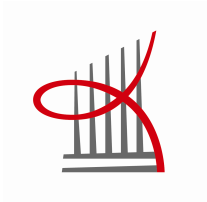


MVC-arkkitehtuurin käyttö AJAX-sovelluksessa

Kare Koho

Opinnäytetyö
Joulukuu 2012
Tietojenkäsittelyn koulutusohjelma
Tampereen ammattikorkeakoulu

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma

KOHO, KARE:
MVC-arkkitehtuurin käyttö AJAX-sovelluksessa

Opinnäytetyö 75 sivua, josta liitteitä 1 sivu
Joulukuu 2012

Tämän opinnäytetyön tavoitteena on tutkia, kuinka ja missä tilanteessa MVC-arkkitehtuuria voidaan hyödyntää web-palvelun asiakassovelluksessa. MVC-arkkitehtuuri ja sen variantit ovat pitkään olleet de facto -standardin asemassa työpöytäsovellusten kehityksessä. MVC:tä voidaan hyödyntää myös web-asiakassovelluksissa, mikäli käytetään tekniikkaa, jonka avulla voidaan ajonaikaisesti suorittaa HTTP-kutsu ja myös ajonaikaisesti päivittää käyttöliittymää vastaanotetulla datalla. Yksi tällainen tekniikka on AJAX, jonka yksi etu muihin saatavilla oleviin tekniikoihin nähden on se, ettei sen käyttö vaadi lisäosien asentamista selaimeen.

AJAX:ia hyödyntävässä web-asiakkassovelluksessa voidaan hyödyntää hyvin pitkälle työpöytäsovelluksista tuttuja ratkaisumalleja, kuten MVC-arkkitehtuuria, koska sovelluksen tieto ja ohjelman kontrolli säilyvät HTTP-kutsun aikanakin. Tässä opinnäytetyössä pyritään havainnollistamaan MVC-arkkitehtuurin rakenne ja toiminta sekä tuomaan esille sen edut ja haittapuolet, jotta lukija voi muodostaa oman käsityksensä siitä, milloin MVC:n käytöstä on hyötyä ja milloin ei.

MCV-arkkitehtuurin toteutusta ja käyttöä havainnollistettiin pienimuotoisessa esimerkkisovelluksessa, jossa sitä hyödynnetään hyvinkin erilaisissa tilanteissa. Esimerkkisovellusta toteutettaessa kävi selväksi, että MVC:n toteuttaminen niin sanotusti puhtaalta pöydältä saattaa olla työlästä ja virhealtista, mutta huolellisesti suunniteltuna ja toteutettuna se helpottaa sovelluksen laajennettavuutta, muokattavuutta, ylläpidettävyyttä ja komponenttien uudelleenkäyttöä huomattavasti. Erityisen hyödyllinen MVC-arkkitehtuuri on tilanteessa, jossa sovellus muodostaa ajonaikaisesti yhdestä tietolähteestä useita synkronoituja näkymiä ja jossa näiden näkymien lukumäärä ei ole välttämättä ennalta tiedossa.

Asiasanat: ajax, model-view-controller, rekursiokooste, tarkkailija, vastuuketju

ABSTRACT

Tampere University of Applied Sciences
Degree Programme in Business Information Systems

KOHO, KARE:
The Use of MVC Architecture in an AJAX-application

Bachelor's thesis 75 pages, appendices 1 page
December 2012

The aim of this thesis is to examine how and in which situations the client application of a web service may utilize the MVC architecture. MVC architecture and its variants have been in a position of the de facto standard in the development of desktop applications for a long time. MVC may also be applied to web client applications, if a method that allows making a run-time HTTP-request and updating the user interface with data received as well is also used. A method of this kind is AJAX, which advantage when compared to other available methods is that it does not require any plug-ins to be installed to the browser.

The same solutions, such as MCV, which are created to use in desktop applications, may fairly similarly be utilized in web client applications that make use of AJAX, because the information and control of an application is retained during the HTTP-request. This thesis aims to demonstrate the structure and operation of MVC architecture and highlight out its pros and cons so the reader may form his own conception of when applying MVC is beneficial and when it is not.

The implementation and usage of MVC architecture is demonstrated in a small-scale example application in which MVC was used in a variety of situations. When implementing the example application, it became evident that implementing MVC, so to say, from ’scratch’ may be laborious and error-prone, but when carefully designed and implemented, it notably improves the scalability, modifiability and maintainability of an application and the reuse of the components of the application. MVC is especially useful in situations where an application creates several synchronous views from a single data source in run-time and the number of these views is not necessarily known beforehand.

Key words: ajax, model-view-controller, composite, observer, chain of responsibility

SISÄLLYS

LYHENTEET JA TERMIT_….....5
1 JOHDANTO.....7
2 ARKKITEHTUURI.....10
2.1 Arkkitehtuuri 10
2.2 Suunnittelumallli.....10
3 MODEL-VIEW-CONTROLLER-ARKKITEHTUURI.....12
3.1 Johdanto..... 12
3.2 Malli (*model*).....13
3.3 Näkymä (*view*).....13
3.4 Ohjain (*controller*).....14
3.5 Tilan muutoksen levittäminen ja tarkkailu: Tarkkailija (*Observer*).....14
3.6 Toteutus.....16
3.6.1 Alustusprosessi.....16
3.6.2 Vuorovaikutusprosessi.....18
3.7 MVC:n variantit: Document-View.....20
4 MVC JA WWW-KÄYTTÖLIIITTYMÄ.....21
4.1 MVC palvelinsovelluksessa.....21
4.2 MVC asiakassovelluksessa.....22
4.3 Rekursiokooste käyttöliittymän toteutuksessa.....24
4.4 Miten ja milloin soveltaa MVC-arkkitehtuuria.....27
5 ASYNCRHONOUS JAVASCRIPT AND XML.....30
5.1 AJAX.....30
5.2 JavaScript.....31
5.3 Muut tekniikat.....34
5.3.1 DOM.....34
5.3.2 JSON.....35
5.3.3 PHP.....36
6 ESIMERKKISOVELLUS.....37
6.1 Johdanto.....37
6.2 Malli..... 38
6.3 Näkymä.....44
6.4 Ohjain.....48
6.5 Hierarkkiset näkymät ja ohjaimet.....49
6.6 Tarkkailija.....52
6.7 Tietokantarajapinta.....54
6.8 Esimerkkisovelluksen toiminta ….....55
6.8.1 Ostoskori.....55
6.8.2 Tilaus.....62
6.8.3 Myynnin graafinen esitys.....67
7 POHDINTA.....71
LÄHTEET..... 74
LIITTEET.....75
Liiite 1. Esimerkkisovellus ja sen JsDOC-dokumentaatio.....75

LYHENTEET JA TERMIT

asiakas	Se sovelluksen tai hajautetun järjestelmän osa, joka pyytää toisen osan palvelua.
asynkronisuus	Ei-realiaikaisuus. Kommunikaation osapuolet eivät ole toi sistaan ajallisesti riippuvaisia. Tietojenkäsittelyssä tämä tar koittaa sitä, että palvelun pyytäjän ei tarvitse odottaa, että palveleva osapuoli vastaa pyyntöön, vaan ohjelman suoritus jatkuu välittömästi palvelupyynnön jälkeen.
coupling	Yhteenliittäminen, kytkentä. Termillä viitataan sovelluksen komponentin riippuvaisuuteen toisista komponenteista.
change-propagation	Katso tapahtumamekanismi.
HTTP	Hypertext Transfer Protocol on sovellustason protokolla hajautettuihin, yhteistyötä tekeviin hypermediajärjestelmiin. HTTP-on yleiskäyttöinen, tilaton protokolla, jota tavallisesti käytetään tekstin ja hypertekstin siirtämiseen, mutta sen avulla voidaan siirtää myös kuvaa, ääntä ja binääristä tietoa. Eräs HTTP-omaisuus on siirrettävää dataa koskeva määrittely ja neuvottelu, joka mahdollistaa järjestelmien itsenäisen rakentumisen siirrettävästä datasta.
HTTP-pyyntö	HTTP-protokollassa määritelty menetelmä, jonka avulla asiakas pyytää palvelinta palauttamaan halutun resurssin.
HTTPS	Hypertext Transfer Protocol Secure on SSL/TLS-protokollan päälle toteutettu laajennus HTTP:stä, jonka tarkoitus on toteuttaa turvattu yhteys asiakkaan ja palvelimen välille.
jäsenmuuttuja	Olioon liitetty ominaisuus.
loose coupling	Ratkaisu, jossa sovelluksen komponentit ovat hyvin vähän tietoisia toisten komponenttien tyypistä, metodeista ja jäsenmuuttujista.
luokka	Tietorakenne, joka määrittelee tietyn käsitteen sisältämät ominaisuudet.
metodi	Olioon liitetty toiminallisuus.
olio	Luokan instanssi, ilmentymä.

synkronisuus	Realiaikaisuus, asynkronisuuden vastakohta. Palvelua pyytä vän osapuolen on odotettava kunnes pyyntöön vastataan. Ohjelman suoritus on pysähdyksissä palvelupyynnön suori tuksen aikana.
tapahtumamekanismi	Prosessi, jossa tarkkailun kohde tiedottaa tarkkailijoitaan itseään koskevasta tapahtumasta, josta tarkkailijat mahdollisesti ovat kiinnostuneita.
World Wide Web, WWW, web	Internetin osat, jotka käyttävät HTTP-protokollaa .
web-asiakas	HTTP-protokollaa käyttävä sovellus, joka suorittaa HTTP-pyyntöjä HTTP-palvelimelle. Tavallisesti selain.
web-asiakassovellus	Selaimeen toteutettu sovellus.

1 JOHDANTO

Erilaiset web-palvelut ovat nykyään osa lähes jokaisen arkipäivää: niitä käytetään sekä työssä että vapaa-aikana Moni web-palvelu on myös lähes kokonaan korvannut sen alkuperäisen reaalimaailman vastineen, esimerkiksi on vaikea perustella miksi lasku pitäisi maksaa pankin maksuautomaatissa, jos sen voi tehdä myös kotoa käsin. Kuten muutkin hyödykkeet, myös web-palvelut kehittyvät jatkuvasti ja niiltä osataan vaatia varmaa toimintaa ja helppokäyttöisyyttä. Viime vuosikymmenen puolivälissä yleistynyt AJAX-tekniikka lähestulkoon mullisti web-sovelluksen käsitteen ja toi sitä lähemmäs perinteistä työpöytäsovellusta. AJAX-tekniikka toi aivan uudenlaisia tapoja suunnitella ja toteuttaa web-sovellus, minkä seurauksena kehittäjät myös saivat koko joukon uusia mahdollisuuksia sekä myös uusia haasteita ja ongelmia.

AJAX-tekniikan yleistyttyä vaatimukset web-palvelujen käytettävyydestä ovat myös muuttuneet. Web-sovelluksilta osataan nykyään vaatia aivan toisenlaista käytettävyyttä kuin vuosikymmen takaperin. Perinteiseen synkroniseen web-kehitysmalliin liittyviä piirteitä, kuten pienestäkin muutoksesta johtuvaa palvelinpyyntöä sivun uudelleenlatauksineen, jatkuvia sivun uudelleenohjauksia ja jonkin sivun elementin latautumisen odottelua pyritään välttämään mahdollisuuksien mukaan korvaamalla ne taustalla tapahtuvilla asynkronisilla toiminnoilla.

Web-sovellus voidaan jopa tarvittaessa toteuttaa siten, että se lähestyy toiminnallisuudessa ja käytettävyydessä työpöytäsovellusta. Tästä esimerkkinä Googlen lukuisat web-palvelut, kuten Google Drive tai Gmail. Edellämainitun kaltaiset web-palvelut hyödyntävät tehokkaasti AJAX-tekniikkaa ja sen myötä suunnittelu- ja ratkaisumalleja, joista ei olisi juurikaan hyötyä, mikäli HTTP-kutsuja ei voitasi suorittaa taustatalla ja ajonaikaisesti päivittää sovelluksen käyttöliittymää vastaanotetulla datalla. Edelleen mainittujen kaltaisissa sovelluksissa voidaan kohdata myös suunniteluun ja toteutukseen liittyviä ongelmia, joita ei täysin synkronisessa web-sovelluksessa esiintyisi. Web-sovelluskehityksessä tuleekin nykyään tuntea suunnittelumalleja ja käytännön ratkaisuja laajalta alueelta, sekä palvelinsovellusten arkkitehtuurillisia ratkaisuja että tyypillisiä graafisiin käyttöliittymiin liittyviä suunnittelumalleja.

Myös web-palvelujen kehitysprosessi on muuttunut. Nykyään web-kehitys perustuu lähes aina johonkin ohjelmistokehykseen, sillä ne tarjoavat toimivan pohjan tyypillisiin toiminnallisuuksiin, jotka toistuvat lähes jokaisessa sovelluksessa. AJAX-sovellusten toteuttamiseen on niinikään tarjolla lukemattomia sovelluskehyksiä, joiden käyttötarkoitus vaihtelee pienten ja toistuvien toimintojen helpottamisesta kokonaisen sovelluksen suunnittelemiseen ja toteuttamiseen jonkin koko järjestelmän laajuisen arkkitehtuurin avulla. Sovelluskehysten käyttö on usein suositeltavaa, sillä ne nopeuttavat kehitystä, helpottavat ylläpitoa ja testausta, sekä vähentävät virheiden määrää.

Joskus kuitenkin saattaa tulla tilanteita, joissa mikään ohjelmistokehys sellaisenaan ei tarjoa riittäviä ratkaisuja sovelluksen toteuttamiseksi. Silloin vaihtoehdoksi jää muokata jotain olemassaolevaa sovelluskehystä tai toteuttaa sovellus niin sanotusti puhtaalta pöydältä.

Riippumatta siitä kumpaan ratkaisuun päädytään, on suunnittelumallien ja arkkitehtuurien tuntemisesta suuresti hyötyä, sillä ne antavat korkean tason abstraktionäkymän ohjelmistoon, mikä mahdollistaa laajojen ja monimutkaistenkin järjestelmien tarkastelun ja arvioinnin kriittisten tekijöiden osalta jo hyvin varhaisessa vaiheessa ohjelmistokehitystä jolloin järjestelmän muuntaminen on vielä halpaa ja helppoa.

Edellämainituista syistä johtuen olen katsonut tarpeelliseksi tarkastella tässä opinnäytetyössä Model-View-Controller-arkkitehtuuria (MVC), joka on ollut graafisten käyttöliittymien suunnittelun pohja ja eräänlainen käytännön standardi jo parin vuosikymmenen ajan. Niiden periaatteiden tunteminen, joihin MVC perustuu, mahdollistaa sen tehokkaan soveltamisen sekä estää myös käyttämästä sitä silloin kun siitä ei ole hyötyä. MVC-arkkitehtuurissa on omat kiistattomat etunsa, mutta myös omat haittapuolensa. Siksi MVC:n tunteminen on tarpeellista myös web-sovelluskehityksessä, jotta sitä voidaan hyödyntää, mikäli eteen tulee tilanne, jossa siitä on hyötyä.

Laajoissa ja monimutkaisissa web-palveluissa MVC-arkkitehtuuri tarjoaa pohjan sovelluksen käyttöliittymän sekä toiminnan suunnitteluun ja kehittämiseen, koska se eristää sovelluksen tiedon ja sen esittämisen omiin komponentteihinsa ja näin mahdollistaa sovelluksen tiedon uudelleenkäytön lähes joka tilanteessa. Jos MVC otetaan sovelluksen

käyttöliittymän suunnittelun pohjaksi, vältytään todennäköisesti hankalilta korjausoperaatiolta ja vaikeasti ylläpidettävältä ohjelmakoodilta kun sovellusta kehitetään sen elinkaaren aikana.

Tämän opinnäytetyön tavoitteena on selvittää mikä on MVC-arkkitehtuuri, miten se toteutetaan web-asiakassovellukseen ja miten sitä voidaan hyödyntää web-asiakassovelluksessa. Opinnäytetyön tarkoituksena on tuottaa pienimuotoinen web-sovellus, jonka avulla tarkastellaan MVC-arkkitehtuurin toteutusta ja hyödyntämistä web-asiakassovelluksessa.

2 ARKKITEHTUURI JA SUUNNITTELUMALLI

2.1 Arkkitehtuuri

Ohjelmistoarkkitehtuurilla tarkoitetaan järjestelmän perusorganisaatiota, joka sisältää järjestelmän osat, niiden keskinäiset suhteet ja niiden suhteet ympäristöön sekä periaatteet, jotka ohjaavat järjestelmän suunnittelua ja evoluutiota.

Arkkitehtuuri ei ainoastaan jaa järjestelmää osiin, vaan se myös määrittelee osien väliset suhteet ja kuinka ne kehittyvät. Koska järjestelmän osien väliset suhteet ovat usein luonteeltaan ajonaikaiseen käyttäytymiseen liittyviä, arkkitehtuuri määrittelee myös järjestelmän käyttäytymisen. Toisaalta arkkitehtuuri ei koske ainoastaan ohjelmiston staattista rakennetta (koodin rakennetta), vaan myös suorituksen aikaisia rakenteita, esimerkiksi dynaamisia oliorakenteita. Säännöt, joita tietyn arkkitehtuurin mukaan rakennettavassa järjestelmässä on noudatettava, voivat koskea esimerkiksi teknologian käyttöä (esim.on käytettävä JavaBeans-komponenttteja), algoritmien valintaa tai suunnittelumallien käyttöä (esim. kommunikointiin on käytettävä Observer-suunnittelumallia).

Arkkitehtuuri määrittelee siis järjestelmän ytimen, joka pysyy olennaisesti samana kehityksen ja ylläpidon aikana. Niitä osia, jotka eivät kuulu tähän ytimeen, voidaan vapaasti muunnella tarkoituksenmukaisemmiksi. (Koskimies & Mikkonen 2005, 19.)

2.2 Suunnittelumalli

Suunnittelumalli on tunnetun ja käytännössä hyväksi havaitun ratkaisun kuvaus ohjelmiston suunnittelua koskevaan ongelmaan tietyssä tilanteessa. Ei ole aina aivan selvää milloin jokin ratkaisuperiaate on arkkitehtuuri ja milloin suunnittelumalli, erityisesti silloin kun jokin suunnittelumalli, esimerkiksi Observer, yleistetään arkkitehtuurin perustaksi . Periaatteellisena erona voidaan pitää kuitenkin sitä, että suunnittelumalli esiintyy usein järjestelmässä monena ilmentymänä ratkaisten paikallisia suunnitteluongelmia, kun taas arkkitehtuuri määrää järjestelmän kokonaisrakenteen. (Koskimies & Mikkonen 2005, 126.) Lisäksi suunnittelumalli voidaan tietyissä tapauksissa vaihtaa toiseen ilman

suurta vaivaa, ja ilman että järjestelmän toiminta muuttuu ei-toivotulla tavalla. Esimerkiksi Facade-suunnittelumalli voidaan vaihtaa Adapter-malliin, jos rajapinnan käyttäytyminen halutaan muuttaa polymorfiseksi.

Tässä opinnäytetyössä tarkastellaan kahta suunnittelumallia: Tarkkailija (*Observer*) ja Rekursiokooste (*Composite*), joista ensiksimainittu luetaan käyttäytymismalleihin ja jälkimmäinen rakenteellisiin malleihin. MVC-arkkitehtuuri on esimerkki suunnittelumallin yleistämisestä koko järjestelmään koskevaksi arkkitehtuuriksi. Sen rakenteen ja käyttäytymisen pohjana toimii Tarkkailija-malli. Itseasiassa Tarkkailija-malli esiteltiin ensimmäisen kerran MVC-arkkitehtuurin yhteydessä. Tarkkailija on yksi tavallisimmista tavoista toteuttaa järjestelmä, jossa vaihtuvankokoinen joukko päivittyy automaattisesti kun järjestelmän tilassa tapahtuu jokin muutos.

Rekursiokoosteen avulla luodaan hierarkisia rakenteita, joissa yksittäisiä komponetteja ja koosterakenteita (*composite*) käsitellään yhdenmukaisesti, siten ettei koosteen sisäisen hierarkian ja rakenteen tarvitse olla käyttäjän tiedossa. MVC-arkkitehtuuri hyödyntää rekursiokoostetta esimerkiksi valikkorakenteissa, jotka usein ovat hierarkkisia, sisältäen sekä koosterakenteita (valikko, alavalikko), että primitiivisiä komponentteja (valikkokomento).

3 MODEL-VIEW-CONTROLLER-ARKKITEHTUURI

3.1 Johdanto

Graafisiin käyttöliittymiin kohdistuu usein erilaisia muutosvaatimuksia. Kun sovelluksen toiminnallisuutta laajennetaan, täytyy muuttaa esimerkiksi käyttöliittymän, jotta uuden toiminnot olisivat käytettävissä. Myös käyttäjät asettavat erilaisia vaatimuksia käyttöliittymälle. Yhdellä on tarve tallentaa informaatiota tekstimuodossa, toinen käyttäjä taas haluaa käyttää samaa sovellusta pääasiassa klikkailemalla ikoneita ja painikkeita. Jos käyttöliittymä on tiukasti toteutettu sovelluksen toiminalliseen ytimeen, sovelluksen ytimen tai käyttöliittymän laajentaminen voi osoittautua hyvin hankalaksi, virhealttiiksi ja kalliiksi. Edelleen sovelluksen ytimen ja käyttöliittymän integraatio saattaa johtaa tilanteeseen, jossa joudutaan ylläpitämään useita lähes samanlaisia sovelluksia, yksi kutakin käyttöliittymätoteutusta varten.

Järjestelmän arkkitehtuurin täytyy siis pystyä toteuttamaan seuraavat vaatimukset:

- Sama informaatio voidaan esittää eri muodossa samanaikaisesti ja synkronoidusti, esimerkiksi pylväs- tai ympyrädiagrammina
- Sovelluksen tuottaman visuaalisen informaation ja käyttäytymisen täytyy reagoida sovelluksen sisäisen tilan tai datan muutoksiin välittömästi
- Muutosten tekeminen käyttöliittymään tulisi olla helppoa, jopa ajonaikaisesti mahdollista.

Tähän ongelmaan ratkaisuksi on vakiintunut Model-View-Controller-arkkitehtuuri (MVC), joka ensimmäisen kerran esiteltiin SmallTalk-80-kehitysympäristön yhteydessä. Ideana on jakaa järjestelmä kolmentyyppisiin osiin: malleihin *(model)*, näkymiin *(view)* ja ohjaimiin *(controller)*.

MVC-arkkitehtuuri voidaan lukea kerrosarkkitehtuureihin, koska siinä on erotettavissa kaksi tasoa, joista ylempi käyttää hyväkseen alemman tason palveluja ja tasot voidaan järjestää laskevaan järjestykseen abstrahointiperiaatteella ihminen/laite. Ylempää tasoa edustavat näkymä ja ohjain, sillä ne ovat lähinnä käyttäjää. Alempaa tasoa edustaa mal-

li, sillä malli ei ole välittömästi tekemisissä käyttäjän kanssa ja ylempi taso käyttää mallin palveluja.

3.2 Malli *(model)*

MVC:n malli komponentti on sovelluksen toiminnallinen ydin. Malli kapseloi itseensä tarvittavan datan ja tarjoaa proseduurit, joilla toteutetaan sovelluksen toiminnallisuus. Ohjaimet toimivat välittäjinä käyttäjän ja mallin proseduurien välillä. Malli myös tarjoaa metodit, joiden avulla näkymä-komponentit saavat käyttöönsä käyttäjälle näytettävän datan.

Malli ylläpitää rekisteriä komponenteista, jotka ovat mallista riippuvaisia. Kaikki näkymät ja tietyt ohjaimet lisäävät itsensä tähän rekisteriin. Muutos mallin tilassa laukaisee tapahtuman, jota kutsutaan englanninkielisessä termillä *change-propagation mechanism* eli suoraan suomennettuna tapahtumamekanismi. Käytämme silti tästä eteenpäin termiä *tapahtumamekanismi*, joka lienee vakiintunut suomenkielinen vastine termille. Tapahtumamekanismi on ainoa yhteys mallin ja sen näkymien sekä ohjainten välillä. Kun jokin sisäinen muutos mallin tilassa laukaisee tapahtumamekanismin, malli kutsuu metodiaan *notify,* joka iteroi näkymistä ja ohjaimista, eli tarkkailijoista koostuvan listarakenteen läpi ja kutsuu jokaisen tarkkailijan kohdalla sen päivitysproseduuria, jonka nimeksi sovimme *update*. Näkymiä ja ohjaimia kutsutaan tarkkailijoiksi, sillä niiden toiminta on riippuvainen mallin tilasta ja ne siksi tarkkailevat sen toimintaa.

3.3 Näkymä *(view)*

Näkymä-komponentin tarkoitus on esittää mallin tieto käyttäjälle. Eri näkymät esittävät saman tiedon eri tavalla. Esimerkiksi myynnin vuosittaista volyymia voidaan kuvata viiva- tai pylväsdiagrammilla. Jokainen näkymä toteuttaa päivitys-proseduurin (*update*), jota mallin tapahtumamekanismi kutsuu. *Update*-proseduurissa näkymä pyytää mallilta haluamansa datan ja esittää sen käyttäjälle.

Näkymän alustuksen yhteydessä näkymä luo ohjaimensa ja tämän jälkeen liittää itsensä mallin tapahtumamekanismiin. Näkymien ja ohjainten välillä on yksi yhteen suhde eli näkymällä on vain yksi ohjain kerrallaan. Näkymä voi kuitenkin vaihtaa ohjaintaan tarpeen vaatiessa. Siksi näkymä-luokkaan toteutetaan metodi, joka vaihtaa ohjainta. Usein tämä toteutetaan siten, että metodi ottaa parametrina vastaan uuden ohjaimen, irrottaa vanhan ohjaimen mallin tapahtumamekanismista ja liittää uuden ohjaimen malliin,.

3.4 Ohjain *(controller)*

Ohjain-komponentit vastaanottavat käyttäjän syötteitä ja kutsuvat mallin palveluja. Mikäli *ohjaimen toiminta on riippuvainen mallin tilasta*, se rekisteröi itsensä mallin tapahtumamekanismiin ja toteuttaa päivitys-proseduurin (*update*), jota mallin kutsuu. Esimerkiksi tekstinkäsittelyohjelman malli laukaisee tapahtumamekanismin, kun käyttäjä muuttaa asiakirjan sisältöä. Ohjain reagoi tähän aktivoimalla *Tallenna*-painikkeen. Jos käyttäjä tallentaa muutokset, *Tallenna*-painike deaktivoidaan jälleen. Jos ohjaimen *ei mitenkään tarvitse reagoida mallin tilan muutoksiin*, sitä ei tarvitse lisätä mallin tapahtumamekanismiin.. Tällöin voidaan toteuttaa käyttäjän syötteiden käsittely ja mallin palvelujen kutsuminen näkymäluokkaan.

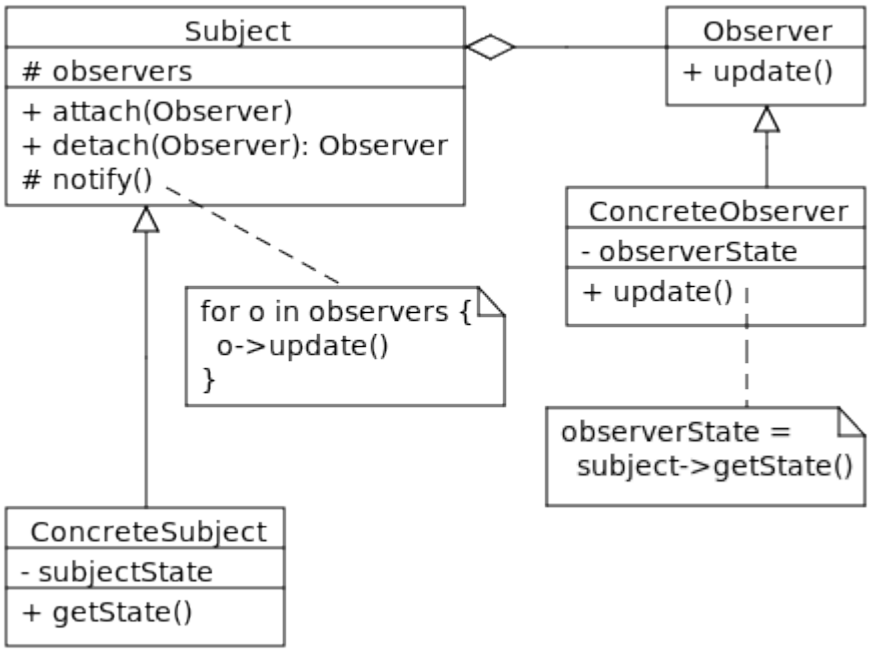
Ohjaimen keskeinen tehtävä on siis sovelluksen toiminnan ja käyttäytymisen muuntelu. Ohjain ei muuta sääntöjä joiden mukaan malli toimii, vaan sovelluksen toiminnan muuntelu tapahtuu muokkaamalla käyttöliittymää (kontrollien aktivointi/deaktivointi) tai muuttamalla tapaa, jolla käyttäjän syötteisiin reagoidaan. Tietyssä tilanteessa ohjain voidaan vaihtaa toiseen ohjaimeen, joka reagoi eri tavalla mallin tilan muutoksiin tai käyttäjän toimenpiteisiin. Sovelluksen ajonaikainen toiminnan muuntelu on siis erillisen ohjainkomponentin idea.

3.5 Tilan muutoksen levittäminen ja tarkkailu: Tarkkailija *(Observer)*

Tarkkailija on käyttäytymismalleihin lukeutuva suunnittelumalli, jota voidaan soveltaa missä tahansa seuraavista tilanteista:

- Käsitteellä on kaksi osapuolta, josta toinen on riippuvainen toisesta. Eristämällä nämä osapuolet omiin olioihinsa niitä voidaan muuttaa ja uudelleenkäyttää kumpaakin osapuolta itsenäisesti.
- Kun muutos osapuoleen vaatii muutoksia toisiin osapuoliin, eikä ole tiedossa kuinka montaa osapuolta pitää muuttaa.
- Kun olion täytyy pystyä tiedottamaan toisia olioita tekemättä oletuksia siitä, minkä tyyppisiä muut oliot ovat. Toisin sanoen halutaan välttää *tight coupling*-tilannetta. (Gamma, Helm, Johnson & Vlisides 1997, 294.)

MVC suunnittelumallissa näkymillä ja ohjaimilla on *tarkkailijan* rooli (o*bserver*) ja malli puolestaan on *tarkkailun kohde* (*subject*). Alla olevasta luokkakaaviosta käy ilmi Tarkkailija-mallin rakenne.



KUVA 1. Tarkkailija-suunnittelumallin luokkakaavio.

Malliluokka ylläpitää tietorakennetta tarkkailijoistaan ja kun tulee tilanne, joka vaatii tarkkailijoiden tiedottamista, malli kutsuu metodia *notify*, jossa iteroidaan läpi tarkkailija-olioista koostuvan listarakenne ja kutsutaan jokaisen tarkkailijan kohdalla sen meto-

dia *update*. *Update*-metodissaan tarkkailijat voivat pyytää mallilta tarkempaa tietoa sen tilasta kutsumalla tarkkailijan metodia *getState.* Mikäli mallin tilan muutos jollain tavalla koskettaa tarkkailijaa, se suorittaa tarvittavat toimenpiteet.

Tarkkailijoilla on tavallisesti yhteinen kantaluokka, esimerkiksi *Observer*, johon on toteutettu *update*-metodi ilman operaatioita. MVC:ssä näkymä ja ohjain perivät tällaisen kantaluokan, jotta vaadittu rajapinta tarkkailijoiden osalta toteutuu. Mallin tulee toteuttaa edellä mainitun *notify*-metodin lisäksi metodit *attach ja detach. Attach* liittää tarkkailijan tarkkailija-olioista koostuvaan listarakenteeseen ja *detach* puolestaan poistaa tarkkailijan.

3.6 Toteutus

3.6.1 Alustusprosessi

MVC-triadin alustus aloitetaan luomalla malli-olio ja näkymä-olio, joka ottaa parametrikseen viitteen mallista. Tämä tapahtuu tavallisesti MVC-triadin ulkopuolella.

```
Model model = new Model();  // Luo malli.
View view = new View(model);  // Luo näkymä ja anna viite malliin.
```

Näkymä ottaa rakentimessaan parametrinä viitteen malliin ja liittää itsensä mallin tapahtumamekanismiin.

```
class View {
    private Model model;
    private Controller controller;
    ...
    View (Model model) {
        model.attach(this); // Näkymä liittää itsensä malliin.
        this.model = model; // Aseta viite malliin.
    }
}
```

Näkymä jatkaa alustusprosessiaan luomalla ohjaimensa. Näkymään toteutetaan ohjaimen luomista varten metodi, jonka perittäessä ylikirjoitetaan. Näkymän ohjainta ei voi

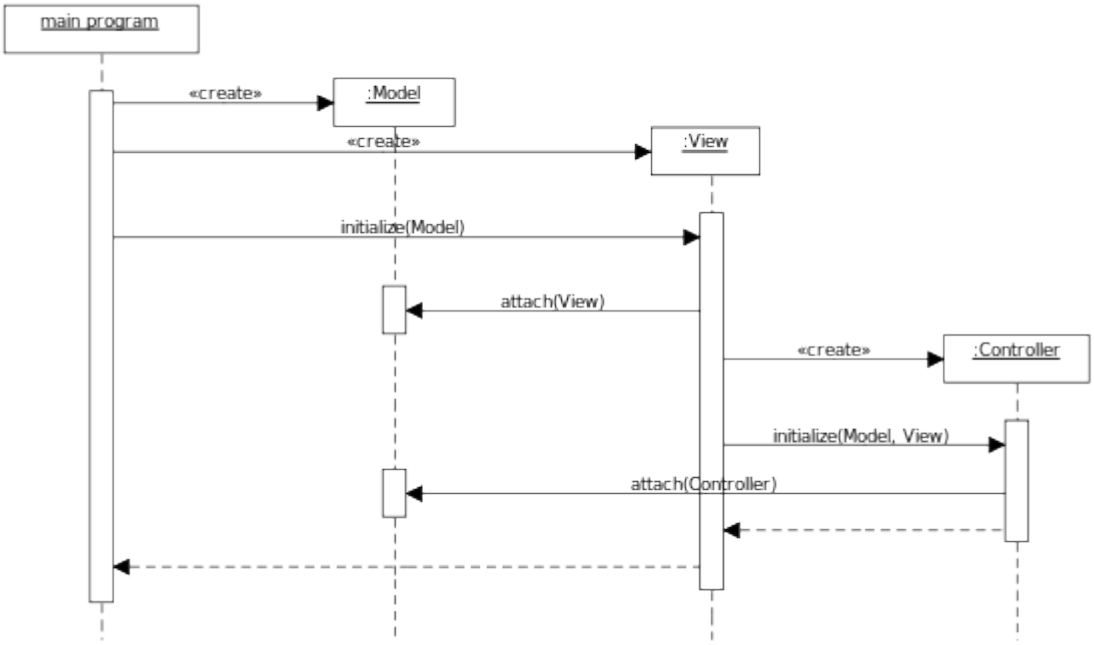
luoda rakentimessa, koska tällöin näkymää perittäessä ohjain luotaisiin useaan kertaan. Metodi *initalize* voi toteuttaa myös muita alustuksessa tarvittavia toimenpiteitä ja myös se ylikirjoitetaan tarvittaessa.

```
view.initialize();
...
Class View {
   ...
   public void initialize () {
       controller = makeController(); // Luo ohjain.
       // Muita alustusproseduureja...
   }
   protected Controller makeController () {
       // Kutsu ohjaimen rakenninta ja anna viite näkymään.
       return new Controller(this);
   }
}
```

Ohjaimen rakennin ottaa parametriksi viitteen näkymään. Ohjain saa mallinsa näkymältä. Jos ohjain liitetään malliin, se tapahtuu ohjaimen rakentimessa.

```
Class Controller {
    private View view;
    private Model model;
    ...
    Controller (View view) {
        this.view = view; // Aseta viite näkymään.
        model = view.getModel(); // Pyydä malli näkymältä.
        model.attach(this); // Ohjain liittää itsenä malliin.
    }
}
```

Alustuksen jälkeen sovellus voi aloittaa käyttäjän syötteiden prosessoinnin.



KUVA 2. MVC-triadin alustus.

3.6.2 Vuorovaikutuprosessi

Tyypillinen tilanne, jossa käyttäjän syöte laukaisee tapahtumamekanismin, alkaa ohjaimen vastaanottamasta tapahtumasta:

```
Class Controller {
    private Model model;
    ...
    public void handleEvent (event) {
       model.service(event.data); // Kutsu mallin palvelua.
    }
}
```

Malli suorittaa pyydetyn palvelun, joka muuttaa mallin sisäistä tilaa.
Sen seurauksena malli kutsuu jokaisen rekisteröiden näkymän ja ohjaimen päivitysproseduuria.

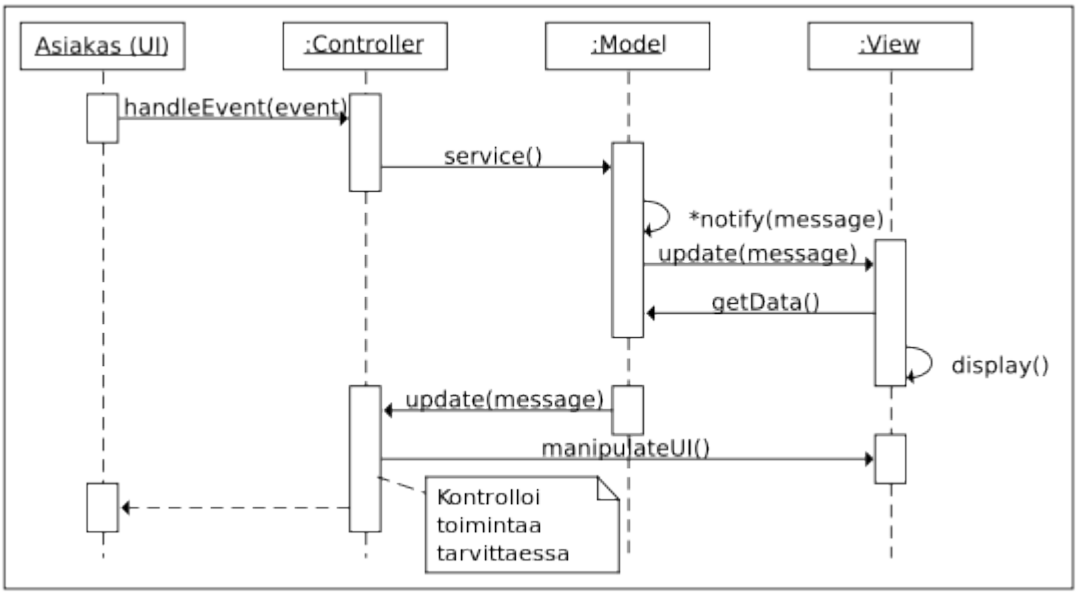
```
Class Model {
    private Vector<Observer> observers;
    ...
    public void service () {
        // Suorita tarvittavat toimenpiteet ...
        notify(); // Kutsu tapahtumamekanismia.
    }

    private void notify () {
        for (Observer o in observers) {
            o.update(); // Kutsu tarkkailijan update-metodia.
        }
    }
}
```

Jokainen näkymä pyytää mallilta datan ja päivittää itsensä näytölle.

```
Class View {
    private Model model;
    ...
    public void update () {
        draw ( model.getData() ); // Päivitä näkymää.
    }
}
```

Jokainen rekisteröity ohjain pyytää mallilta datan, jotta voi tarvittaessa muokata käyttöliittymää. Esimerkiksi tallennuspainikkeen aktivoiminen voi olla seurauksena mallin tilan muuttumisesta.

```
Class Controller {
    private Model model;
    private View view;
    ...
    public void update () {
        if ('modified' == model.getState()) {
            view.activateSomething(); // Muokkaa käyttöliittymää.
        }
    }
}
```



KUVA 3. Vuorovaikutus MVC-arkkitehtuurissa.

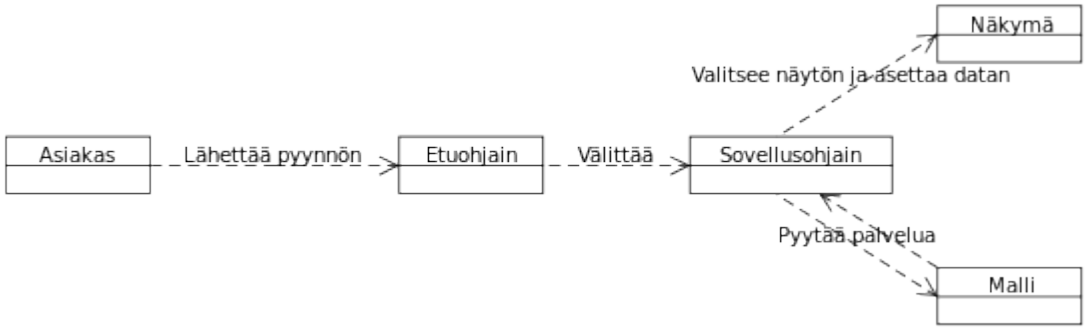
3.7 Variantit: Document-View

MVC-arkkitehtuurista on olemassa joitakin variaatioita. Document-View variantti yhdistää näkymän ja ohjaimen samaan komponenttiin (Buschmann, Meunier, Rohnert, Sommerlad & Stal 1996, 140). Document-komponentti vastaa MVC:n mallia ja View yhdistää ohjaimen ja näkymän tehtävät. MVC:ssä voidaan myös yhdistää näkymä ja ohjain samaan komponenttiin, mutta tällöin menetetään mahdollisuus vaihtaa näkymän ohjain ajonaikaisesti. MVC.n tapaan Document-View mahdollistaa useat samanaikaiset synkronoidut, mutta erilaiset näkymät.

4 MVC JA WWW-KÄYTTÖLIITTYMÄ

4.1 MVC palvelinsovelluksessa

MVC-arkkitehtuuria hyödynnetään laajalti myös palvelinsovelluksissa. Se tarjoaa pohjan myös web-palvelun suunnittelulle, jakamalla sovelluksen datan, logiikan ja tiedon esittämisen omiin komponentteihinsa. Palvelinsovelluksessa ohjaimella on keskeinen rooli, sillä se valitsee käytettävät mallit ja näkymät. Kuva 4 esittää tavanomaista palvelinsovelluksen MVC-arkkitehtuuria.



KUVA 4. Tyypillinen MVC-arkkitehtuuri palvelinsovelluksessa.

Kuten kuva 4 esittää, kaikki asiakkaan palvelupyynnöt kohdistuvat aluksi niinkutsuttuun etuohjaimeen (*front controller*), joka valitsee sopivan sovellustason ohjaimen. Sovellustason ohjain valitsee mallin ja kutsuu sen palvelua. Kun palvelupyyntö on käsitelty ohjain valitsee sopivan näkymän ja asettaa datan sen saataville. Lopuksi näkymä palauttaa datan asiakkaalle. Tätä versiota MVC:stä kutsutaan yleensä nimellä Model 2 tai MVC Model 2.

Esimerkiksi Struts-sovelluskehyksessä HTTP-pyynnön http://some.service/products.action käsittely voitaisiin ohjata *ProductAction*-ohjainluokalle, ja edelleen kutsua sen *listProdcuts*-metodia. *ProductAction.listProducts*-metodissa kutsutaan mallin proseduuria ja lopuksi valittu näkymä esittää datan asiakkaalle.

Web-asiakas voi HTTP-pyynnöissään, varsinkin AJAX-sovelluksissa, vaatia vastauksena erilaisia MIME-tyyppejä, kuten XML, JSON tai ASCII-teksti. HTTP-pyynnön otsikon *Accept*-kentästä ohjain voi päätellä missä formaatissa asiakas haluaa vastauksen olevan ja sen mukaisesti muuntaa datan oikeanlaiseksi. Esimerkiksi *Accept: application/json, text/javascript, */*; q=0.01* tarkoittaa, että vastaus tulisi olla JSON-formaatissa.

Web-palvelulla voi olla erityyppisiä asiakkaita. Selaimen lisäksi asiakkaana voi olla esimerkiksi MIDP-laite tai niinkutsuttu raskas asiakas (*rich client*). Useiden asiakastyyppien tukemiseksi voidaan toteuttaa erilliset etuohjaimet kutakin asiakastyyppiä varten. Asiakkaat tekevät HTTP-pyynnön itselleen tarkoitetulle etuohjaimelle.

Keskitetyn hallinnan kannalta, erityisesti tietoturvan kannalta, on helpompaa hallita sovelluksen yhdestä kohtaa. Tätä tarkoitusta varten voidaan toteuttaa *protokollareitys,* jolloin HTTP-pyynnöt tehdään asiakkaan tyypistä riippumatta samalle ohjaimelle. Protokollareitityksestä huolehtiva etuohjain päättelee asiakaan tyypin HTTP-pyynnön otsikon *User-Agent*- kentästä ja välittää pyynnön edelleen sopivalle asiakastason etuohjaimelle.

4.2 MVC web-asiakassovelluksessa

AJAX-tekniikan määrittävin piirre on asynkronisuus. Synkronisessa web-mallissa sovelluksen koko käyttöliittymä luodaan jokaisen HTTP-pyynnön yhteydessä uudelleen. Asiakkaan muistiin tallennettu tieto ei myöskään säily kutsujen välillä, ellei sitä ole tallennettu evästeisiin. Asynkronisuus muuttaa tilanteen, sillä HTTP-pyyntö voidaan nyt suorittaa taustalla omassa säikeessään ja kun pyyntöön saadaan vastaus, palvelimen palauttama data voidaan sijoittaa näkymässä haluttuun kohtaan ilman, että koko näkymää tarvitsee luoda uudelleen. Raja desktop- ja web-asiakkaan välillä vaikuttaa hämärtyvän. Kannatta kuitenkin pitää mielessä, että World Wide Web perustuu asiakas-palvelin-malliin, eikä mikään uusi asiakaspuolen teknologia muuta sitä seikkaa miksikään. Asiakas-palvelin-arkkitehtuurin ideana on kapseloida tietyn resurssin hallinta yhteen paikkaan, siten ettei asiakkaan tarvitse huolehtia resurssin käyttöön liittyvistä yksityiskohdista ja mahdollisista ongelmista. Tämä idea toimii hyvin; web-palvelut ovat käytettä-

vissä missä vain ja millä selaimella tahansa. Tästä syystä asiakkaalla ei tarvitse olla omaa dataa, vaan asiakkaan tietomalli vastaa palvelimen tietomallia tai on siitä johdettu. Asiakkaan tietomalli on vain eräänlainen näkymä palvelimen tietomalliin.

Asiakkaan malliin ei juuri kannata toteuttaa liiketoimintalogiikkaa. Tähän on monta syytä: ensinnäkin JavaScript-lähdekoodi on kaikkien luettavissa, eikä lähdekoodin minifointi tai edes hämärtäminen luotettavasti estä sen tulkitsemista. HTTP-kutsujen URL-osoitteet ovat helposti löydettävissä lähdekoodista, joten mikään ei estä pahantahtoista käyttäjää suorittamasta HTTP-pyyntöjä esim. cURL-ohjelmalla kyseisiin osoitteisiin. (Asleson & Schutta 2007, 22.) Asiakkaaseen toteutettu liiketoimintalogiikka olisi todennäköisesti toteutettava palvelinsovelmaankin.

Esimerkiksi verkkokaupassa tehtävän tilauksen loppusumma voitaisiin laskea selaimessa ennen palvelimelle lähettämistä ja tallentaa se palvelimella tietokantaan ilman uudelleenlaskemista . Asiakassovelluksen lähettämän HTTP-pyynnön sisällön saa selville esimerkiksi WireShark-ohjelmalla. Jos pyynnössä lähetetään tilauksen loppusumma, se on tällöin laskettu selaimen sovelluksessa ja sen perusteella on olemassa *mahdollisuus*, että loppusummaa ei tarkisteta palvelimella. Tämä antaa käyttäjälle tilaisuuden generoida väärennetty HTTP-pyyntö ja lähettää se tilauksen vastaanottavalle sovellukselle selaimen lähettämän pyynnön sijaan. Mikäli käytetään HTTPS-protokollaa, ei selaimen tekemien HTTP-pyyntöjen sisältö ole luettavissa, mutta JavaScript-sovelluksen lähdekoodista on yhä mahdollista selvittää mitä parametrejä sovellus generoi HTTP-pyyntöön. Edes minifointi ei tee koodin tulkitsemista teoreettisesti mahdottomaksi.

```
POST /order.php HTTP/1.1
Host: someservice.com
// Muita otsikoita...
Content-Type: application/x-www-form-urlencoded

// Pyyynnön parametreja ...
itemid[]=10&itemid[]=11&totalsum=5.0 // Väärä loppusumma
```

Edelläoleva HTTP-pyyntö lähettää tilauksen, jossa on väärä loppusumma. Tälläinen huijaus mitä luultavimmin huomattaisiin ja siitä olisi tekijälle enemmän haittaa kuin hyötyä, mutta tarkoitus onkin antaa esimerkki JavaScript- ja AJAX-sovellusten tietotur-

variskeistä. Kaikki palvelimelle saapuva data, jonka sisältöön liittyy sääntöjä tai rajoitteita, on tarkistettava myös palvelimella. Esimerkin tapauksessa tilauksen loppusumman laskeminen selaimessa olisi täysin tarpeetonta ja esimerkki huonosta suunnittelusta..

Web-asiakkaan rooli ei siis AJAX-teknologian myötä muutu, sillä asiakas-palvelin-malli on edelleen lähtökohtana. Asiakkaan käyttöliittymän toteuttamiseksi ja päivittämiseksi sen sijaan tulee vaihtoehtoisia tapoja ja käyttökokemus muuttuu niiden myötä.

Koska asiakassovelluksen elinkaari ulottuu asynkronisuuden myötä HTTP-kutsujen ylitsekin, voidaan AJAX-sovellusten käyttöliittymän suunnittelussa ja toteutuksessa halutessa hyödyntää desktop-sovelluksissa hyödynnettäviä suunnittelumalleja.

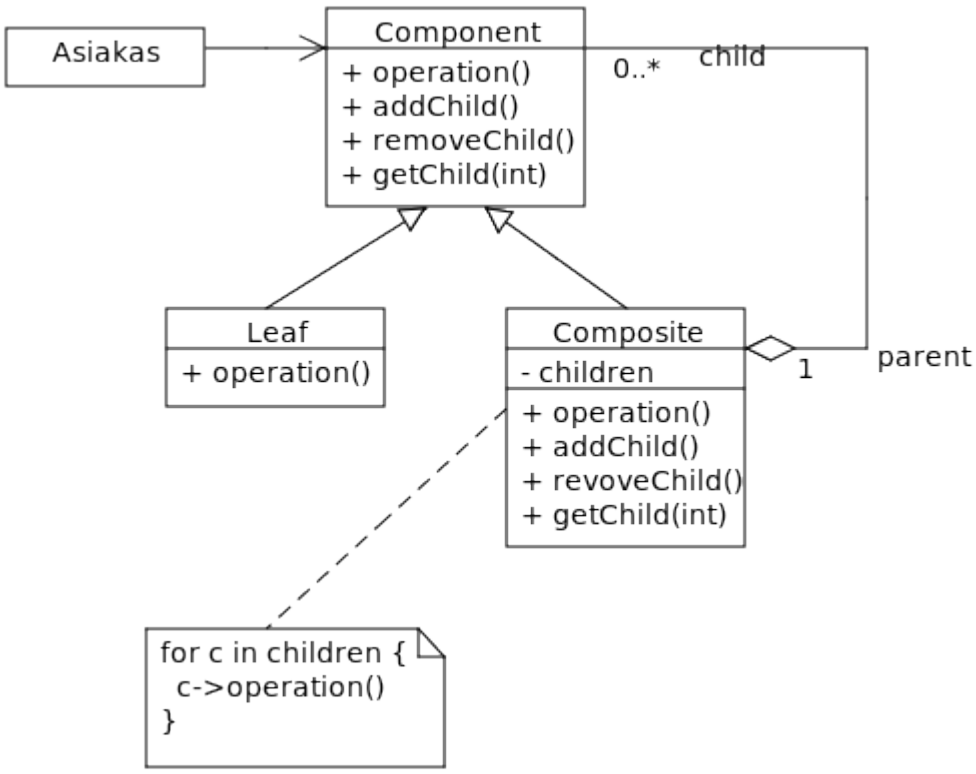
Jos tarkoitus on taata sovelluksen muokattavuus, laajennettavuus ja ylläpidettävyys pitkälle tulevaisuuteen, on MVC-arkkitehtuuri hyvä lähtökohta, koska sen keskeisiä periaatteita on sovelluksen tiedon ja logiikan eristäminen tiedon esitystavasta. Tämä on MVC:ssä toteutettu siten, että malli ei tunne tarkkailijoitaan eikä tiedä niiden lukumäärää, koska ainoa yhteys mallin ja tarkkailijoiden välillä on mallin tapahtumamekanismi. Ei ole siis mitään hyötyä toteuttaa malliin tiedon esittämistä koskevaa koodia. Muutoksen tai laajennuksen tekeminen tiettyyn näkymään tai ohjaimeen *ei pitäisi* vaikuttaa toiseen näkymään tai ohjaimeen. Muutoksen tekeminen taas malliin vaikuttaa kaikkiin tarkkailijoihin, jos niin halutaan.

4.3 Rekursiokooste-suunnittelumalli käyttöliittymän toteutuksessa

MVC-arkkitehtuurin mukaan toteutetussa web-asiakassovelluksessa on erityisen helppo muodostaa dynaamisesti luotuja näkymiä, koska malliolio säilyttää sovelluksen dataa sellaisenaan ja sillä on metodit joilla data saadaan helposti käyttöön halutussa muodossa. MVC:n näkymä- luokilla on yleensä yhteinen isäntäluokka, joka sisältää näkymän luomiseen ja tuhoamisen liittyvät yleispätevät metodit. Jos näitä metodeja joudutaan lähes poikkeuksetta ylikirjoittamaan perityissä näkymissä, jossain on todennäköisesti suunnitteluvirhe.

Kuvitellaan tilanne, jossa DOM-elementin A lapsielementit halutaan korvata uusilla lapsielementeillä B, C ja D. Web-sovelluksen näkymä-luokilla on yhteinen isäntäluokka *View*, joka lähtökohtaisesti tyhjentää elementin A sisällön ennenkuin luo sen sisään uuden näkymän eli käytännössä DOM-elementin lapsineen. Tästä seuraa tietenkin se, että B-lapsielementti luodaan vielä onnistuneesti, mutta C poistaa B:n ja D puolestaan C:n, jolloin lopputuloksena saadaan vain yksi kolmesta halutusta uudesta näkymästä eli D. Koska emme halua ylikirjoittaa *View*-luokan alustusproseduureja, on ratkaisuksi sovellettava komposiittirakennetta.

Rekursiokooste (*Composite*) on rakenteellinen suunnittelumalli. Se määrittelee joukon olioita, joita asiakas käsittelee yhtenä oliona, tuntematta oliojoukon todellista hierarkiaa. Rekursiokooste-mallin avainajatus on komponentti-rajapinta (*component*), joka edustaa sekä primitiivisiä komponentteja eli *lehtiä (leaf)*, että *koosteita (composite)*, jotka koostuvat näistä primitiiveistä. Asiakas kutsuu aina *komponentin* palvelua, eikä ole tietoinen onko palvelun suorittava komponentti kooste- vai lehtiolo. Komponentti määrittelee kaikki palvelut eli metodit, joita rekursiokoosteen instanssin on tarkoitus tarjota. Sekä lehti- että koosteoliot toteuttavat tämän rajapinnan, ja määrittelevät miten tietty palvelu käsitellään.



KUVA 5. Rekursiokooste-suunnittelumallin rakenne.

Lehti-oliolla on aina isäntä, jonka komponentti lehti on, eikä lehti voi sisältää toisia lehtiä tai kooste-olioita. Koosteolio voi sen sijaan sisältää lehtiolioita ja/tai koosteolioita, jotka puolestaan saattavat toistaa samankaltaista rakennetta hyvinkin pitkään.

Tämä rakenne tarjoaa yleisluontoisen ja suoraviivaisen ratkaisun edellä kuvattuun ongelmaan. Määrittelemme näkymän B koosteolioksi, jonka lehtiolioita ovat näkymät C ja D. Lisäksi määrittelemme, että lehtiolio ei koskaan tyhjennä sen DOM-elementin sisältöä, jonka lapsielementti se on. Koosteolio, kuten sanottu, voi olla toisen koosteolion komponentti, mutta tietenkin koko oliohierarkialla on juurielementti, jolla ei ole kuulu mihinkään toiseen koosteolioon. Tällainen juurielementti on esimerkkiongelmassa näkymä B, joka alustuksessaan tyhjentää DOM-elementin A ja sen jälkeen liittää itsensä sen lapsielementiksi.

DOM itsessään on myös rekursiokooste, mikä tekee hierarkisten näkymien käytöstä luontevaa. Kun hiearkinen näkymä poistetaan DOM-puusta, eli jokaisen näkymän ylimmäinen HTML-elementti poistetaan, ei näkymäluokkiin tarvitse toteuttaa rekursiivista DOM-elementin poisto-operaatiota, sillä DOM-metodi *element.removeChild(node)* poistaa DOM-puusta myös poistettavan elementin (*node*) lapsielementit.

4.4 Miten ja milloin soveltaa MVC-arkkitehtuuria

MVC:n toteuttaminen web-asiakassovellukseen poikkeaa jonkin verran siitä miten se toteutetaan työpöytäsovellukseen. Yksi ero on miten näkymäluokan käyttöliittymäkomponentit luodaan. Monissa web-palveluissa on viitattava resurssiin, esim. verkkokaupan tuotteeseen, URL-osoitteella, jotta hakukoneet voivat indeksoida kyseisen sivun. Myös jos vaatimuksena on, että sivu on voitava tulostaa, ei dynaamisesti luotu näkymä tule kysymykseen. Tällöin näkymäluokan käyttöliittymäkomponentit eli HTML-elementit ja mahdollisesti myös koko sivu sekä myös sovelluksen data vastaanotetaan normaalin tapaan vastauksena HTTP-pyyntöön, jolloin web-asiakkaaaseen ei tarvitse toteuttaa näkymän rakentamisproseduuria, vaan ainoastaan riittää, että näkymä etsii sivun valmiista DOM-rakenteesta näkymän juurilementin. Tämän jälkeen näkymää mahdollisesti muokataan dynaamisesti, mikäli tapahtumamekanismin kutsumiselle on tarvetta.

Näkymä voidaan myös luoda dynaamisesti, eli näkymä lisätään sovelluksen DOM-rakenteeseen sen luonnin jälkeen. Tätä tapaa on käytettävä luonnollisesti silloin kun samasta tietolähteestä muodostetaan useita näkymiä ajonaikaisesti, esimerkiksi muodostettaessa erilaisia diagrammeja samasta tilastollisesta datasta. Myös niin sanottujen single-page- sovellusten näkymät joudutaan rakentamaan dynaamisesti, koska sovelluksen DOM-rakenne vastaanotetaan kokonaisuudessaan vain kerran, sovelluksen alustuksen yhteydessä. Tämän jälkeen tehdään tyypillisesti ainoastaan asynkronisia HTTP-kutsuja.

Näkymän-luokan rakentaminen on siis erityispiirre toteutettaessa MVC:tä web-asiakkaaseen. Myös JavaScript, joka on lähes ainoa vaihtoehto MVC:n toteuttamiseen, poikkeaa melko paljon esimerkiksi C++:sta tai Javasta, joita tavallisimmin käytetään toisissa ympäristöissä MVC-sovelluksen toteuttamiseen. JavaScript on prototyyppipohjainen

ohjelmointikieli ja se tukee monia olio-ohjelmoinnin piirteitä, kuten kapselointia, jäsenmuuttujien ja metodien määrittelyä ja oliokielien tapaista jäsenmuuttujien ja metodien käyttöä, mutta siitä puuttuu varsinainen luokkamäättely. MVC:n toteuttaminen JavaScriptillä vaatii perusteellista prototyyppipohjaisen perinnän ymmärtämistä, jotta MVC:n tarjoamaa sovelluskomponenttien uudelleenkäyttöä voidaan hyödyntää tehokkaasti.

Yksi MVC:n perusajatuksista on mahdollisuus tuottaa erilaisia näkymiä samasta tietolähteestä. Web-palvelussa erityyppisiä näkymiä voidaan tuottaa jo palvelinpuolen sovelluksessa. Tällöin yksinkertaisesti tietyn käyttäjätyypin vaatimuksia varten palautetaan palvelimelta vaaditunlainen näkymä, eikä sen muokkaamiseen ajonaikaisesti ole useinkaan suurempaa tarvetta. Muutama pienehkö DOM-operaatio voidaan hyvin upottaa mihin kohtaa tahansa sovellusta ilman, että sillä olisi ylläpidettävyyden kannalta suurtakaan merkitystä. Tämä asettaa MVC:n toteuttamisen web-asiakkaaseen usein kyseenalaiseksi. MVC:n toteuttaminen web-asiakkaaseen on aina työlästä, jopa valmista sovelluskehystä käyttäen. Lisäksi MVC hidastaa sovelluksen suorituskykyä ainakin ”paperilla”, koska se lisää sovellukseen ylimääräisen arkkitehtuurillisen tason sekä Tarkkailija-mekanismin, joka lisää proseduurikutsujen tarvetta riippumatta sovelluksen kulloinkin suorittamasta tehtävästä.

Monesti ainoaksi aukottomasti perustelluksi kriteeriksi MVC:n käytölle web-asiakkaassa jää tarve muodostaa erityyppisiä näkymiä ajonaikaisesti, eikä näiden näkymien määrää voida mitenkään tietää ennalta. Tällöin MVC tai oikeammin sen toteuttama *Tarkkailija*-suunnittelumalli on yksinkertaisin ja paras ratkaisu kaikista mahdollisista haittapuolista huolimatta.

Web-asiakassovelluksessa, kuten missä tahansa muussakin sovelluksessa, jossa hyödynnetään MVC-arkkitehtuuria, on näkymä-komponentti ainoa osa sovellusta, joka sisältää käyttöliittymän luomiseen ja muokkaamiseen liittyvää ohjelmakoodia. Tämä puolestaan tarjoaa erinomaiset mahdollisuudet jatkokehittää sovelluksen käyttöliittymää, koska käyttöliittymään tehdyt muutokset eivät vaikuta sovelluksen liiketoimintalogiikkaan. Näkymä-komponentissa on melko vaivatonta hyödyntää rakenteellisia suunnittelumalleja kuten Rekursiokooste tai käyttäytymismalleja kuten Vastuuketju (Gamma E., Helm R., Johnson, R. & Vlisides J.1997, 223). Rekursiokooste ja Vastuuketju toisiinsa lähei-

sesti liittyviä suunnittelumalleja, koska Rekursiokooste jo itsessään sisältää implisiittisen vastuuketjun. MVC-arkkitehtuurissa taas on tavallista muodostaa sisäkkäisiä ja hierarkisia näkymiä eli niinsanottuja komposiittinäkymiä Rekursiokoosteen avulla.

Tämän opinnäytetyössä selvitetään miten MVC-arkkitehtuuri itsessään toteutetaan web-asiakassovellukseen käyttäen JavaScript-ohjelmointikieltä. Tarkastelun kohteena on myös tavanomaiset MVC:n yhteydessä käytettävät suunnittelumallit kuten Rekursiokooste ja Vastuuketju. MVC:n, Rekursiokoosteen ja Vastuuketjun välinen suhde on varsin hyödyllistä tuntea, koska se tarjoaa monipuoliset mahdollisuuden käyttöliittymän ja toiminnallisuuden muokkaamiseen sekä käyttöliittymäkomponenttien uudelleenkäyttöön.

5 ASYNCRHONOUS JAVASCRIPT AND XML

5.1 Asynchronous JavaScript And XML (AJAX)

AJAX on yhdistelmä tekniikoita, joiden avulla on mahdollista hakea tietoa palvelimelta ilman tarvetta ladata koko sivua uudelleen. Alkujaan akronyymi AJAX tulee sanoista Asynchronous JavaScript And XML, mutta sen yhteydessä käytettäviä tekniikoita ovat myös JavaScript, DOM ja CSS. AJAX:in keskeisin komponentti on XMLHttpRequest-rajapinta (XHR), joka toteutettiin ensimmäisen kerran ActiveX-objektina Internet Explorerin viidennen version yhteydessä keväällä 1999. XMLHttpRequest-rajapinnan toteuttaa yleisimmin WWW-selain ja selaimessa se on käytettävissä esimerkiksi JavaScriptillä.

XHR:n tarkoitus on suorittaa HTTP-pyyntöjä asynkronisesti, eli keskeyttämättä ohjelman suoritusta pyynnön ajaksi. Toimintaperiaate on se, että XHR-objektilla *onreadystatechange*-niminen callback-metodi, jota se kutsuu kun HTTP-pyynnön suorituksessa tapahtuu jokin muutos. *onreadystatechange*-metodin toteuttaminen on ohjelmoijan vastuulla. HTTP-kutsulla on viisi vaihetta, joihin viitataan XHR objektin attribuutilla readyState. ReadyState-attribuutti sisältää numeerisen arvon välillä 0-4. *onreadystatechange*-metodissa määritellään, miten toimitaan kutsun kussakin vaiheessa. Merkityksellisin on vaihe 4, joka merkitsee sitä, että vastaus HTTP-pyyntöön on vastaanotettu kokonaisuudessaan.

```
// Luo XMLHttpRequest-olio.
httpRequest = new XMLHttpRequest();
// Aseta viittaus tapahtumankäsittelijään
httpRequest.onreadystatechange = displayContents;
// Avaa yhteys resurssiin.
httpRequest.open('GET', ”somehost.com/someservice?itemid=15”);
// Lähetä pyyntö.
httpRequest.send();

function displayContents () {
    // Vastaus saatu.
    if (httpRequest.readyState == 4) {
        // Pyydetty resurssi löytynyt ja luettu.
```

```
            if (httpRequest.status == 200) {
                // Laita data HTML-sivulle.
                document.getElementById(”contents”).innerHTML =
                httpRequest.responseText;
            } else { // Tapahtui jokin virhe.
                document.getElementById(”contents”).innerHTML =
                ”Tapahtui virhe. ” + httpRequest.getAllResponseHeaders();
        } else {
            // Pyyntö kesken. Näytä esim. latausanimaatio.
        }
    }
```

Oheisessa esimerkissä suoritetaan HTTP-kutsu käyttäen GET-metodia. `httpRequest.onreadystatechange` tapahtumankäsittelijälle on määritelty funktioksi `displayContents`. ReadyState arvo 4 tarkoittaa sitä että, vastaus HTTP-pyyntöön vastaanotettu kokonaisuudessaan. Tämä ei kuitenkaan vielä merkitse sitä, että pyyntö on onnistunut toivotulla tavalla. XHR-objekilla on attribuutti *status*, joka on palvelimen palauttama HTTP-statuskoodi. Esimerkissä statuskoodi on 200, joka tarkoittaa että haluttu resurssi on löytynyt, eikä sen suorittamisessa ole tapahtunut virhettä. Vastaavasti statuskoodi 404 merkitsisi, ettei palvelin löytänyt haluttua resurssia.

POST-metodia käytettäessä query string ”`itemId=15″` sijoitettaisiin metodiin *`httpRequest.send,`* koska POST-metodissa dataa ei lähetetä URL-osoitteeseen lisättynä, vaan HTTP-otsikoiden jälkeen pyynnön sisältönä. XHR-objekti tarjoaa myös mahdollisuuden synkroniseen HTTP-pyyntöön, mutta silloin muun ohjelman suoritus keskeytyy pyynnön ajaksi ja menetetään mahdollisuus päivittää käyttöliittymää pyynnön aikana.

5.2 JavaScript

JavaScript on pääasiassa web-ympäristössä käytetty skripti- eli komentosarjakieli. Komentosarjakieli on ohjelmointikieli, jota käytetään muokkaamaan, erikoistamaan ja automatisoimaan jo olemassaolevaa järjestelmää. JavaScriptin tapauksessa, järjestelmän tarjoaa yleensä web-selain. Muita alustoja ovat mm OpenOffice, joka käyttää JavaScrptiä yhtenä skriptikielenään, sekä Adoben Systemsin Acrobat-julkaisuohjelmisto.

JavaScriptin kehitti Brendan Eich työskennellessään Netscapella ja se esiteltiin ensi kerran yhtiön Navigator 2.0 selaimen yhteydessä maaliskuussa 1996. JavaScriptin saaman suosion seurauksena Microsoft kehitti JavaScriptistä yhteensopivan murteen, nimesi sen JScriptiksi välttääkseen tavaramerkkiongelmia ja julkisti sille tuen Internet Explorer-selaimen versiossa 3.0 elokuussa 1996. Tässä vaiheessa uudella kielellä ei ollut vielä minkäänlaista standardia, joten Netscape pyysi Ecma International-organisaatiota luomaan kielelle standardin. Standardille annettiin nimeksi ECMA-262 ja sen määrittely aloitettiin loppuvuodesta 1996. Ensimmäinen versio ECMA-262:sta julkistettiin kesäkuussa 1997. ECMAScript on ECMA-262-standardin määrittelemän kielen nimi ja JavaScript on yksi ECMAScriptin toteutuksista sekä samalla sen murre, koska se toteuttaa joitakin ECMA-262-standardissa määrittelemättömiä ominaisuuksia. Muita tunnettuja toteutuksia ovat muiden muassa jo mainittu JScript sekä Adobe Systemsin ActionScript, jota käytetään yhtiön Flash-kehitysympäristössä.

Selaimista ainoastaan Opera toteuttaa ECMAScriptin, Microsoft kutsuu toteutustaan JScriptiksi ja muut selainvalmistajat JavaScriptiksi. AJAX-teknologian kannalta edellämainitut ECMAScript-toteutukset ovat korvaamattomia, sillä ne kaikki toteuttavat XMLHttpRequest-rajapinnan. Tosin Internet Explorerin versioissa 5-6 XMLHttpRequest-rajapintaa vastaa ActiveXObject-olio.

JavaScriptillä oli pitkään hieman kyseenalainen maine ohjelmointikielenä, eikä se johtunut vähiten siitä, että sillä tuotettiin web-sivuille toimintoja, jotka lähinnä häiritsivät käyttäjää sen sijaan, että olisivat parantaneet käytettävyyttä. Lisäksi selaintuki oli puutteellista ja JavaScript-sovellusten debuggaaminen hankalaa, joten monet kehittäjät tyytyivätkin käyttämään sitä lähinnä lomakkeiden validointiin.

Tilanne alkoi hiljalleen muuttua kun XMLHttpRequest- rajapinta alkoi tulla tunnetuksi viime vuosikymmen puolivälissä, sillä se toi aivan uuden ulottuvuuden JavaScriptin hyödyntämiseen ja verkkopalvelujen toteuttamiseen. Nykyisin on olemassa lukuisia web-sivustoja, jotka hyödyntävät JavaScriptä ja XMLHttpRequest- rajanpintaan perustuvaa AJAX-tekniikkaa erittäin tehokkaasti, esimerkiksi Google Apps.

JavaScriptissä ei ole luokkia samalla tavalla kuin ne ymmärretään esimerkiksi C++:n tai Javan yhteydessä. Luokkapohjaisissa oliokielissä tilaa säilytetään luokan instansseissa eli olioissa, metodeita säilytetään luokissa ja perinnässä on kysymys rakenteesta ja käyttäytymisestä. JavaScriptissä tilaa ja metodeita säilytetään olioissa. Tila, metodit, käyttäytyminen ja rakenne kaikki peritään. (Standard Ecma-262 2011, 3.)

JavaScriptissä jokainen olio viittaa johonkin toiseen olioon. Tästä toisesta oliosta käytetään nimitystä prototyyppi ja ensimmäinen olio perii ominaisuutensa prototyyppi-oliolta. Lause *var a = new Array()* luo uuden taulukko-olion, jonka prototyyppi on *Array.prototype*. Olio *a* on siis perityt kaikki *Array.prototype*-olion ominaisuudet ja se voi myös lisätä uusia ominaisuuksia perimäänsä prototyyppiin. JavaScriptissä on kuitenkin yksi olio, joka ei peri prototyyppiä miltään toiselta oliolta ja se on *Object*-olio. *Object*-olio on JavaScriptin oliohierarkiassa korkeimmalla. Kaikki sisäänrakennetut oliot sekä käyttäjän määrittelemät oliot perivät lopulta *Object*-olion prototyypin.

Olion prototyyppi on ominaisuus, joka on tarkoitettu toisten olioiden perittäväksi ja ylikirjoitettavaksi. Edellä esitetty Array-olio on yksi JavaScriptin sisäänrakennetuista oliosta, mutta käyttäjä voi tietenkin määritellä olion myös itse. Olion määritteleminen voi tapahtua niin sanotun literaalin avulla. Lause *var b = {foo:"bar"}* luo literaalin avulla olion b, jolla on jäsenmuuttana merkkijono foo. Literaalit ovat monissa tilanteissa suositeltavia, mutta niissä on yksi puute: literaalin avulla luotu olio ei saa prototyyppi-oliota, jonka toiset oliot voisivat periä. Jotta oliolla olisi prototyyppi, täytyy olion luonnissa sen sijaan käyttää avainsanaa *new* ja *Function*-oliota, joka toimii rakentimena. Luodaan nyt olio rakentimen avulla: *function B(){}; var b = new B();*
Kaikki *new*-avainsanan ja rakentimen avulla luodut oliot saavat prototyyppi-olion, jonka nimi viittaa rakentimen nimeen, joka tässä tapauksessa on *B.* Olion b:n prototyyppiin viitataan lauseella *b.prototype*. Voidaan edelleen luoda edellä kuvatulla tavalla olio *c,* joka voi periä *b:*n prototyypin. Toisen olion prototyyppi täytyy periä erillisellä käskyllä: *C.prototype = new B()*. Käytännössä olion *b* prototyyppi-olio kaikkine jäsenmuuttujineen kopioidaan ja se asetetaan *c*:n prototyypiksi.

Koska itsemääriteltyjen olioiden rakentimena käytetään Function-oliota ja koska Function-olio, niinkuin muunkintyyppiset oliot, voi sisältää mielivaltaisen määrän erityyppi-

siä jäsenmuuttuja, voi toisen olion rakentimen ominaisuuksien hyödyntäminen olla joskus tarpeellista. Toisen olion rakentimen jäsenmuuttujat voidaan ottaa käyttöön kutsumall rakennin-funktiota sen *call*- tai *apply*-metodin avulla. *Call*-tai *apply*-metodien käyttäminen eroaa tavallisesta metodikutsusta, siten että ensimmäinen parametri edellämainituille metodeille on aina viite kutsujaan eli avainsana *this.*

```
function B(name) {
   this.name = name || ”B”
}
function C() {
   B.call(this, ”C”);
}
var c = new C();
c.name // ”C”
```

Edelläolevassa esimerkissä ”lainataan” rakenninta B (Stefanov 2010, 120). B:n *this*-viittaus ei viittaa B:n itseensä, vaan kutsujaan.

Apply-metodi eroaa call-metodista siten, että parametrin this, jälkeen parametriksi annetaan taulukko. Olion rakennin-funktiossa voidaan kutsua rajattomasti toisten olioiden rakentimia, jolloin toteutetaan eräänlainen moniperintä.

5.3 Muut tekniikat

5.3.1 DOM

DOM on lyhenne sanoista Document Object Model. DOM on alusta- ja kieliriippumaton rajapinta, joka sallii ohjelmien ja skriptien dynaamisesti muuttaa dokumentin sisältöä, rakennetta ja tyyliä.

DOM- määrittelee dokumentin puumaiseksi rakenteeksi, jossa jokainen elementti on myös solmu (*node*), joka voi sisältää toisia elementtejä eli lapsisolmuja (*child node*). Koko dokumentti itsessään on solmu, sillä erotuksella muista solmuista, ettei sillä ole

elementtiä johon se itse kuuluisi (*parent node*). HTML-dokumentin tapauksessa ylimmäinen elementti (*root node*) DOM-puussa on <html/>.

DOM myös määrittelee metodit, joilla dokumentin rakennetta ja sisältöä muokataan ja joilla dokumentti esitetään. DOM-rajapinta on erittäin laajasti tuettu ohjelmointikielissä.

DOM jaetaan kolmeen osaan/tasoon:

- Core DOM on standardi mille tahansa rakenteelliselle dokumentille.
- XML DOM on standardi XML-dokumenteille.
- HTML DOM on standardi HTML-dokumenteille.

AJAX-teknologiassa DOM on olennainen tekniikka, sillä web-sovelluksen käyttöliittymän päivitys eli HTML-dokumentin muokkaus tapahtuu lähes yksinomaan DOM-metodien avulla.

5.3.2 JSON

JSON on kevyt ja yksinkertainen tiedonsiirtoformaatti. JSON on lyhenne sanoista JavaScript Object Notation. JSON on JavaScriptin osajoukko ja se standardisoitiin ECMA:n kolmannessa versiossa 1999. JavaScriptin yhteydessä JSON:in käytän tekee erityisen houkuttelevaksi JavaScriptin JSON.parse- metodi, joka muuntaa JSON-notaation JavScript-olioksi.

JSON-objekti on olla joko:
- Kokoelma avain-arvo-pareja. Useimmista kielissä tällaisesta rakenteesta käytetään nimistystä olio, tietue, hash-taulukko, avainlista, tai assosiatiivinen taulukko.
- Arvoista koostuva lista. Useimmissa kielissä tämä vastaa taulukkoa, listaa tai vektoria.

Avain on JSON-notaatiossa tyypiltään merkkijono. Arvo toi olla tyypiltään merkkijono, numero, olio, taulukko, true, false tai null (Introducing JSON).

Tyypillinen JSON-tietorakenne voisi näyttää esimerkiksi tältä:

```
{ "foo": "bar", "baz": 10, "foobar": [12, 14, 16] }
```

JSON on vaihtoehto XML:lle monissa tapauksissa: sen syntaksi on huomattavasti yksinkertaisempi ja se nivoutuu usein suoraan kielen tietorakenteisiin. Lisäksi vaadittu tiedonsiirtomäärä on huomattavasti pienempi kuin XML:n kohdalla. Toisaalta JSON ei ole laajennettava, eli siihen ei voi määritellä uusia tageja tai attribuuteja kuvaamaan sen sisältämää dataa.

5.3.3 PHP

AJAX-sovellusten palvelinsovelmat voidaan toteuttaa periaatteessa millä tahansa ohjelmointikielellä, jota www-palvelin tukee ja josta löytyy tuki halutulle tietokannalle. Syitä valita PHP palvelinsovelmien toteutuskieleksi voisivat olla esimerkiksi seuraavat:

– PHP:n kehitys- ja tuotantoympäristön käyttöönotto on helppoa ja PHP-tuki on mahdollista asentaa lähes kaikkiin markkinoilla oleviin palvelimiin.

– PHP:n tietokantarajapinnat ovat helppokäyttöisiä ja se tuettujen tietokantojen määrä on vähintään kattava.

– PHP tukee olio-ohjelmointia, joten laajempienkin sovellusten toteutus on mahdollista. Toisaalta PHP:n syntaksi on selkeää ja sitä voidaan käyttää myös täysin proseduraalisesti. PHP toteuttaa monessa suhteessa Perl:stä tuttua lähestymistapaa, jonka mukaan helppojen asioiden tulee olla helppoja ja vaikeiden mahdollisia.

– PHP toteuttaa XML DOM-rajapinnan ja tarjoaa natiiviin tuen JSON-formaatin käsittelyyn.

– Tulkattavuudestaan huolimatta PHP:n suorituskyky on varsin hyvä. Lisäksi on olemassa lukuisia laajennuksia, joilla PHP:n tavukoodi voidaan tallentaa palvelimen välimuistiin tai jaettuun muistiin, jolloin tavukoodikäännöstä ei tarvitse tehdä jokaisen pyynnön yhteydessä uudelleen. Tästä seuraa huomattava suorituskyvyn paraneminen. Eräs tällainen laajennos on Alternative PHP Cache (APC). APC on PHP:n sisarprojektin PECL.n ylläpitämä laajennus ja sen käyttöönotto melko helppoa.

6 ESIMERKKISOVELLUS

6.1 Johdanto

Esimerkkisovelluksesi on valittu verkkokauppa, jonka myyntiartikkeleina toimivat vuosikertaviinit. Tavanomainen verkkokauppa on siinä mielessä kelvollinen esimerkkisovellukseksi, sillä sen perustoimintojen logiikka on lähes kaikille tuttua: käyttäjä selailee tuotteita, lisää niitä ostoskoriin, mahdollisesti muokkaa tilausta ja lopuksi tekee tilauksen tai peruu sen. MVC arkkitehtuurin käyttö verkkokauppasovelluksessa tuntuu ylimitoitetulta ja on myös sitä, eikä esimerkkien tarkoitus olekaan esittää parasta tai oikeaa tapaa toteuttaa verkkokauppasovellus. Esimerkkisovelluksen ja sen JsDOC-dokumentaation saatavuus on mainittu liittessä 1.

Verkkokauppasovelluksissa on tavallista, että ostoskorin datasta muodostetaan useita näkymiä. Verkkokaupan sivulla näytetään poikkeuksetta jonkinlainen yksinkertaistettu, pieni ostoskorinäkymä, jonka tarkoitus on lähinnä informoida käyttäjälle koriin lisättyjen artikkeleiden määrä sekä kokonaissumma. Verkkokauppasovelluksessa on myös tavallisesti erillinen näkymä ostoskorin muokkausta varten, tosin monesti tämä muokkausnäkymä on liitetty tilausvaiheeseen.

Kriteeri MVC-arkkitehtuurin käytölle tulee siis kutakuinkin täytetyksi, koska useimmissa verkkokaupoissa on jossain vaiheessa tilausprosessia kaksi samanaikaista ja synkronoitua näkymää ostoskorin sisältämästä tiedosta. Lisäksi MVC:n ominaisuus vaihtaa näkymän ohjainta ajonaikaisesti tarjoaa helposti ylläpidettävän ja uudelleenkäytettävän tavan muokata tilausprosessin toimintaa. Tätä ominaisuutta hyödynnetään esimerkkisovelluksessa useassa tilanteessa, joista mainittakoon tilanne jossa käyttäjää pyydetään tilausvaiheessa tunnistautumaan. On varsin hyödyllistä, että tunnistautumaton käyttäjä ei voi viedä tilausta loppuun, joten vaihdettavat ohjaimet tarjoavat tavan manipuloida käyttöliittymää tilausprosessin eri vaiheissa.

Tänä päivänä useimmat verkkokaupat yhdistävät tilauksen eri vaiheet yhdelle sivulle, koska yksisivuinen tilausprosessi käyttäjätutkimusten mukaan johtaa todennäköisimmin loppuun asti tehtyyn tilaukseen. Esimerkkisovelluksen tilaussivulla hyödynnetään Re-

kursiokooste-suunnittelumallia muodostamalla tilausprosessin eri vaiheista *komposiittinäkymä,* joka voidaan esimerkiksi avata tai sulkea yhdellä komennolla riippumatta sen sisältämien lapsinäkymien määrästä ja niiden keskinäisistä suhteista.

Verkkokauppaohjelmistojen hallintapaneelit sisältävät usein jonkinlaisia myynninedistämistyökaluja. Tällaisilla työkaluilla seurataan erilaisia myynnin tunnuslukuja kuten myynti- tai käyttökatetta graafisten esitysten, kuten erityyppisten diagrammien avulla. MVC:n käyttöä on helppo perustella luotaessa graafisia esityksiä AJAX:in avulla: palvelimelta vastaanotetusta datasta voidaan asiakassovelluksen näkymä-komponentissa muodostaa tarpeen mukaan erilaisia, samanaikaisia ja synkronisia näkymiä. Tämä päivänä on olemassa lukuisia JavaScript-kirjastoja erilaisten raporttien tuottamiseen, esimerkiksi Dojo Toolkitin Charting-moduuli. Tällaisten JavaScript-kirjastojen avulla voidaan visuaalisen presentaation kehitystyö siirtää kokonaan selaimeen ja palvelinsovelluksen tehtäväksi voidaan jättää puhtaasti datan kerääminen.

Juuri erilaisia raportteja tuottavissa web-palveluissa, esimerkiksi business intelligence-sovelluksissa, on MVC:n toteuttaminen asiakassovellukseen kannattavaa, koska tuotettavien raporttien tyyppi ja määrä voi muuttua ohjelman elinkaaren aikana ja on tavallista, että samaa tietolähdettä tarkastellaan ja analysoidaan yhtäaikaisesti erityyppisten raporttien avulla. Usein raportteja halutaan lisäksi muunnella antamalla tietolähteelle eli mallille parametreja, esim.demonstraatiotarkoituksessa. Komposiittinäkymän sisäinen tai erikseen määritelty vastuuketju tarjoaa mahdollisuuden luoda erittäin monipuolista ja muokattavaa toiminnallisuutta sovelluksen käyttöliittymään.

6.2 Malli

Sovelluksen malliluokkien keskeinen tehtävä noutaa data palvelimelta, säilyttää sitä sovelluksen elinkaaren ajan sekä tallentaa muutokset takaisin tietokantaan. Palvelinmallin datan lisäksi malliluokilla voi olla laskennallisia tai muulla tavalla muodostettuja muuttujia. Esimerkiksi kun käyttäjä muutta artikkelin kappalemäärää ostoskorissa, *Cart*-malliluokka voi laskea reaaliaikaisesti uutta hintaa artikkelille sekä uutta kokonaissummaa tilaukselle, jolloin käyttäjä näkee mitä hänen potentiaalinen tilauksensa tulisi maksa-

maan. Käyttäjän etu- ja sukunimestä voidaan muodostaa koko nimi. Omaa dataa malleilla ei kuitenkaan ole.

Malliluokat ottavat vastaan käyttäjän syötteitä ja prosessoivat niitä. Esimerkkisovelluksen tapauksessa kysymys ei ole muusta kuin käyttäjän syötteiden validoinnista, esim. artikkelin kappamäärän on oltava nollaa suurempi tai käyttäjän sähköpostiosoite on annettava. Käyttäjän syötteen validointi voitaisiin tietenkin toteuttaa esimerkiksi implementoimalla lomakkeen onSubmit-metodi, mutta tässä tapauksessa validointi toimikoon esimerkkinä mallin toiminnasta. Mikäli jokin mallin jäsenmuuttuja on jollain tavalla kelvoton, se tiedottaa tilanteesta kaikille rekisteröityneille tarkkailijoille kutsumalla nofity-metodiaan, joka saa parametrikseen merkkinojonon 'invalid'. Kun kontrolleri vastaanottaa *update*-metodissaan tämän viestin, se voi esimerkiksi deaktivoida Tallenna-painikkeen, ja pyytää näkymäänsä esittämään jonkinlaisen virheilmoituksen käyttäjälle.

Yksinkertaisimmassa tapauksessa malli kutsuu tarkkailijoiden *update*-metodia ja tarkkailija päivittää itsensä. Näin suoraviivainen toimintatapa saattaa kuitenkin olla tehoton ja onkin useita keinoja estää tarkkailijoita tekemästä turhaa työtä. Yksi tapa on, kuten edellisessä kappaleessa jo mainittiin, välittää tarkkailijoiden update-proseduurille jonkinlainen tilan muutosta kuvaava muuttuja parametrina, jolloin tarkkailija voi helpommin päätellä, tarvitseeko sen reagoida muutokseen mitenkään. (Buschmann, Meunier, Rohnert, Sommerlad & Stal 1996, 134.)

Esimerkkisovelluksen *Cart*- ja *User*- malliluolilla on kolme yhteistä tilaa kuvaavaa muuttujaa: 'restored', 'saved', 'modified' ja 'invalid'. Sovelluksen mallit käyttävät lähestymistapaa, jossa tila 'restored' tarkoittaa, sitä että data noudettu tietokannasta, eikä se ole sen jälkeen muuttunut mitenkään. Mallin alustuksen yhteydessä tarkkailijoille lähetetään viesti 'restored', jolloin esimerkiksi näkymät päivittävät itsensä kokonaan. Tila 'saved' viestittää tarkkailijoille tiedon siitä, että jokin mallin muokattu ominaisuus on tallennettu palvelimelle ja vastauksena on saatu tallennettu muutos, esimerkiksi ostoskorin muokattu artikkeli. Tällöin ostoskori-näkymä päivittää ainoastaan *sen artikkelin*, jota on muokattu. Kuten edellä jo mainittiin tila 'invalid' tarkoittaa jollain tavalla ei toivottua tilaa. Tarkempia johtopäätöksiä tilanteesta tarkkailijat voivat tehdä noutamalla dataa mal-

lilta. Tallentamattomista muutoksista malli tiedottaa viestillä 'modified', jolloin ohjain voi muokata näkymää, esimerkiksi aktivoida Tallenna-painikkeen.

Lisäksi *User*-malliluokalla on kaksi omaa tilaa: 'authorized' ja 'unauthorized', joiden tarkoitus on ilmaista onko käyttäjä kirjautunut sisään vai ei. Tilastojen malliluokka ei puolestaan tarvitse tilaa kuvaavia muuttujia; näkymät yksinkertaisesti päivitetään uudelleen, kun jokin tilaston parametri muuttuu.

Esimerkkisovelluksen kaikilla malleilla on yhteinen yliluokka *Model,* johon on kirjoitettu joitakin malliluokille yhteisiä toimintoja, kuten HTTP-kutsun suorittaminen (*Model.request*), tapahtumamekanismi eli *notify*-metodi sekä metodit *attach*, joka liittää tarkkailijan mallille ja *detach*, joka irrottaa sen mallista.

```
Model.prototype = {
    notify : function (state) { // Kutsu tarkkailijoiden update-metodia.
        for (var i = 0, max = this._registry.length; i < max; i++) {
            this._registry[i].update(state);
        }
    },
    attach : function (o) { // Lisää tarkkailija mallille.
        if (o instanceof Observer) {
            this._registry.push(o);
        }
    },
    detach : function (o) { // Irroita tarkkailija mallista.
        for (var i = 0, max = this._registry.length; i < max; i++) {
            if (o === this._registry[i]) {
                return this._registry.splice(i, 1)[0];
            }
        }
        return o;
    },
};
```

Konkreettiset malliluokat kuten, *Cart* ja *User* perivät nämä metodit *Model*-luokan prototyypin kautta.

```
function Cart () {
    // Jos instanssi on olemassa, palauta se.
    if (Cart.instance instanceof Cart) {
        return Cart.instance;
    }
    Cart.instance = this; // Luodaan uusi instanssi.
}
Cart.prototype = new Model();
```

Konkreettiset malliluokat palauttavat aina saman instanssin itsestään (*Singleton*). Mallin ja tarkkailijoiden suhde on yksi moneen, joten on hyödyllistä varmistaa, että tarkkailijoilla on aina sama instanssi mallistansa. Tästä menettelystä on myös toinen hyöty: missä tahansa kohtaa sovellusta voidaan huoletta kutsua mallin rakenninta, ja asettaa saatu instanssi paikalliseen muuttujaan.
Konkreettiset malliluokat toteuttavat metodin *setData,* joka nimensä mukaisesti asettaa palvelimelta vastaanotetun datan omiin jäsenmuuttujiinsa. Malli kutsuu *setData*-metodia HTTP-pyynnön valmistuttua.

```
Model.prototype.request = function (url, ...) {
    var self = this;
    return $.ajax({
        ...
        // jQuery.ajax() kutsuu success-metodia, kun pyyntö on valmis.
        success : function (data, textStatus, jqXHR) {
            self.setData(data); // Aseta vastaanotettu data.
            self.notify('restored'); // Tapahtumamekanismi
        }
     }
    ...
}
```

Asynkronisuus on AJAX-tekniikan keskeisin ominaisuus. Käyttäjää on usein hyödyllistä informoida ,että jotain on juuri tapahtumassa tai jokin prosessi on juuri päättynyt. MVC-arkkitehtuurille luonteenomaista on useiden samanaikaisten näkymien muodostaminen samasta tietolähteestä, joten *kaikille* tarkkailijoille täytyy välittää viesti prosessin keskeneräisyydestä.

Keskeneräisen HTTP-kutsun tai virhetilanteen voidaan ajatella olevan normaalista poikkeava tilanne, joka ei tule jatkumaan loputtomasti. Näkymien kohdalla puhumme siksi tästä eteenpäin poikkeustilanteesta ja myös koodiesimerkeissä sana *exception* viittaan keskeneräiseen HTTP-kutsuun tai virhetilanteeseen.

Mallin tarkkailijoille voitaisiin tietenkin tiedottaa poikkeustilanteesta *notify*-metodilla, jolle annettaisiin parametriksi jokin poikkeustilaa kuvaava viesti, kuten *XMLHttpRequest.readyState*-arvo tai palvelimen virheilmoitus, mutta vielä suoraviivaisempi tapa on toteuttaa malliin ja tarkkailijoihin omat metodit poikkeustilanteita varten.

Kun *XMLHttpRequest.readyState* arvo on pienempi kuin 4, *Model*- kutsuu *notify*-metodinsa sijaan *notifyProgress*-metodiaan, joka puolestaan kutsuu tarkkailijoiden *progress*-metodia.

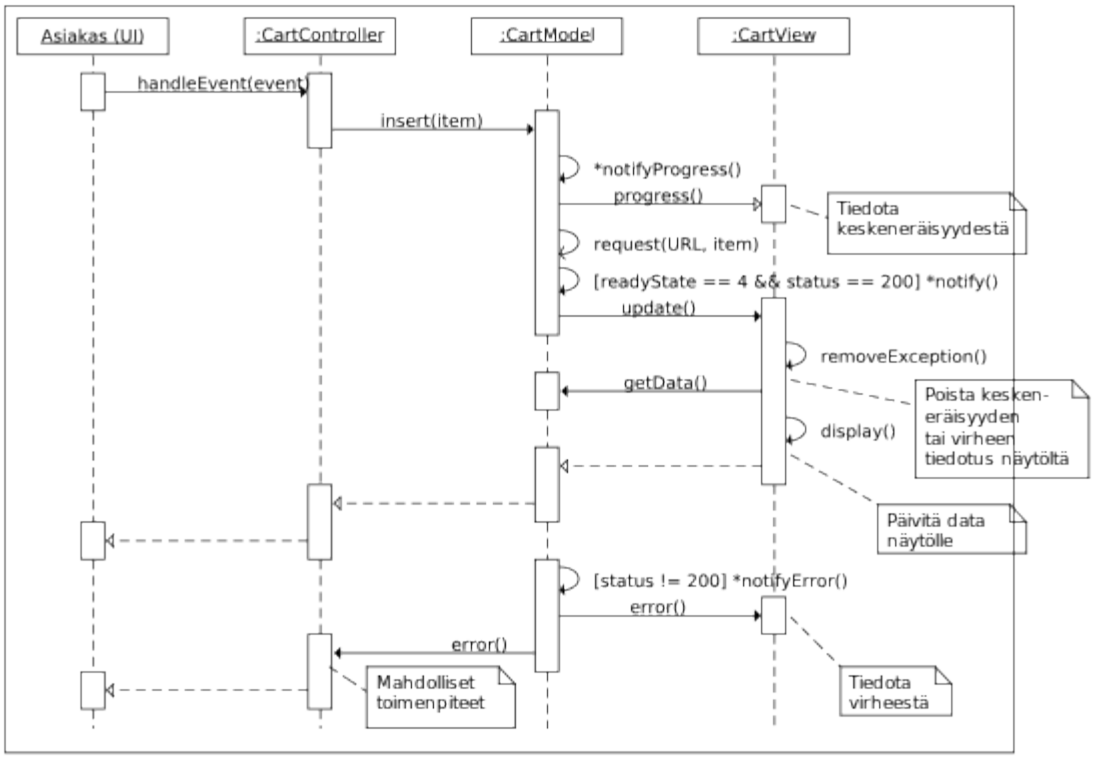
```
Model.prototype = {
    notifyProgress : function (jqXHR) {
        for (var i = 0, max = this._registry.length; i < max; i++) {
            // Kutsu tarkailijan progress-metodia, jos se on näkymä.
            if (this._registry[i] instanceof View) {
                this._registry[i].progress(jqXHR);
            }
        }
    } ...
}
```

Virhetilanteen kohdalla *Model* kutsuu *notifyError*-metodia:

```
Model.prototype = {
    notifyError : function (jqXHR, textStatus, errorThrown) {
        // Kutsu tarkkailijan error-metodia.
        this._registry[i].error(jqXHR, textStatus, errorThrown);
    }  ...
}
```

Kun poikkeustilanteiden käsittely on eristetty omiin metodeihinsa sekä mallissa että tarkkailijoissa, varsinaisessa näytönpäivitysproseduurissa ei tarvitse selvittää voidaanko

näkymä päivittää normaalisti vai onko kysymys keskeneräisestä HTTP-kutsusta tai palvelinvirheestä.



KUVA 6. Keskeneräisyyden tai virhetilanteen (*exception*) tapahtumamekanismi.

Web-asiakkaaseen toteutetussa MVC-arkkitehtuurissa mallin tehtävä on pääasiassa palvelimelta noudetun tiedon säilyttäminen HTTP-kutsujen välillä ja tarkkailijoiden informoiminen aina kun tila jotenkin muuttuu. Käyttäjän syötteiden validointi voidaan antaa mallin tehtäväksi, niinkuin esimerkkisovelluksessa on tehty, jotta tapahtumamekanismi saadaan demonstroitua. Malli voi myös heittää poikkeuksen, jos jokin käyttäjän syöte on vääräntyyppinen. Poikkeus otetaan kiinni näkymässä. Tällöin vältytään *notify*-kutsulta ja mallin sekä toimintaa saadaan yksinkertaistettua.

Syötteiden validointi voidaan myös yhtä hyvin toteuttaa tavalliseen tapaan lomakkeen onSubmit-tapahtumaan, jos validointiin ei liity sen kummempaa logiikka kuin esimerkiksi onko arvo tyypiltään oikea.

Malliolion olemassaololle on kuitenkin olemassa perusteita. Sen data on helpommin saatavissa kuin vaikkapa lomakkeessa säilytettävä data: ei tarvita DOM-metodeita. Malliin voidaan kirjoittaa loputtomasti logiikkaa, jos sellainen katsotaan tarpeelliseksi ja turvalliseksi. Sovelluksen laajentamisen ja muokkaamisen mahdollisuudet ovat huomattavasti paremmat jos sovelluksen tieto ja liiketoimintalogiikka eristetään tiukasti tiedon esittämisestä.

6.3 Näkymä

Web-asiakassovelluksessa ja kuten myös muunkinlaisissa sovelluksissa MVC-triadin alustus alkaa näkymäluokan alustamisella. Tässä prosessissa näkymälle annetaan aina parametrina sen malliluokka. Edelleen alustuksen yhteydessä näkymä luo ohjaimensa, mikäli sellaista tarvitaan. MVC-arkkitehtuuria hyödyntävä web-palvelu ei välttämättä ole ns. single-page-sovellus, vaan se saattaa koostua useista erillisitä sivuista, joista jokainen saattaa alustaa heti sivun latauksen yhteydessä näkymäluokkia. Tällöin palvelimelta palautetaan JavaScript-koodi, joka alustaa halutut näkymät. Tämä koodi sijoitetaan tavallisesti head-elementtiin ja sen sisältö voi näyttää esimerkiksi seuraavalta:

```
<head>
<script type=”text/javascript”>
    $(document).ready(function () {
        // Alusta näkymä ja näytä se.
        var model = new Model(),
            view = new View(model);
        view.initialize().display();
    });
</script>
</head>
```

Erittäin usein web-palvelun ulkoasu on rakennettu siten, että sen tuottama sisältö hyödyntää yhtä perusrakennetta eli *layoutia*. Palvelun palauttaman sisältö siis vaihtelee, riippuen tehdyistä HTTP-pyynnöistä, mutta sivun perusrakenne pysyy samana. Tällöin *aina* luotavien näkymien lisäksi on pystyttävä päättelemään mitä muita näkymiä sovel-

luksen tulee mahdollisesti alustaa. Tämä voidaan tehdä esimerkiksi tutkimalla URL-osoitteen sisältöä. Edellä olevaa koodiesimerkki on täydennettävä seuraavasti:

```
$(document).ready(function () {
    // Edellisen esimerkin koodi eli aina luotavat näkymät.
    ...
    // Etsi query stringistä avaimen page sisältö.
    // URL: someservice?page=catalog.
    var page = window.location.search.match(/\?page=(\w+)/)[1];

    if ('catalog' == page) {
        // Alusta ja näytä catalog-näkymä.
        var catalog = new Catalog(),
            catalogView = new catalogView(catalog);
        catalogView.initialize().display();
    }
});
```

Tällaisissa sivun alustuksen yhteydessä luotavissa näkymissä ei mallin *välttämättä* tarvitse alustuksen yhteydessä noutaa dataa palvelimelta, koska HTTP-pyynnön yhteydessä voidaan näkymän käyttöliittymäkomponenttien lisäksi palauttaa myös mallin data. Tällöin palautettavan datan mime-tyyppi on tietenkin text/html. Esimerkkisovelluksen malliluokat on toteutettu siten, että niiden rakentimille voidaan antaa parametri, joka määrittää, ettei mallin alustuksen yhteydessä tarvitse tehdä yhtään mitään, koska malliluokan tieto on jo valmiiksi sisällytetty näkymään.

Kun sovelluksen eli HTML-sivun DOM-rakenne on valmis, ei näkymiä enää voida luoda edellä kuvatulla tavalla. Sen sijaan näkymät luodaan dynaamisesti eli sivun DOM-rakennetta muokataan ajonaikaisesti DOM-metodien avulla. Tämä prosessi käynnistyy käyttäjän tekemästä toimenpiteestä, esim. painikkeen tai linkin painalluksesta. Esimerkkisovelluksessa kaikki näkymät luodaan dynaamisesti ja näitä prosesseja on kuvataan myöhemmistä kappaleissa.

Kun esimerkkisovellus ladataan, alustetaan ja näytetään aina ostoskori, sillä ostoskorin tulee olla jatkuvasti näkyvillä. Ostoskorin malli tosin noutaa datansa JSON-formaatissa

palvelimelta alustauksen yhteydessä, mutta muuten menettely on sama kuin ensimmäisessä esimerkissä on kuvattu.

Luotaessa uutta näkymää, otetaan lähtökohdaksi, että mallin tilaa ei haluta muuttaa, vaan mallilta on tarkoitus ainoastaan pyytää data näkymän luomista ja näyttämistä varten. Mallin ei siis tarvitse laukaista tapahtumamekanismia, koska mallin tila pysyy muuttumattomana, eikä muita mahdollisia näkymiä tarvitse päivittää. XMLHttpRequest-rajapinta hyväksikäyttää takaisinkutsu-periaatteetta ja samaa periaatteetta voidaan hyödyntää myös MVC:n tapauksessa. Takaisinkutsu on tapa toteuttaa ohitustilanne kerrosarkkitehtuurissa. Takaisinkutsussa alempi kerros tarjoaa mekanismin, jolla ylempi kerros antaa koodinsa alemman kerroksen käyttöön. Mitä toiminallisuutta ylemmän kerroksen koodi sisältää, selviää vasta ohjelman suorituksen aikana. Takaisinkutsu ei tee alempaa kerrosta riippuvaiseksi ylemmästä, koska alempi kerros (malli) ei edelleenkään tunne kutsujaa. *Model.request*-metodin takaisinkutsumekanismi näyttää seuraavalta:

```
Model.prototype.request = function (url, data, opts) {
    var self = this;
    return $.ajax({
    ...
    // Jos opts.success.fn on funktio, kutsu sitä.
    // Muutoin laukaise tapahtumamekanismi.
    success : function(data, textStatus, jqXHR) {
        (opts.success && "function" == typeof opts.success.fn)
          // Takaisinkutsu. Tapahtumamekanismia ei laukaista.
          ? opts.success.fn.apply(opts.self, opts.success.args)
          : self.notify('restored'); // Tapahtumekanismin kutsu.
    }
  }
}
```

Uutta näkymää luotaessa voidaan hyödyntää takaisinkutsua, myös jos halutaan että näkymä näytetään käyttäjälle, vasta kun HTTP-pyyntö on valmistunut kokonaan. Jos HTTP-kutsu kestää pitkään, eikä syystä tai toisesta haluta ilmaista käyttäjälle, että HTTP-pyyntö on kesken, takaisinkutsulla estetään käyttäjää näkemästä esim. tyhjiä lomakkeita. Takaisinkutsuna välitetään mallille tässä tapauksessa se metodi joka, luo käyttöliittymän HTML-komponentit ja asettaa niiden datan. Malli kutsuu tätä metodia, eikä siis kutsu tarkkailijoiden *update*-proseduuria.

Näkymä-luokkien tehtävä on tietenkin myös poikkeustilanteiden esittäminen käyttäjälle. Poikkeustilanteen esittäminen on hyvin suoraviivaista: usein prosessin keskeneräisyydestä riittää kertomaan jonkinlainen latausanimaatio, virhetilanteessa taas näytetään virheilmoitus, jos niin halutaan. Jotta vältyttäisiin turhalta työltä kannatta näkymille toteuttaa jonkinlainen oletusproseduuri poikkeustilanteita varten.

Näkymien yhteiseen kantaluokkaan *View* on toteutettu metodi *progress*, joka piilottaa varsinaisen näkymän ja laittaa sen tilalle keskeneräisestä tapahtumasta ilmoittavan näkymän. Konkreettisten luokkien toteutukset voivat vaihdella, mutta yksinkertaisin oletusarvoinen toteutus on piilottaa näkymän DOM-juurielementti ja laittaa latausnäkymän juurielementti välittömästi sen perään DOM-rakenteessa.

```
View.prototype.progress = function () {
    this._exceptionContainer = $(”<div/>”)
            .append(”<img/>”, ”src”:”loader.gif”);
    this._container.hide();
    this._exceptionContainer.insertAfter(this._container);
};
```

Ylläoleva metodi voidaan ylikirjoittaa konkreettisissa aliluokissa, jos halutaan esittää käyttäjälle tarkempaa tietoa meneillä olevasta HTTP-kutsusta. Virhetilanteen kohdalla *Model* kutsuu *notifyError*-metodia:

```
Model.prototype = {
    notifyError : function (jqXHR, textStatus, errorThrown) { ... }
}
```

View-kantaluokkaluokkaan on toteutettu *error*-metodi, joka ilmoittaa virhetilanteesta. Metodi toimii samaan tapaan kuin *View.progress*, mutta näyttää käyttäjälle virheilmoituksen:

Konkreettiselle näkymäluokalle voidaan yksinkertaisesti kirjoittaa tyhjä *progress*- tai *error*- metodi, jos halutaan ettei poikkeustilanteesta informoida käyttäjälle.

HTTP-kutsun valmistuttua kutsutaan näkymien *update*-metodia.
Ensimmäiseksi täytyy poistaa mahdollinen lataus- tai virhenäkymä. Sen jälkeen

näytän päivitys voi jatkua normaalisti.

```
SomeView.prototype.update = function () {
    // Poista exception-näkymä, jos olemassa.
    if (this._exceptionContainer.length) {
       this._exceptionContainer.remove(); // Poista elementti lapsineen.
    }
    this._container.show();
};
```

6.4 Ohjain

Ohjain-luokkien tehtävät esimerkkisovelluksessa, kuten MVC-arkkitehtuurissa yleensäkin, on toimia välittää käyttäjän syötteitä malliluokille. Toisinaan jokin muutos mallin tilassa vaatii ohjaimelta toimenpiteitä. Esimerkkisovelluksen malleissa ei ole lainkaan liiketoimintalogiikkaa, joten ohjaimen tarvitsee puuttua vain tilanteisiin, joissa käyttäjä muuttaa mallin tilaa.

```
CartItemController.prototype.update = function (state) {
   // Aktivoi tai passivoi näkymän painikkeet.
   this.view.activateSaveButton('modified' == state);
};
```

Edellä olevassa esimerkissä ostoskorin ohjain on kiinnostunut vain yhdestä asiasta: onko käyttäjä muuttanut mallia ('modified'). Mikäli on, aktivoidaan Tallenna-painike, jotta käyttäjä tallentaisi muutokset, muussa tapauksessa se voidaan pitää passivoituna. Aiemmin jo esitettiin, että ohjain liitetään mallin tarkkailijaksi, mikäli sen toiminta riippuu mallin tilasta. Tämä tarkoittaa sitä, että jos ohjaimen tehtäväksi jäisi ainoastaan ottaa vastaan käyttäjän syötteitä ja kutsua mallin palveluja, ei ohjaimen *update*-proseduurilla ole mitään käyttöä. Web-sovelluksessa, jossa MVC:n komponentit toteutetaan JavaScriptillä, on tällaisessa tilanteessa täysin asianmukaista jättää ohjain-luokka toteuttamatta kokonaan, ja kutsua mallin palveluja suoraan näkymä-luokasta. JavaScript ei ole suorituskyvyltään verrattavissa Javaan tai C++:saan, joten turhia rajapintoja ja ”ylitoteuttamista” on syytä välttää.

6.5 Hierarkkiset näkymät ja ohjaimet

Rekursiokoostetta hyödynnetään etenkin käyttöliittymän toteutuksessa monella tasolla. Web-sovelluksessa alimmalla tasolla on DOM-malli. Valikkorakenteet ovat toinen esimerkki rekursiivisista rakenteista käyttöliittymässä. Rekursiokoosteelle tyypillinen ominaisuus on, että se käyttäytyy asiakkaalle aina samalla tavalla a riippumatta siitä, minkätyyppisiä komponentteja se sisältää tai mikä on komponenttien lukumäärä. Esimerkiksi komento, joka avaa valikon, saa aikaan sen, että valikon jokainen kohde näytetään. Jos valikkoon lisätään ohjelman suorituksen aikana lisää kohteita, myös ne tulevat näkyville samalla komennolla. Kun valikko suljetaan, samalla komennolla suljetaan myös kaikki avoinna ole alavalikot.

Myös MVC-mallin näkymäluokista ja niiden ohjaimista voidaan muodostaa rekursiivinen ja hierarkkinen rakenne, jota kutsutaan *komposiittinäkymäksi* (Gamma, Helm, Johnson & Vlissides 1997, 5). Kun esimerkiksi avoinna olevien dialogien muodostaman hierarkian ylin näkymä suljetaan, suljetaan tavallisesti myös sen lapsinäkymät. Avoinna olevien näkymien ohjaimet voivat olla kiinnostaneita yhtäaikaisesti jostakin tapahtumasta. Esimerkiksi painike dialogi-ikkunassa reagoi hiiren klikkaukseen, mutta ei 'a'-kirjaimen painallukseen näppäimistöllä. Jos isäntädialogi sisältää tekstikentän, 'a'-kirjain lähetetään isäntädialogin ohjaimelle. Tapahtumat jaetaan kaikkien aktiivisten näkymien ohjaimille jonkin määritellyn järjestyksen mukaisesti. Tämän järjestyksen määrittelemiseen käytetään Vastuuketju (*Chain of Resposibility)*-suunnittelumallia (Gamma, Helm, Johnson & Vlissides 1997, 224). Vastuuketju-mallissa joukolle objekteja, joista kukin osaa käsitellä vain tietyntyyppistä palvelupyyntöä, määritellään hierarkia, jonka mukaisesti objektit käsittelevät pyyntöjä. Jos pyynnön ensimmäisenä vastaanottanut objekti ei osaa käsitellä pyyntöä, se välittää sen hierarkiassa seuraavana olevalle objektille eli *seuraajalle* (*successor*). Seuraaja joko käsittelee pyynnön tai välittää sen edelleen eteenpäin.

Komposiittinäkymässä ei välttämättä tarvitse määritellä vastuuketjua eksplisiittisesti. Jos komposiittinäkymän lapsi-isäntä suhde sellaisenaan vastaa haluttua vastuuketjua, näkymän viite sen isäntään voi toimia seuraajana. Muussa tapauksessa vastuuketju täytyy asettaa erikseen. Komposiittinäkymässä pyynnön käsittelijänä toimii tavallisesti nä-

kymän ohjaimen *handleEvent*-metodi. Vastuuketjussa pyynnön tuleminen käsitellyksi ei ole taattu. Mikäli pyyntö välitetään koko vastuuketjun läpi tulematta käsitellyksi, se yleensä päätyy jonkinlaisen oletusmetodin käsiteltäväksi, jonka toiminta riippuu tietysti täysin sovelluksesta. Komposiittinäkymässä tällainen metodi voi olla *Controller*, joka on konkreettisten ohjainluokkien kantaluokka ja jokaisen näkymän oletusohjain, mikäli ohjainta ei ole erikseen määritelty. Tyypillisesti *Controller* ei reagoi mihinkään käyttäjän toimenpiteeseen. Koska komposiittinäkymässä seuraaja on aina joko isäntänäkymä tai erikseen määritelty näkymä, käsittelemätön pyyntö päätyy sovelluksen oletusohjaimelle, ellei toisin ole määritelty.

Komposiittinäkymässä lapsinäkymä lisätään isäntänäkymän lapsinäkymiä säilyttävään listarakenteeseen ja samalla lisätyn lapsinäkymän isännäksi tulee lisäysoperaation suorittanut näkymä. Samalla asetetaan oletusarvoinen vastuuketju, joka vastaa komposiittinäkymän hierarkiaa.

```
View.prototype.addChild = function (view) {
    // Lehtiolioon ei voi lisätä komponenttia.
    if (!this.getComposite()) {
        throw { message : ″Cannot add child to leaf"};
    }
    view._parent = this; // Aseta viite isäntään.
    // Aseta vastuuketjun seuraajaksi isäntänäkymän ohjain.
    view._successor = this._controller;
    this._children.push(view); // Lisää 'view' lapsinäkymäksi.
    return this; // Palauta isäntä.
};

var v1 = new View1(m),
    v2 = new View2(m),
    v3 = new View3(m);

v2.addChild(v3);
v1.addChild(v2);
```

Komposiittinäkymään määritellään metodit eksplisiittisen seuraajan asettamista ja palauttamista varten.

```
// Palauta eksplisiittisesti määritelty tai oletusarvoinen seuraaja.
View.prototype.getSuccessor = function () {
    if (this._successor) { // Palauta seuraaja, jos on.
        return this._successor;
    }
    if (this._parent) { // Palauta isäntänäkymän ohjain.
        return this.getParent().getController();
    }
    // Palauta oletusohjain.
    return new Controller();
};

// Aseta seuraajaksi parametrina annetun näkymän ohjain.
View.prototype.setSuccessor = function (view) {
    if (view instanceof View) {
        this._successor = view.getController();
    } else { // getSuccessor palauttaa isäntänäkymän ohjaimen.
        this._successor = null;
   }
};
```

Jos komposiittinäkymän alustuksessa muodostettua oletusarvoista vastuuketjua halutaan muuttaa, se tapahtuu seuraavasti.

```
V1.setSuccessor(undefined);
V2.setSuccessor(v1);
V3.setSuccessor(v2);
```

Ohjain välittää käsittelemättömän tapahtuman komentoketjussa seuraavalle käsittelijälle, joka komposiittinäkymässä voi olla isäntänäkymän ohjain tai eksplisiittisesti määritelty seuraaja.

```
View1.prototype.handleEvent = function (event) {
    if (81 == event.keyCode) { // kirjain 'q'
        //  Yrittää sulkea dialogit hierarkiassa.
    }  else {
        // Seuraajaa ei ole, pyyntö välitetään oletusohjaimelle.
        this.getSuccessor().handleEvent(event);
    }
};
```

```
View2.prototype.handleEvent = function (event) {
    if ('mouseover' == event.type) {
        // Käsittele tapahtuma.
    } else {
       this.getSuccessor().handleEvent(event);
    }
};

View3.prototype.handleEvent = function (event) {
    if ('click' == event.type) {
        // Käsittele tapahtuma.
    } else {
        this.getSuccessor().handleEvent(event);
    }
};
// Painetaan kirjainta 'q' näkymän v3 ollessa aktiivisena.
```

6.6 Tarkkailija

MVC-arkkitehtuurissa malliluokka on tarkkailun kohde. Näkymä- ja ohjainluokat ovat tarkkailijoita. Tarkkailijat ovat riippuvaisia mallista, mutta malli itse ei millään tavalla tarkkailijoistaan. Mallin ei tule tehdä päätelmiä siitä, minkä tyyppisiä sen tarkkailijat ovat ja mikä on niiden lukumäärää. Tällaisessa tilanteessa sovelletaan Tarkkailija-suunnittelumallia.

Mallien yhteinen kantaluokka *Model* toteuttaa metodin *notify,* jolla tarkkailijoita kutsutaan. Lisäksi *Model*-luokkaan on toteutettu metodit *attach* ja *detach*, jotka nimensä mukaisesti liittävät ja irrottavat tarkkailija-oliota rekisteristään. Malli ylläpitää listarakennetta tarkkailijoista. JavaScriptissä tämä lista on tyypiltään Array. Tarkkailijan lisääminen tapahtuu yksinkertaisesti metodilla `observers.push(observer)`. *Detach*-metodi ei tarkista, onko lisättävä olio jo listassa, sen sijaan asiakkaan vastuulle jää huolehtia, ettei samaa tarkkailijaa lisätä useampaan kertaan. Tarkkailijaa poistettaessa sen sijaan on pakko iteroida tarkkailijalista läpi. Jos poistettava tarkkailija löytyy, se poistetaan listasta lauseella `observers.splice(indexOfObserver,1),` jossa ensimmäinen parametri *indexOfObserver* on poistettavan alkion indeksi ja toinen poistettavien alkioiden määrä.

Koska näkymä liittää ensin itsensä malliin ja luo sen vasta sen jälkeen ohjaimen, joka myös liittää itsensä malliin, on mallin rekisterissä aina näkymä-olio ennen ohjain-oliota. Näkymää kutsutaan siis aina ennen ohjainta.

Näkymä- ja ohjainluokkien tulee olla *Observer*-luokan instanssi, jotta malli ei rekisteröi vääräntyyppisiä tarkkailijoita. JavaScript ei tue rajapintoja Javan tapaan tai abstrakteja luokkia kuten C++, joten tarkkailija perii *Observer*-luokan prototyypin, jolloin siitä tulee *Observer*-luokan instanssi.

```
function Observer() {}
Observer.prototype = {
    update : function (state) {}
};

View.prototype = new Observer();
Controller.prototype = new Observer();
```

On tärkeää poistaa tarkkailija mallista, kun tarkkailijaa ei enää tarvita.
Mikäli tarkkailijaa ei poisteta mallista kun näkymä tuhotaan, saman tarkkailijan instansseja kertyy mallin rekisteriin useampia kappaleita, josta seuraa, että sama toiminto suoritetaan useampaan kertaan. Esimerkiksi jos mallin tarkkailijarekisterissä on kolme ostokori-instanssia, jokainen ostoskoniin liittyvä toiminto suoritetaan kolmeen kertaan.

Web-sovelluksessa voidaan ottaa lähtökohdaksi, että näkymä sekä sen ohjain irrotetaan mallista, kun näkymän DOM-juurielementti poistetaan *window.document*-oliosta, koska silloin näkymää ei voida enää mitenkään käyttää, vaan se on luotava tarvittaessa uudelleen. Irrottaminen on myös toteutettu näkymien yhteiseen kantaluokkaan *View*.

```
// Poista tarkkailijat mallista.
View.prototype.detach = function () {
   this.model.detach(this); // Irrota näkymä.
   this.controller = this.model.detach(this.controller); // Irrota ohjain.
   this.controller = null; // Vapauta ohjaimelle varattu muisti.
   return this;
};
```

6.7 Tietokantarajapinta

Eräs Ajax-sovellusten eduista on, että asiakas- ja palvelinsovellus voidaan toteuttaa toisistaan täysin riippumattomiksi. Oleellista on vain se, että HTTP-pyyntöön saadaan vastaus, jonka formaattia ja rakennetta asiakassovellus ymmärtää. Esimerkkisovellus käyttää JSON-formaattia ja vastauksen rakennetta kuvaillaan tuonnenpana.

Haettaessa koko ostoskorin sisältö, kutsutaan URL-osoitetta *cart.php* käyttäen GET-metodia, ja vastauksena saadaan ostokorin sisältö JSON-formaatissa. Esimerkkisovelluksen c*art.php*-tiedosto näyttää seuraavalta:

```
$request = Request::getInstance();
$user = Auth::getUser();
$cart = Cart::getInstance($user);

if ($request->isGet()) { // Listaa ostoskorin artikkelit.
    $cart->listItems();
} else if ($request->isPost()) {
    foreach ($request->post() as $item) {
        // Lisää uusi artikkeli tai muuta kappalemäärää.
        $cart->updateItem($item['id'], $item['quantity']);
    }
}

if ('json' == $request->accept) {
    // Tulosta koko ostoskori JSON-formaatissa.
    header('Content-type: application/json');
    echo $cart->toJSON();
} else if ('html' == $request->accept) {
    // Käsittele HTML-pyyntö...
} exit();
```

Lisäys- , ja muokkaus -operaatiossa käytetään POST-metodia. Siirrettävän tietomäärän vähentämiseksi vastauksena saadaan ainoastaan käsitellyn ostoskori-artikkelin tiedot JSON-formaatissa ja artikkelin tiedot päivitetään asiakassovelluksen malliluokassa olevaan ostoskori-tietorakenteeseen. Mikäli URL-osoitteista poistettaisiin php-tiedostopäätе erimerkiksi Apache:n URL Rewrite-tekniikalla, voitaisiin palvelinsovellus vaihtaa esim. Java-servlettiin tarvitsematta tehdä mitään muutoksia AJAX-sovellukseen.

6.8 Sovelluksen toiminta

6.8.1 Tuotelistaus- sivu ja ostoskori

Ostoskori alustetaan heti kun, HTML-dokumentin DOM-rakenne on valmis, sillä se on aina näkyvillä. Sovelluksella on globaali *App*-olio, johon on toteutettu ostoskorin alustus sekä joitakin yleishyödyllisiä metodeja.

```
// Palauttaa objektin, joka sijoitetaan muuttujaan App.
var App = function () {
    $(document).ready(function () {
         var cart = new Cart(),
             cartView = new CartView(cart, 'cart');
         cartView.initialize().display();
    });
    return { someMethod : function () { ... } }
})( ); // Kutsu funktiota välittömästi.
```

Ostoskori-näkymälle eli *CartView*-luokalle annetaan malliksi *Cart*-luokka, sekä toisena parametrina näkymän nimi. Tuotelistaus-sivulla käyttäjä lisää artikkeleita ostoskoriin. Tuotelistaus-sivun Osta-painikkeesen on sidottu *onclick*- tapahtuma , jonka käsittelijä kutsuu ohjaimen *eventHandler*-metodia.

Todellisuudessa ostoskori ei tarvitsisi erillistä ohjainta lainkaan, vaan mallin palveluja voitaisiin kutsua suoraan tapahtumankäsittelijästä , mutta esimerkkisovelluksessa ohjain kuitenkin on toteutettu, sillä tietyssä tilanteessa ostokorin ohjain vaihdetaan sovelluksen oletusohjaimeen, joka ei reagoi mihinkään tapahtumaan.

```
// Ota vastaan ostokorin tapahtumat.
CartController.prototype.handleEvent = function (event, data) {
    switch (event.namespace) {
        case 'insert':
            // Lisää artikkeli koriin.
            this.model.insert(data);
            break;
            ...
         }
}
```

Ohjain kutsuu mallin metodia *insert*, joka puolestaan kutsuu kantaluokan *request*-metodia, jossa HTTP-pyyntö suoritetaan.

```
Cart.prototype.insert = function (item) {
    return this.request("cart.php",
                        item,
                        { method : "POST",
                          state : 'saved' } );
};
```

Parametri state = 'saved', on se viesti, jonka mallin halutaan lähettävän tarkkailijoilleen, kun se laukaisee tapahtumamekanismin.

Kun vastaus HTTP-pyyntöön on saatu, päivitetään mallin data. Yksinkertaisimmillaan ostokori voisi toimia siten, että riippumatta siitä, lisätäänkö, muutetaanko vai poistetaanko artikkeleita korista, vastauksena saadaan aina korin uusi sisältö kokonaisuudessaan, jonka jälkeen se voidaan suoraan päivittää näytölle. Esimerkkisovelluksessa toimintaa on kuitenkin viety hieman pitemmälle. Kun artikkeli lisätään ostoskoriin, sen lukumäärää muutetaan tai se poistetaan ostoskorista, saadaan vastauksena operaation kohteena olleen artikkelin tai artikkeleiden tiedot sekä ostokorin muuttuneet tiedot. Artikkelin lisäyksen suorittaneen HTTP-pyynnön JSON-muotoinen vastaus voi näyttää esimerkiksi seuraavalta:

```
itemCount: 1,
items: [{id:505,
        isNew:false,
        op:”insert”,
        product:”rw01”,
        name:”Cordelia Red Wine 1”,
        price:10,
        quantity:1}],
VATsum: 2.3,
total: 10,
error: null
```

Tarkastellaan edelläolevan JSON-olion sisältöä. Avain *itemCount* on ostokorin artikkeleiden lukumäärää. Items-taulukko sisältää lisätyn artikkelin tunnistetietoja, jotka merkitsevät seuraavaa: *id* on *cart_items*-taulun *id*-sarakkeen arvo, *isNew* viittaa artikkelin

olemassoloon *order_items*-taulussa, eli artikkeli on taululle uusi tai ei. O*p* on suoritettu tietokantaoperaatio, *product* on tuotekoodi, *name* artikkelin kaupallinen nimi, *price* on listahinta ja *quantity* artikkelin lukumäärä *order_items*-taulussa. *Error* on mikä tahansa virhetilanne, esimerkiksi epäonnistuneen tietokantaoperaation Exception-olio tai jonkinlainen virhe liiketoimintalogiikassa. *VATsum* on arvonlisäveron osuus potentiaalisen tilauksen loppusummasta ja *total* loppusumma.

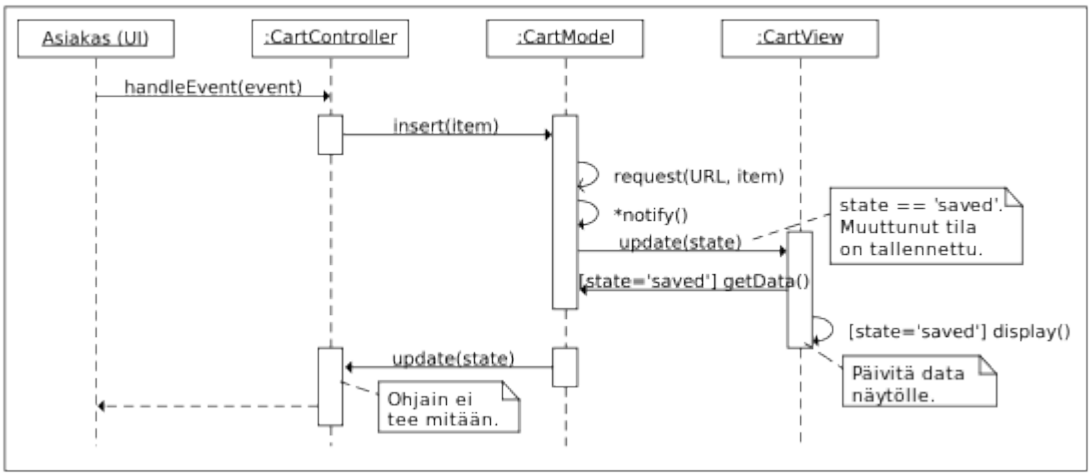
Vastaavasti kappalemäärä muokattaessa *items[index].op* sisältäisi arvon *update,* muuten arvot ovat samat. Poisto-operaatiossa *items[index].op* saa arvon *delete* ja *items[index].isNew* arvon *true,* koska sitä ei enää ole tietokannassa.

```
Cart.prototype.setData = function (cart) {
    var idx, max, item, i;

    this._newItems  = [];

    for (idx = 0, max = cart.items.length; idx < max; idx++) {
        item = cart.items[idx];
        if ('update' == item.op) {
            i = this.hasItem(item);
            if (i > -1) {
                this._cart.items[i] = item;
                this._newItems.push(item);
            }
            continue;

        } if ('delete' == item.op) {
            i = this.hasItem(item);
            if (i > -1) {
                this._cart.items.splice(i, 1);
                this._newItems.push(item);
            }
            continue;
        }
        // insert or read
        this._cart.items.push(item);
        this._newItems.push(item);
    }
};
```

Cart.setData-metodin toiminta on melko suoraviivaista. Ostokori-operaation vastauksena saadun tietorakenteen *cart.items*- taulukko iteroidaan. Mikäli *cart.items[index].op* arvo on '*update'*, päivitetään kyseisen artikkelin lukumäärään selaimen muistissa olevan, käyttäjälle näytettävän ostokori-tietorakenteessa. Mikäli *cart.items[index].op* arvo on '*delete',* se poistetaan olemassa olevasta ostoskorista. Jos taas kysymyksessä on uuden artikkelin lisäys (*cart.items[index].op == 'insert')* tai koko ostokorin sisältö on palautettu palvelimelta (*cart.items[index].op == 'read'),* artikkeli yksinkertaisesti lisätään olemassa olevaan ostoskoriin.

Kun edellä oleva operaatio on saatu suoritettua, malli herättää tapahtumamekanisminsa ja kutsuu tarkkailijoitaan parametrilla 'saved', koska mallin muokattu tila on tallennettu palvelimelle ja vastauksena on saatu tallennettu omaisuus, eli lisätty, muokattu tai poistettu artikkeli. Näkymäluokka (*cartView*) päivittää mallin datan näytölle.

```
CartView.prototype.update = function (state) {
    var item, idx, max;
    // Iteroi taulukko, joka sisältää uudet,
    // muokatut tai poistetut artikkelit.
    for (idx = 0, max = items.length; idx < max; idx++) {
        item = this.model.newItems[idx];
        switch (item.op) { // Tietokantaoperaation tyyppi
            case 'update': // Päivitä artikkelin kappalemäärä
            ...
            break;
            case 'delete':// Poista artikkeli
            ...
            break;
            default: // 'insert' tai 'read'. Lisää uusi artikkeli
            ...
            break;
      }
    }
    // Päivitä loppusumma.
    $("div#total").text(this.model.total);
};
```

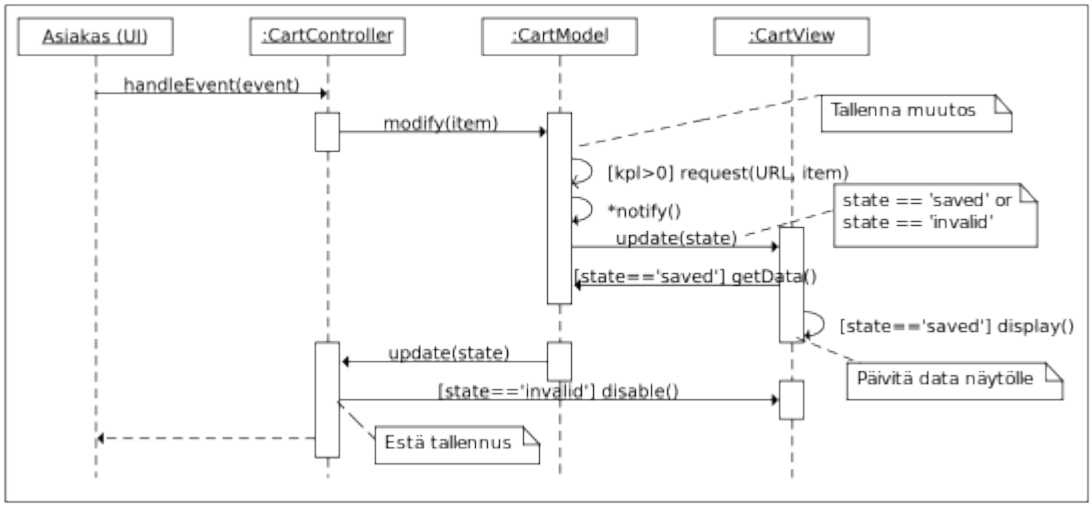


KUVA 7. Vuorovaikutus lisättäessä artikkelia ostoskoriin.

Ostokoria muokataan erillisessä näkymässä, joka on nimeltään *CarItemView*. Muokkaustapahtuma, jossa artikkelin kappalemäärää muutetaan tai artikkeli poistetaan ostoskorista, on havainnollinen esimerkki MVC:n komponenttien välisestä vuorovaikutuksesta. Mikäli käyttäjä asettaa yhdenkin artikkelin kappalemääräksi 0 tai sitä pienemmän luvun, mallin tila muuttuu ”vialliseksi”, sillä artikkelia ei voi ostaa 0 tai -5 kappaletta. Tällöin malli herättää tapahtumamekanisminsa ja lähettää tarkkailijoille viestin ”invalid”. Muokkausnäkymän ohjain reagoi tähän deaktivoimalla *Tallenna*-painikkeen. Jos käyttäjä korjaa tilanteen ja asettaa kappalemääräksi nollaa suuremman luvun, malli lähettää tarkkailijoille viestin ”modified”, johon muokkausnäkymän ohjain reagoi aktvoimalla jälleen *Tallenna*-painikkeen.

KUVA 8. Ohjain on passivoinut *Tallenna*-painikkeen.

Jos käyttäjä tallentaa muutokset, malli tekee HTTP-kutsun joka tallentaa muutokset tietokantaan ja kutsuu sen jälkeen tarkkailijoita viestillä ”saved”, koska muuttunut tila on tallennettu. Sen seurauksena näkymän ohjain deaktivoi *Tallenna*-painikkeen, koska mitään tallennettavaa ei enää ole. Näkymän ei tarvitse reagoida tähän mitenkään. Mikäli käyttäjä poistaa artikkelin, malli tekee HTTP-kutsun, joka poistaa artikkelin tietokannasta ja sen jälkeen kutsuu tarkkailijoita viestillä ”saved”. Tämän seurauksena näkymä päivittää näytölle ostoskorin muutokset. Ohjaimen ei sen sijaan tarvitse reagoida mitenkään.



KUVA 9. Tyypillinen vuorovaikutus muokattaessa mallia (ostoskori).

Kun *CarItemView*-näkymä avataan, halutaan estää käyttäjää toimenpiteet, jotka kohdistuvat *CartView*-näkymään eli ostoskoriin. Tämä voitaisiin tietenkin myös tehdä poistamalla ostokoriin sidotut tapahtumat, tai vaikka piilottamalla ostoskori, mutta esimerkin vuoksi toimitaan toisin.

MVC-arkkitehtuurin keskeisiä ominaisuuksia on mahdollisuus vaihtaa näkymän ohjainta ajonaikaisesti. Niinpä ohjaimen vaihtoa demonstroidaksemme tilausnäkymän avaaminen estetään vaihtamalla muokkausnäkymän avaamisen yhteydessä ostoskorin ohjaimeksi järjestelmän oletusohjaimen, joka ei reagoi mitenkään tapahtumaan.

```
CartController.prototype.handleEvent = function (event, data) {
    switch (event.namespace) {
        case 'open': // Avaa ostoskorin muokkausnäkymä.
            ...
            // Aseta ostoskorin ohjaimeksi oletusohjain.
            var newController = new Controller(this.view);
            // setController vaihtaa uuden vanhan tilalle ja palauttaa vanhan.
            var oldController = this.view.setController(newController);
            oldController = null;
            break;
       ...
    }
}
```

SetController-metodi asettaa uuden ohjaimen, irroittaa vanhan ja palauttaa sen.

```
View.prototype.setController = function (controller) {
    var oldController = this.controller; // Ota vanha ohjain talteen.
    this.controller = controller; // Aseta uusi ohjain.
     // Irrota ja palauta vanha ohjain.
    return this.model.detach(oldController);
};
```

Uusi ohjain, tässä tapauksessa kaikkien ohjainten kantaluokka *Controller*, on jo rekisteröinyt itsensä mallille rakentimessaan.

```
function Controller(view) {
    this.view = view; // Ohjaimella on sekä näkymän että mallin instanssi.
    this.model = view.getModel();
    // Ohjain liittää itsensä mallin tarkkailijaksi.
    this.model.attach(this);
    ...
}
```

Koska muokkausnäkymä on luotu dynaamisesti, eli lisätty sovelluksen DOM-rakenteenseen sivun latauksen jälkeen, se katoaa mikäli käyttäjä syystä tai toisesta lataa sivun uudelleen kyseisen näkymän ollessa avoinna. Dynaamisesti luotu DOM-rakenne voidaan kuitenkin pakottaa avautumaan uudelleen sivulatauksen yhteydessä.
Kun käyttäjä avaa näkymän normaalilla tavalla käyttöliittymästä, lisätään osoiteriville olemassa olevan URL-osoitteen perään #-merkillä (*hash*) alkava merkkijono. Tässä ta-

pauksessa se on *#cart*. Operaation jälkeen osoiterivi saattaisi näyttää esimerkiksi seuraavalta:

```
http://opinnaytetyo/products.php#cart
```

Jokaisen sivunlatauksen yhteydessä tutkitaan löytyykö URL-osoitteesta *#cart*-merkkijonoa. Jos sellainen löytyy, laukaistaan keinotekoisesti sama tapahtuma, jonka käyttäjä normaalisti aiheuttaa käyttöliittymästä käsin:

```
if ('#cart' == window.location.hash) {
    // <a>-elementtin on sidottu myös custom-event 'cart'.
    $('a#cartopen').trigger('cart'); // Aiheuta näkymän avaava event.
}
```

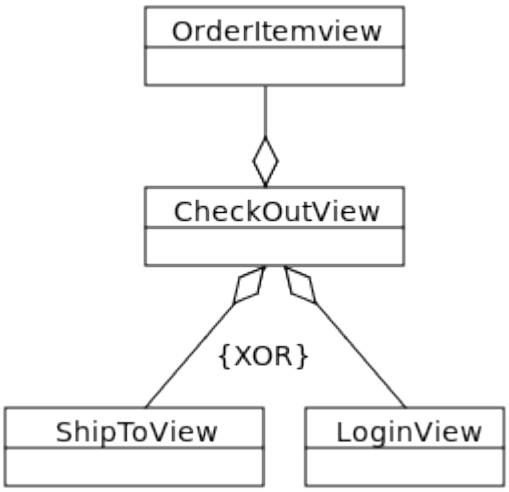
Edelläolevan koodin seurauksen näkymä aukeaa uudelleen, jos sivu ladataan uudelle näkymän ollessa auki.

6.8.2 Tilaus

Tilauksen tekeminen sisältää tavallisesti vähintään kolme vaihetta: tilaustietojen- ja henkilötietojen tarkistuksen ja tilauksen hyväksymisen. Tästä prosessista käytetään monesti englanninkielistä termiä *checkout* eli tarkistus. Esimerkkisovelluksessa kaikki vaiheet on yhdistetty samaan näkymään ikäänkuin ”yhdelle sivulle”. Tilausnäkymä, nimeltään *CheckOutView*, on rekursiokooste, jonka komponentteja tilauksen muita vaiheita edustavat näkymät ovat. *CheckOutView* on *tilausprosessin asiakas*, sillä se alustaa rekursiokoosteen komponentit. Tilausnäkymän ohjain (*CheckOutController*) on myös tilausprosessin asiakas, sillä se ohjailee tilausprosessia ja manipuloi tilausnäkymän komponenttien käyttäytymistä.

Tilausprosessin komponentit ovat siis seuraavat:
- *CheckOutView* eli *tilausnäkymä* on koko tilausprosessin eräänlainen käsitteellinen isäntä ja viitekehys. Toteutuksen tasolla se on rekursiokoosteen ylin komponentti, myös DOM-tasolla.

- *OrderItemView* eli *tilaustietonäkymä* sisältää tilattavat artikkelit.
- *ShipToView* eli *yhteystietonäkymä* sisältää asiakkaan tiedot ja toimitusosoitteen.
- *LoginView* eli *kirjautumisnäkymä* näytetään mikäli käyttäjä ei ole kirjautunut sisään.

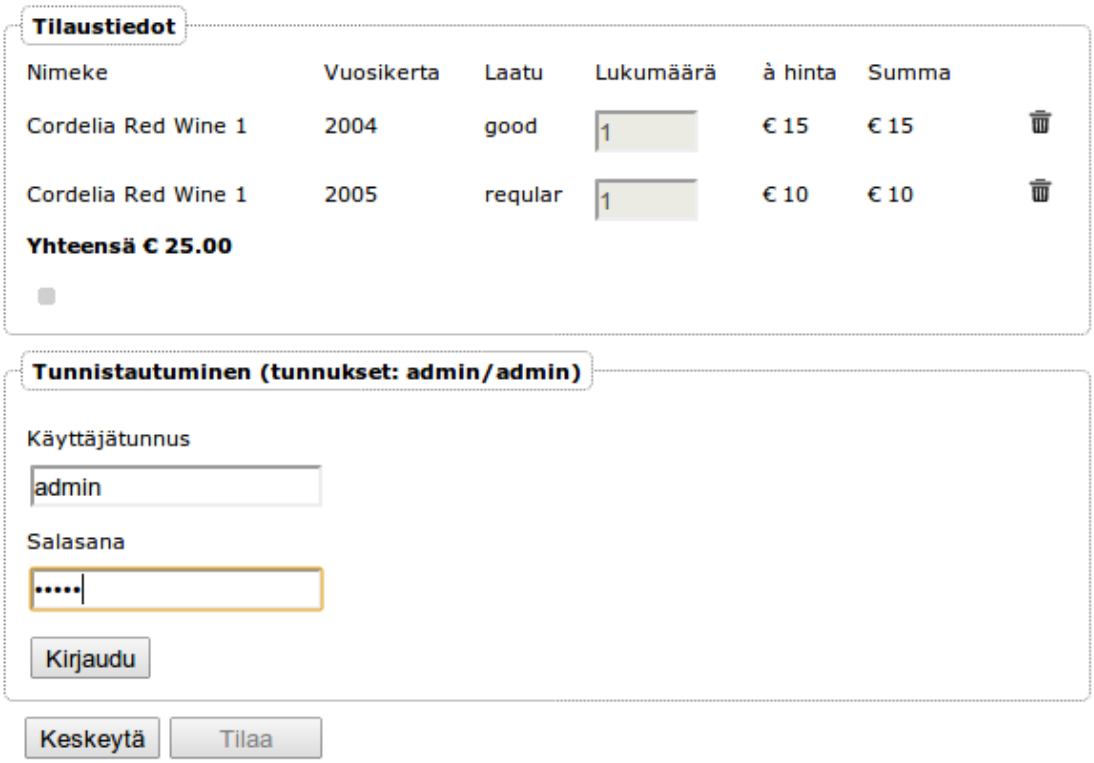


KUVA 10. Luokkakaavio tilauksen näkymäluokista.

Alustusproseduurissaan tilaustietonäkymä luo instanssit lapsinäkymistään ja lisää lapsinäkymiä säilyttävään listarakenteeseen.

```
Var checkOutView = new CheckOutView(Cart::getInstance(), 'checkOut');
checkOutView.initialize().display(); // Alusta ja näytä rekursiivisesti.
...
CheckOutView.prototype.initialize = function () {
    var cart = this.model, var user = cart.user;
    // Tilattavat tuotteet.
    this.orderItemView = new OrderItemView(cart, 'orderItem'),
    // Yhteystiedot.
    this.shipToView = new ShipToView(user, 'shipTo'),
    // Kirjautuminen.
    this.loginView = new LoginView(user, 'login');
    // Lisää lapsinäkymät.
    this.addChild(this.orderItemView);
    this.addChild(this.shipToView);
    this.addChild(this.loginView);
}
```

Mikäli käyttäjä ei ole kirjautunut, näytetään kirjautumisnäkymä . Jos kirjautuminen onnistuu, kirjautumisnäkymä piilotetaan ja yhteystietonäkymä näytetään. Tilaustietonäkymä on näkyvillä koko ajan, mutta jos käyttäjä ei ole kirjautunut, vaihdetaan tilaustietonäkymän ohjaimeksi sovelluksen oletusohjain, joka ei reagoi mihinkään käyttäjän toimenpiteeseen. Näin estetään käyttäjää tekemästä muutoksia tilaukseen ennen kirjautumista. Jos käyttäjä kirjautuu onnistuneesti, asetetaan tilaustietonäkymän ohjaimeksi tietenkin näkymän oma ohjain.

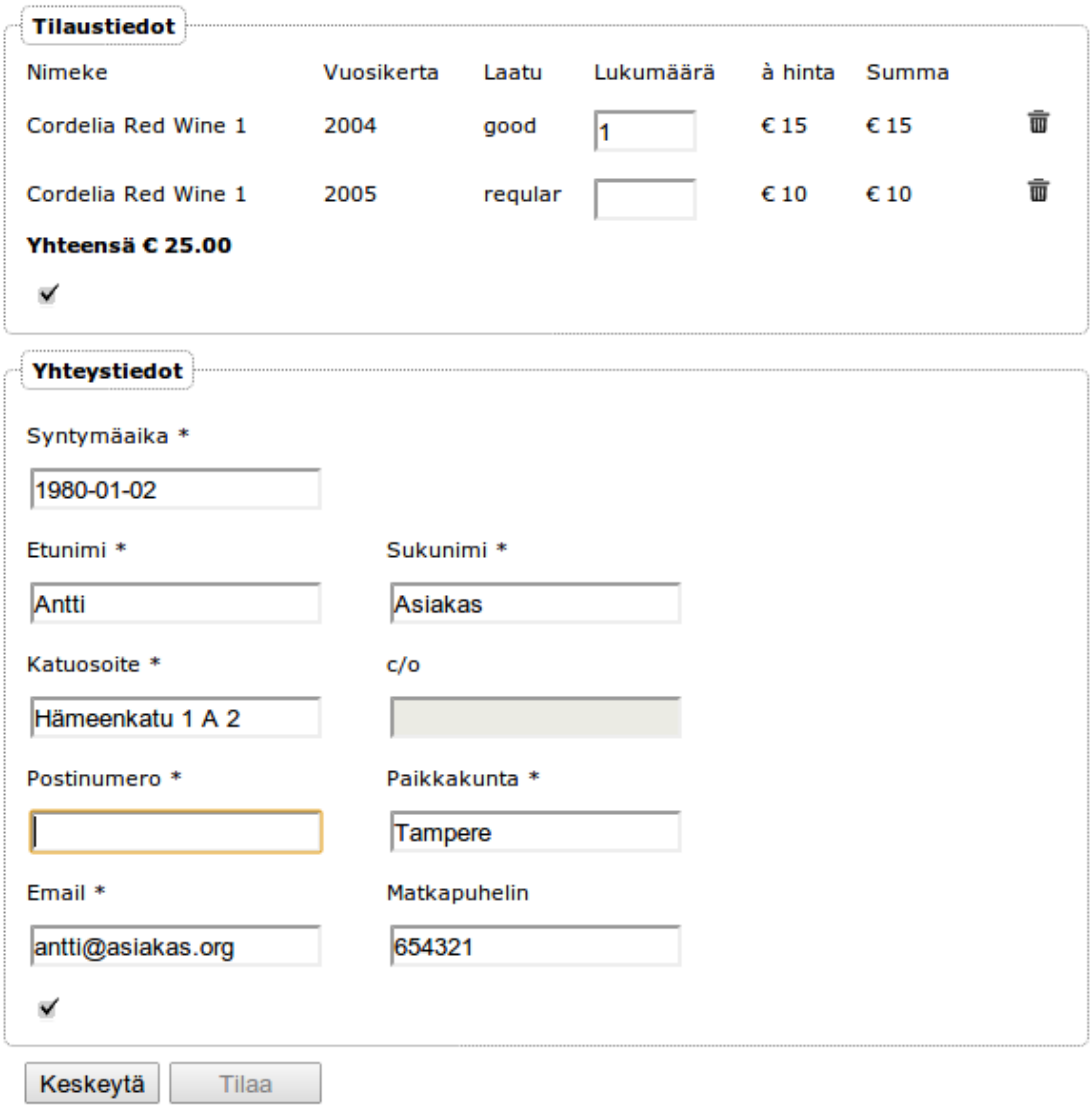


KUVA 11. Käyttäjän tunnistautuminen tilausvaiheessa.

Tilaustietonäkymä on tosiasiassa vain tietynlainen näkymä ostoskori-olion (*cart*) datasta. Samoin yhteystietonäkymä on yksi käyttäjä-olion (*user*) näkymistä. *CheckOutView*-rekursiokoosteen mallina on cart-olio. Koska ostoskorin ja käyttäjän suhde on yksi yhteen, on *Cart*-luokalla täytyy olla jäsenmuuttujanaan *user*-olio. Tällöin tilausnäkymän ja sen ohjaimen on mahdollista hyödyntää molempia malliolioita.

Käyttäjä voi muokata tilaus- tai yhteystietoja vielä tilausvaiheessa ja muutokset täytyy tietenkin tallentaa ennen tilauksen hyväksymistä. Muutokset tallennetaan automaattises-

ti juuri ennen tilausta. Muutosten tallentaminen edellyttää, että molemmat tilausprosessin mallit ovat tallennukseen sopivassa tilassa, eli esimerkiksi artikkelin kappalemäärä on nollaa suurempi tai pakolliset yhteystiedot on annettu. Mikäli tilauksen tiedot ovat ”OK”, *Tilaa*-painike voidaan aktivoida, muutoin se pysyy passivoituna, eikä käyttäjä pysty tallentamaan tilausta.



KUVA 12. Isäntänäkymän ohjain on estänyt tilauksen tallentamisen, koska yhden tai useamman lapsinäkymän mallin tila on epäkelpo.

Koska koko tilauksen mahdollistaminen on hierarkian ylimmäisen näkymän eli tilausnäkymän (*CheckOutView*) vastuulla, tilaus- ja yhteystietonäkymät ensin prosessoivat

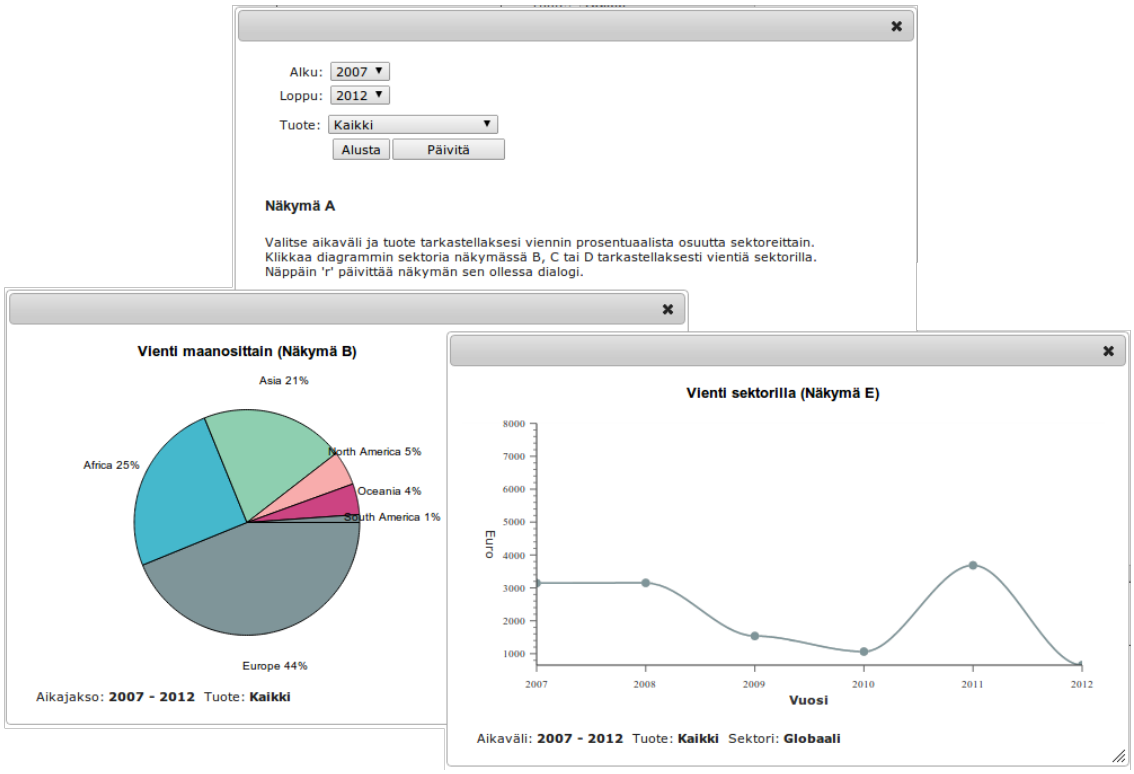
käyttäjän syötteet normaalisti, ja sen jälkeen välittävät tapahtuman seuraajalle implisiittisessä vastuuketjussa. Seuraajana tässä tapauksessa toimii isäntänäkymän ohjain, joka tarkistaa *cart*- ja *user*- olioiden data tilan. Mikäli molempien olioiden data on validia, isäntänäkymän ohjain tekee tilauksen tallentamisen mahdolliseksi aktivoimalla tilauspainikkeen.

```
// Tilaustietonäkymän ohjain.
OrderItemController.prototype.handleEvent = function (event, data) {
    switch (event.namespace) {
       case 'modify' :
                // Muuta tuotteen kappalemäärää.
                this.model.setProperty(data.id, data.quantity);
                // Kutsu isäntänäkymän ohjaimen handleEvent-metodia.
                this.getSuccessor().handleEvent(event, data);
            break; ...
     }
}

// Yhteystietotietonäkymän ohjain.
ShipToController.prototype.handleEvent = function (event, data) {
    switch (event.namespace) {
       case 'modify' :
                // Muuta yhteystietoa.
                this.model.setProperty(data.prop, data.val);
                // Kutsu isäntänäkymän ohjaimen handleEvent-metodia.
                this.getSuccessor().handleEvent(event, data);
            break; ...
     }
}

// Isäntänäkymän ohjain.
CheckOutController.prototype.handleEvent = function (event, data) {
    var cart = this.view.model, user = cart.user;
    switch (event.namespace) {
        case 'modify':
            cart.isValid && user.isValid // Tilaus- ja yhteystiedot OK.
                ? // Aktivoi näkymän Tilaa-painike.
                : // Passivoi näkymän Tilaa-painike.
        break; ...
    }
}
```

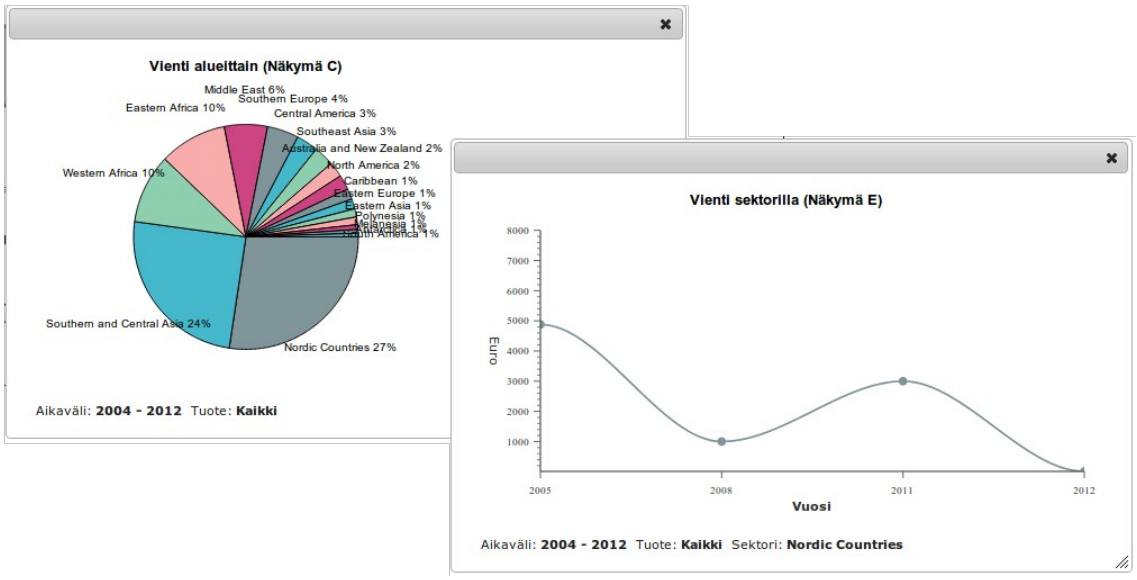
Kun käyttäjä hyväksyy tilauksen ja painaa ”Tilaa”-painiketta, tilaus tallennetaan. Ensin tallennetaan mahdolliset muutokset tilaus- tai yhteystiedoissa tauluihin *cart* ja *user*. Itse tilaus on suoritettu kun *cart*- ja *cart_item*- taulujen tiedot kyseessä olevan tilauksen osalta on kopioitu *order*- ja *order_item*- tauluihin.

6.8.3 Myynnin graafinen esitys

Tilastollisen tiedon graafinen esittäminen lienee parhaimpia esimerkkejä samanaikaisista ja synkronoiduista näkymistä, koska samasta tietolähteestä muodostaa lukemattomia erilaisia diagrammeja käyttötarkoituksen mukaan. Esimerkissä tarkastellaan mm. kuvitellun viennin volyymiä maanosittain, alueittain (esim. pohjoismaat, eteläinen Eurooppa) ja maittain.



KUVA 13. Kuvassa kaksi näkymää samasta tietolähteestä: vienti maanosittain ja vienti euroina globaalisti eli kaikilla ympyrädiagrammin sektoreilla. Ylimpänä näkyy kontrollidialogi, joka asettaa näkymien parametrit ja päivittää näkymät.



KUVA 14. Alueellista (Pohjoismaat, Lähi-Itä jne.) vientiä kuvaava ympyrädiagrammi ja myyntiä kuvaava viivadiagrammi. Käyttäjä on valinnut sektorin Pohjoismaat ja viivadiagrammi on päivittynyt.

Esimerkin tilasto-osio kokonaisuudessaan on komposiittinäkymä. Rekursiokoosteen isäntänäkymänä ja DOM-rakenteen juurena toimii *StatsView*-näkymä, jonka lapsikomponentteja muut näkyvät ovat. Esimerkin koko tilasto-osiota kutsutaan tästä eteenpäin *tilastonäkymäksi* ja sen lapsinäkymiä *kaavionäkymiksi.*

StatsView alustetaan komposiittinäkymälle tyypilliseen tapaan seuraavasti:

```
var statsView = new StatsView(undefined, 'stats'),
    barChartView = new BarChartView(model, 'barChart'),
    lineChartView = new LineChartView(model, 'lineChart');

// Lisää lapsikomponentit.
statsView.addChild(barChartView);
statsView.addChild(lineChartView);

statsView.initialize(); // Alusta näkymä rekursiivisesti.
statsView.display();  // Näytä näkymä rekursiivisesti.
```

Uuden lapsinäkymän lisääminen ajonaikaisesti tapahtuu seuraavasti:

```
var pieChartView = new PieChartView(m, 'pieChart');
statsView.addChild(pieChartView);
pieChartView.initialize().display();
```

Poistaminen isäntänäkymästä vastaavasti tapahtuu seuraavasti:

```
pieChartView = statsView.removeChild(pieChartView);
pieChartView.destroy();
```

Koko tilastonäkymä lapsineen voidaan poistaa poistamalla isäntänäkymä.

```
statsView.destroy();
```

Edelläoleva komento poistaa ensin isäntänäkymän ja lapsinäkymien DOM-rakenteen HTML-dokumentin DOM-puusta ja sen jälkeen irrottaa rekursiivisesti lapsinäkymät ja niiden ohjaimet niiden malleista. Näkymän DOM-elementtien poistamiseen ei ole tarvinnut erikseen toteuttaa rekursiivista poisto-operaatiota, sillä kuten jo aiemmin mainittiin, DOM-malli itsessään on rekursiokooste ja DOM-metodi *element.removeChild(node*) poistaa elementin lapsineen.

Diagrammeja esittävien näkymien toiminta on hyvin suoraviivaista. Mallin ei tarvitse lähettää tarkkailijoille mitään lisäparametreja aktivoidessaan tapahtumamekanismin, vaan näkymät yksinkertaisesti päivittävät näkymän, kun päivitysproseduuria kutsutaan. Diagrammit on toteutettu Dojo Toolkit-kirjaston avulla. Kaavionäkymien päivitysproseduurin periaate on yksinkertaistettuna seuraavanlainen:

```
var self = this;
dojo.require("dojox.charting.Chart2D"); // Tuo tarvittavat moduulit.
...
dojo.ready(function () { // Moduulit on ladattu. Dojo kutsuu.
    if (null === self.chart) { // Diagrammi on alustamatta.
        // Luo diagrammi ja aseta data...
    } else  { // Päivitä diagrammin data... }
    self.chart.render(); // Näytä diagrammi.
}
```

Koska tilastonäkymä on komposiittinäkymä, diagramminäkymät hyödyntävät lapsi-isäntäsuhteen määrittämää implisiittistä vastuuketjua. Näkymä F on ympyrädiagrammi, joka kuvaa tuotteiden prosentuaalista osuutta myynnistä tiettynä ajanjaksona. Kun diagrammin tuotetta edustavaa sektoria klikataan hiirellä, tapahtuma välitetään näkymän ohjaimelle. Ohjain ei kuitenkaan käsittele tapahtumaa mitenkään, koska näkymä esittää prosentuaalisia osuuksia, eikä yhden tuotteen tietoja voida käsitellä mitenkään. Siksi ohjain välittää tapahtuman seuraavalle käsittelijälle, joka tässä tapauksessa on näkymän F isäntä (näkymä A), joka voi käsitellä tapahtuman. Näkymä A, joka on myös kaikkien muiden diagramminäkymien isäntä, on lomakekontrolleja sisältävä näkymä, jonka tarkoitus on asettaa hakuehtoja tilastotiedon noutamista varten. A-näkymän ohjain voi käsitellä F:n lähettämän tiedon tuotteesta, joten se laittaa mallille hakuehdoksi vastaanotetun tuotteen ja pyytää mallia tekemään uuden HTTP-kutsun. Ratkaisu ei varmasti ole kovin käytännöllinen, mutta se on esimerkki kuinka komposiittinäkymässä voidaan hyödyntää hierarkian muodostamaa implisiittistä komentoketjua ja uudelleenkäyttää rekursiokoosteen komponentteja ilman eksplisiittisiä viittauksia toisiin komponentteihin.

7 POHDINTA

Tämän opinnäytetyön esimerkit toteutettiin käyttäen MVC-arkkitehtuuria, ja lähes kaikki toiminnot toteutettiin huomattavasti monimutkaisemmin kuin olisi ollut tarpeen. On täysin selvää, että useat verkkokaupan toiminnot, kuten ostoskorin, voi toteuttaa jo pelkästään muutamalla `XMLHttpRequest.send`-kutsulla, jossa kaikki mahdollinen liiketoimintalogiikka ja näytönpäivitys tapahtuu takaisinkutsumetodissa. (`XMLHttpRequest.onreadystatechange`). Melko selvää on myös se, että verkkokaupan määrittelyvaiheessa on tiedossa, kuinka monta ”näkymää” kustakin sovelluksen ominaisuudesta tai käsitteestä tullaan muodostamaan . Tällöin voi olla käytännöllisempiä, helpompia ja nopeampia ratkaisuja olemassa kuin MVC:n toteuttaminen.

Webiin on kuitenkin mahdollista toteuttaa varsin erikoisiakin sovelluksia, kuten erilaisia hallintapaneeleita, web-desktopeja tai jopa toimisto-ohjelmistoja kuten Google Apps. Tällaisten sovellusten kohdalla on mahdollista, että sovellusta joko halutaan muuntaa tai laajentaa tai sovellusta on pakko muuttaa jossain vaiheessa sen elinkaarta. Esimerkiksi web-palvelimen etähallintapaneeli täytyy saada tukemaan palvelimen uuden version ominaisuuksia mahdollisimman vähillä muutoksilla sovellukseen. Tai etähallintapaneeliin halutaan lisätä toisenlainen käyttöliittymä tiettyä käyttäjäryhmää varten. Tällaisissa tapauksissa on mielekästä ottaa sovelluksen suunnittelun pohjaksi MVC-arkitehtuuri, koska se eristää sovelluksen ydintoiminnot, käyttäytymisen ja tiedon esittämisen omiin komponentteihinsa, joita on mahdollista muunnella vaikuttamatta niiden sovellusten osien toimintaan, jotka ovat ko. komponentista riippuvaisia.

Suunniteltaessa MVC:n käyttöä sovelluksen arkkitehtuurina, on hyvä pitää mielessä joitakin periaatteita. Ohjaimen tarkoitus sovelluksessa on toiminnan muuntelu *ajonaikaisesti* vaihtamalla ohjainta tarvittaessa. Mikäli on yksiselitteistä, että jonkin sovelluksen osa-alueen toimintaa tai käyttäytymistä ei tarvitse muunnella ajonaikaisesti, erillisen ohjainkomponentin voi jättää toteuttamatta ja toteuttaa ohjaimen toiminnot näkymäluokkaan. Erillisen ohjaimenhan voi lisätä jälkeenpäin, jos sellainen tarve ilmaantuu.

Mikäli ohjaimen ei tarvitse reagoida mallin tilan muutoksiin, eli toisinsanoen ohjaimen *update*-metodi jäisi ”tyhjäksi”, ohjainta ei tarvitse liitää malliin. Näin vältytään turhilta päivityskutsuilta.

Komposiittinäkymää käytetään, jos tarvitaan sisäkkäisiä näkymiä ja niiden määrä ja käyttäytyminen muuttuu ajonaikaisesti. Näkymät voivat olla mitä tahansa yksintaisesta painikeryhmästä monimutkaiseen tietorekisterin muokkausnäkymään. Vastuuketju-mallia hyödynnetään, jos käyttäjän toimenpiteen eli pyynnön käsittelijä ei ole etukäteen tiedossa tai pyynnön voi käsitellä useampi objekti. Komposiittinäkymän implisiittistä vastuuketjua hyödynnetään, jos se vastaa tarkoitusta. Muutoin asetetaan eksplisiittinen vastuuketju. Vastuuketju-malli helpottaa sovelluksen komponenttien uudelleenkäyttöä, koska suoria viittauksia (*tight coupling*) muihin objekteihin ei tarvitse asettaa, vaan viittaus pyynnön käsittelijään on aina epäsuora ja dynaaminen.

Kuten luvussa 4.2 esitettiin, AJAX-sovellusten tietoturva on olematon, koska lähdekoodi on aina saatavilla ja sen tulkitsemisen estäminen on ainakin teoreettisesti mahdotonta. Tästä seuraa se että palvelinsovelluksen logiikan siirtämisellä selaimeen voi olla tietoturvan kannalta arvaamattomat seuraukset. Kräkkerin tarvitsee vain saada selville joko lähdekoodia tulkitsemalla (salattu yhteys) tai verkkoliikennettä tarkkailemalla (salaamaton yhteys), mitä parametreja sovellus lähettää HTTP-pyynnössä, ja seurauksena kaikki asiakassovelluksen liiketoimintalogiikka on hyödytöntä. Tämä tietysti koskee kaikkia web-palveluja. Esimerkiksi lomakkeen validointitoimintojen tarkoitus on pohjimmiltaan käyttäjän opastaminen ja turhien HTTP-kutsujen välttäminen. Kaikki palvelimelle lähetetty data aina tarkistettava, jos siihen liittyy jotain kriteereitä. Web-palvelun tietoturva toteutetaan aina palvelimensovellukseen.

Tästä syystä AJAX-sovelluksen rooliksi jää lähinnä tiedon säilyttäminen HTTP-kutsujen välillä. Asiakassovelluksen tieto ladataan palvelimelta ja se tallennetaan sinne, jos sitä muokataan. Tietoa voidaan järjestellä tai sen rakennetta voidaan muuttaa, mutta sen tulee silti olla aina palvelimen tietomallin osajoukko.

Avainkysymyksiksi MVC-valinnassa web-sovelluksen arkkitehtuuriksi jää tiedon synkronointi ja sovelluksen laajennettavuus. Onko tarvetta synkronoida useita ja samanaikai-

sia näkymiä? Onko tarvetta lisätä tuki erityyppisille käyttöliittymille tai erilaisille tiedonsyöttötavoille sovelluksen elinkaaren aikana.

Vielä yksi näkykulma on hyvä ottaa esiin. Jos edellämainitut kriteerit täyttävä moninutkainen ja kriittinen web-palvelu toteutetaan intranetiin jonkin rajatun ja kontrolloidun käyttäjäryhmän käyttöön, onko mielekästä toteuttaa web-client eli web-pohjainen asiakassovellus. Esimerkiksi Swing-sovellus voi olla tietoturvan, suorituskyvyn ja jatkokehityksen kannalta parempi vaihtoehto. Web-clientin keskeinen idea on sen riippumattomuus asiakkaan tyypistä ja sijainnista. Asiakas voi siis olla mikä tahansa selain, matkapuhelin tai tabletti kunhan se vain sijaitsee WWW:ssä ja on oikeutettu käyttämään kyseistä palvelua. Jos näitä periaatteet eivät täyty, kannattaa punnita vaihtoehtona Destop-sovellusta.

LÄHTEET

Asleson, R. & Schutta, T. 2007. Ajax – tehokas hallinta.

Buschmann, F. 1996. A System of Patterns - Pattern-Oriented Software Architecture. Wiley.

Crane D, Pascarello E & James D. 2006. Ajax in action. Manning.

Ecma International. 2011. Standard Ecma-262, 5.1 edition. Luetttu 17.11.2012. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

Gamma E., Helm R., Johnson, R. & Vlisides J. 1997. Design Patterns - Elements of Reusable Object Oriented Software, Addison-Wesley,

Koskimies, K. & Mikkonen, T. 2005. Ohjelmistoarkkitehtuurit. Jyväskylä: Talentum Media Oy.

Stefanov, S. 2010. JavaScript Patterns. O'Reilly.

LIITTEET

Liite 1. Esimerkkisovellus ja sen JsDOC-dokumentaatio

Esimerkkisovellus on osoitteessa http://home.tamk.fi/~c8kkoho/ONT/ ja sen JsDOC-dokumentaatio on osoitteessa http://home.tamk.fi/~c8kkoho/ONT/jsdoc.