMImalli

public class **BMImalli**

Method Detail

lisaa

```
boolean lisaa(java.lang.String nimi,
              double pituus,
              double paino,
              boolean onMies)
```

Lisää henkilön malliin.

Parameters:

`nimi` - henkilön nimi

`pituus` - henkilön pituus

`paino` - henkilön paino

`onMies` - true, jos henkilö on mies

Returns:

true, jos onnistuu

false, jos lisäys ei onnistu

---

poista

`boolean` **`poista`**`(int indeksi)`

Poistaa järjestynumeron mukaisen henkilöm.

Parameters:

`indeksi` - poistettavan henkilön järjestysnumero.

Returns:

true, jos onnistuu
false, jos ei onnistu poistamisessa.

---

haeLkm

`public int` **`haeLkm`**`()`

Palauttaa mallin yksilöiden (henkilöt) lukumäärän.

Returns:

yksilöiden lukumäärä

---

haeNimi

`public java.lang.String` **`haeNimi`**`(int indeksi)`

Palauttaa mallin indeksiä vastaavan yksilön nimen.

Parameters:

`indeksi` - yksilön järjestysnumero

Returns:

nimi

---

haePaino

`public double` **`haePaino`**`(int indeksi)`

Palauttaa mallin indeksiä vastaavan yksilön painon

Parameters:

`indeksi` - yksilön järjestysnumero

Returns:

paino

---

haePituus

`public double` **`haePituus`**`(int indeksi)`

Palauttaa mallin indeksiä vastaavan yksilön pituuden.

Parameters:

`indeksi` - yksilön järjestysnumero

Returns:

pituus

---

haeBMI

`public double` **`haeBMI`**`(int indeksi)`

Palauttaa mallin indeksiä vastaavan yksilön painoindeksin.

Parameters:

`indeksi` - yksilön järjestysnumero

Returns:

painoindeksi

---

haeSukupuoli

```
public java.lang.String haeSukupuoli(int indeksi)
```

Palauttaa mallin indeksiä vastaavan yksilön sukupuolen.

Parameters:

`indeksi` - yksilön järjestysnumero

Returns:

"mies" tai "nainen"

Class BMI

public abstract class **BMI**
extends java.util.Observable

Rekursiokooste-suunnittelumallin abstrakti koosteluokka.

Method Detail

BMIlkm

```
public abstract int BMIlkm()
```

Palauttaa yksilöiden lukumäärän.

Returns:

yksilöiden lukumäärä

---

haeBMI

```
public abstract double haeBMI()
```

Palauttaa painoindeksin.

Returns:

painoindeksi

---

haePituus

```
public abstract double haePituus()
```

Palauttaa pituuden.

Returns:

pituus

---

haePaino

```
public abstract double haePaino()
```

Palauttaa painon.

Returns:
paino

---

haeBMIyksilo

`public abstract painoindeksi.BMI` **`haeBMIyksilo`**`(int numero)`
Palauttaa yksilön järjestysnumeron perusteella.
Parameters:
`numero` - yksilön järjestysnumero
Returns:
yksilö

---

lisaa

`public abstract boolean` **`lisaa`**`(painoindeksi.BMI lisattavaBMI)`
Lisää joukkoon alijoukon (solmu) tai yksilön (lehti).
Parameters:
`lisattavaBMI` - lisättävä joukko tai yksilö
Returns:
true, jos onnistuu
false, jos lisäys ei onnistu

---

poista

`public abstract boolean` **`poista`**`(int numero)`
Poistaa järjestynumeron mukaisen yksilön/joukon listasta.
Parameters:
`numero` - poistettavan yksilön/joukon järjestysnumero.
Returns:
true, jos onnistuu
false, jos ei onnistu poistamisessa.

---

luoListaiteraattori

`public abstract java.util.ListIterator` **`luoListaiteraattori`**`()`
Luodaan listan iteroija.
Returns:
listaiteraattori

---

asetaVanhempi

`public void` **`asetaVanhempi`**`(painoindeksi.BMI asetettavaVanhempi)`
Asettaa yksilölle/joukolle linkin yläjoukkoon.
Parameters:
`asetettavaVanhempi` - yläjoukko

---

haeVanhempi

`public painoindeksi.BMI` **`haeVanhempi`**`()`

Palauttaa yksilön/joukon yläjoukon.

Returns:

yläjoukko

---

asetaNimi

`public void` **`asetaNimi`**`(java.lang.String asetettavaNimi)`

Asettaa yksilön/joukon nimen.

Parameters:

`asetettavaNimi` - nimi

---

haeNimi

`public java.lang.String` **`haeNimi`**`()`

Palauttaa yksilön/joukon nimen.

Returns:

nimi

Class BMIjoukko

public class **BMIjoukko**
extends painoindeksi.BMI

Rekursiokooste-suunnittelumallin koosteluokka. Rakenteen solmukohta. Sovelluksen joukot: Kaikki, Miehet ja Naiset ovat tämän luokan ilmentymiä.

Constructor Detail

BMIjoukko

```
public BMIjoukko(java.lang.String nimi,
                 java.util.Observable tarkkailtava)
```

Luo uuden joukon ja linkitetyn listan.

Parameters:

`nimi` - joukon nimi
`tarkkailtava` - linkki tarkkailijaan

Method Detail

haePituus

`public double` **`haePituus`**`()`

Palauttaa joukossa olevien yksilöiden pituuksien keskiarvon. Käy läpi rekursiivisesti myös alijoukot.

Returns:

pituuksien keskiarvo.

---

haePaino

`public double` **`haePaino`**`()`

Palauttaa joukossa olevien yksilöiden painojen keskiarvon. Käy läpi rekursiivisesti myös alijoukot.

Returns:

painojen keskiarvo.

---

haeBMI

`public double` **`haeBMI`**`()`

Palauttaa joukossa olevien yksilöiden painoindeksien keskiarvon. Käy läpi rekursiivisesti myös alijoukot.

Returns:

painoindeksien keskiarvo.

---

haeBMIyksilo

`public painoindeksi.BMI` **`haeBMIyksilo`**`(int numero)`

Käy joukon ja sen alijoukot läpi rekursiivisesti ja palauttaa järjestysnumeron mukaisen yksilön. Luotu BMItilastonakyma-luokan JTable-komponenttia varten.

Parameters:

`numero` - yksilön järjestysnumero

Returns:

yksilö, jos onnistuu
null, jos numeroa vastaavaa yksilöä ei löydy.

BMIlkm

`public int` **`BMIlkm`**`()`

Palauttaa joukon ja sen alijoukkojen yksilöiden lukumäärän.

Returns:

yksilöiden lukumäärä

---

lisaa

`public boolean` **`lisaa`**`(painoindeksi.BMI lisattavaBMI)`

Lisää joukkoon alijoukon (solmu) tai yksilön (lehti). Ilmoittaa tarkkailijalle lisäyksestä.

Parameters:

`lisattavaBMI` - lisättävä joukko tai yksilö

Returns:

true, jos onnistuu
false, jos lisäys ei onnistu

---

poista

```
public boolean poista(int numero)
```

Poistaa järjestynumeron mukaisen yksilön/joukon listasta. Ilmoittaa tarkkailijalle poistosta.

Parameters:

`numero` - poistettavan yksilön/joukon järjestysnumero.

Returns:

true, jos onnistuu
false, jos ei onnistu poistamisessa.

---

luoListaiteraattori

```
public java.util.ListIterator luoListaiteraattori()
```

Luodaan listan iteroija.

Returns:

listaiteraattori

Class BMIyksilo

public class **BMIyksilo**
extends painoindeksi.BMI

Rekursiokooste-suunnittelumallin lehtiluokka. Sovelluksen yksittäiset henkilöt ovat tämän luokan ilmentymiä.

Constructor Detail

BMIyksilo

```
public BMIyksilo(java.lang.String nimi,
                 double pituus,
                 double paino)
```

Luo uuden yksilön.

Parameters:

`nimi` - henkilön nimi
`pituus` - henkilön pituus
`paino` - henkilön paino

Method Detail

BMIlkm

```
public int BMIlkm()
```

Palauttaa henkilöiden lukumäärän.

Returns:

1

---

haePituus

`public double` **`haePituus`**`()`

Palauttaa henkilön pituuden.

Returns:

pituus

---

haePaino

`public double` **`haePaino`**`()`

Palauttaa henkilön painon.

Returns:

paino

---

haeBMI

`public double` **`haeBMI`**`()`

Palauttaa henkilön painoindeksin.

Returns:

painoindeksi

---

haeBMIyksilo

`public painoindeksi.BMI` **`haeBMIyksilo`**`(int numero)`

Palauttaa järjestysnumeron mukaisen henkilön.

Parameters:

`numero` - järjestysnumero

Returns:

itsensä, jos numero on 0
null, jos numero on erisuuri kuin 0

---

lisaa

`public boolean` **`lisaa`**`(painoindeksi.BMI lisattavaBMI)`

Dummy-metodi. Toteuttaa abstraktin luokan vaatiman metodin. Yksilö ei voi sisältää aliyksilöitä.

Parameters:

`lisattavaBMI` - lisättävä yksilö/joukko

Returns:

false

---

poista

`public boolean` **`poista`**`(int numero)`

Dummy-metodi. Toteuttaa abstraktin luokan vaatiman metodin. Yksilö ei voi sisältää aliyksilöitä, joten niiden poistaminenkaan ei ole mahdollista.

Parameters:

`numero` - poistettavan yksilön/joukon järjestysnumero

Returns:

false

---

luoListaiteraattori

```
public java.util.ListIterator luoListaiteraattori()
```

Dummy-metodi. Toteuttaa abstraktin luokan vaatiman metodin. Yksilö ei voi sisältää linkitettyä listaa.

Returns:

null

Class BMInakyma

public class **BMInakyma**
extends javax.swing.JFrame

MVC-arkkitehtuurin nakyma (view). Luo pääikkunan, asettaa siihen lista- ja tilastonäkymät.

Constructor Detail

BMInakyma

```
public BMInakyma(java.util.Observable t)
```

Luo lista- ja tilastonäkymän. Tekee niistä tarkkailijoita. Kutsuu jbInit funktiota.

Parameters:

`t` - tarkkailija

Method Detail

jbInit

```
private void jbInit()
```

Lisää lista- ja tilastonäkymän välilehdille, asettaa ikkunan koon ja perusominaisuuksia. JBuilder Designerin luoma funktio, joten sen nimeä ei kannata muuttaa.

Throws:

`Exception` -

---

haeLisaaIkkuna

```
public painoindeksi.BMIlisaa haeLisaaIkkuna()
```

Palauttaa henkilöiden lisäysikkunan.

Returns:
lisaysikkuna

---

haeValittuRivi

```
public int haeValittuRivi()
```
Palauttaa listanäkymässä valitun rivin.
Returns:
valittu rivi

---

asetaOhjain

```
public void asetaOhjain(painoindeksi.BMIohjain o)
```
Asettaa ohjaimen kuuntelemaan listanäkymän ja lisäysikkunan painikkeita.
Parameters:
`o` - asetettava ohjain

Class BMItilastonakyma

public class **BMItilastonakyma**
extends javax.swing.JPanel
implements java.util.Observer
Luo tilastonäkymän, joka sisältää pylväsdiagrammin joukkojen painoindeksien keskiarvoista. Tarkkailee mallin muutoksia.

Constructor Detail

BMItilastonakyma

```
public BMItilastonakyma(java.util.Observable tarkkailtava)
```
Asetetaan tilastonäkymän komponentit paikalleen.
Parameters:
`tarkkailtava` - malli, jota tarkkaillaan

Method Detail

update

```
public void update(java.util.Observable o,
                   java.lang.Object arg)
```
Päivittää diagrammin pylväiden pituudet ja tekstit, kun malli on muuttunut.
Parameters:
`o` - tarkkailtava
`arg` - objekti, joka lisättiin/poistettiin

Class BMIlistanakyma

public class **BMIlistanakyma**
extends javax.swing.JPanel
implements java.util.Observer

Luo listanäkymän, joka sisältää listan henkilöistä, sekä lisää- ja poista-painikkeet. Tarkkailee mallin muutoksia.

Constructor Detail

BMIlistanakyma

```
public BMIlistanakyma(java.util.Observable tarkkailtava)
```

Asettelee listan, painikkeet paikoilleen

Parameters:

`tarkkailtava` - malli, jota tarkkaillaan

Method Detail

asetaOhjain

```
public void asetaOhjain(painoindeksi.BMIohjain o)
```

Asettaa painikkeiden tapahtumakuuntelijaksi ohjaimen.

Parameters:

`o` - ohjain

---

haeValittuRivi

```
public int haeValittuRivi()
```

Palauttaa taulukosta valitun rivin numeron

Returns:

rivinumero

---

update

```
public void update(java.util.Observable o,
                   java.lang.Object arg)
```

Päivittää taulukon, kun malli on muuttunut

Parameters:

`o` - tarkkailtava

`arg` - objekti, joka lisättiin/poistettiin

Class BMIlisaa

public class **BMIlisaa**
extends javax.swing.JFrame

Luo lisäysikkunan, jonka avulla saadaan luotua uusia yksilöitä.

Constructor Detail

BMIlisaa

`public` **`BMIlisaa`**`()`

Käynnistää jbInit-funktion.

Method Detail

asetaOhjain

`public void` **`asetaOhjain`**`(painoindeksi.BMIohjain o)`

Asettaa painikkeiden tapahtumakuuntelijaksi ohjaimen.

Parameters:

`o` - ohjain

---

haeNimi

`public java.lang.String` **`haeNimi`**`()`

Palauttaa tekstikenttään syötetyn nimen.

Returns:

nimi

---

haePituus

`public double` **`haePituus`**`()`

Palauttaa tekstikenttään syötetyn pituuden.

Returns:

pituus

---

haePaino

`public double` **`haePaino`**`()`

Palauttaa tekstikenttään syötetyn painon.

Returns:

paino

---

onMies

`public boolean` **`onMies`**`()`

Returns:

true, jos kyseessä on mies
false, jos kyseessä on nainen

---

jbInit

```
private void jbInit()
```
Asettaa lisäysikkunan perusominaisuudet.

Throws:

`Exception` -

Class BMIohjain

public class **BMIohjain**
implements java.awt.event.ActionListener

Luodaan MVC-arkkitehtuurin mukainen ohjain (controller).

Constructor Detail

BMIohjain

```
public BMIohjain(painoindeksi.BMInakyma n,
                 painoindeksi.BMImalli m)
```
Luo mallille alijoukot, miehet ja naiset.

Parameters:

`n` - nakyma
`m` - malli
`t` - tarkkailtava

Method Detail

haeLkm

```
public int haeLkm()
```
Palauttaa mallin yksilöiden (henkilöt) lukumäärän.

Returns:

yksilöiden lukumäärä

---

haeNimi

```
public java.lang.String haeNimi(int indeksi)
```
Palauttaa mallin indeksiä vastaavan yksilön nimen.

Parameters:

`indeksi` - yksilön järjestysnumero

Returns:

nimi

---

haePaino

```
public double haePaino(int indeksi)
```
Palauttaa mallin indeksiä vastaavan yksilön painon

Parameters:
`indeksi` - yksilön järjestysnumero
Returns:
paino

---

haePituus

`public double` **`haePituus`**`(int indeksi)`
Palauttaa mallin indeksiä vastaavan yksilön pituuden.
Parameters:
`indeksi` - yksilön järjestysnumero
Returns:
pituus

---

haeBMI

`public double` **`haeBMI`**`(int indeksi)`
Palauttaa mallin indeksiä vastaavan yksilön painoindeksin.
Parameters:
`indeksi` - yksilön järjestysnumero
Returns:
painoindeksi

---

haeSukupuoli

`public java.lang.String` **`haeSukupuoli`**`(int indeksi)`
Palauttaa mallin indeksiä vastaavan yksilön sukupuolen.
Parameters:
`indeksi` - yksilön järjestysnumero
Returns:
"mies" tai "nainen"

---

actionPerformed

`public void` **`actionPerformed`**`(java.awt.event.ActionEvent e)`
Painettaessa listanäkymän lisää-nappulaa, avaa lisäysikkunan. Painettaessa listanäkymän poista-nappulaa, poistaa valitun yksilön. Painettaessa lisäysikkunassa ok-nappulaa, lisää malliin yksilön ja sulkee lisäysikkunan. Painettaessa lisäysikkunassa peruuta-nappulaa, sulkee lisäysikkunan.
Parameters:
`e` - ActionEvent
