

Ohjelmistotekniikan menetelmät

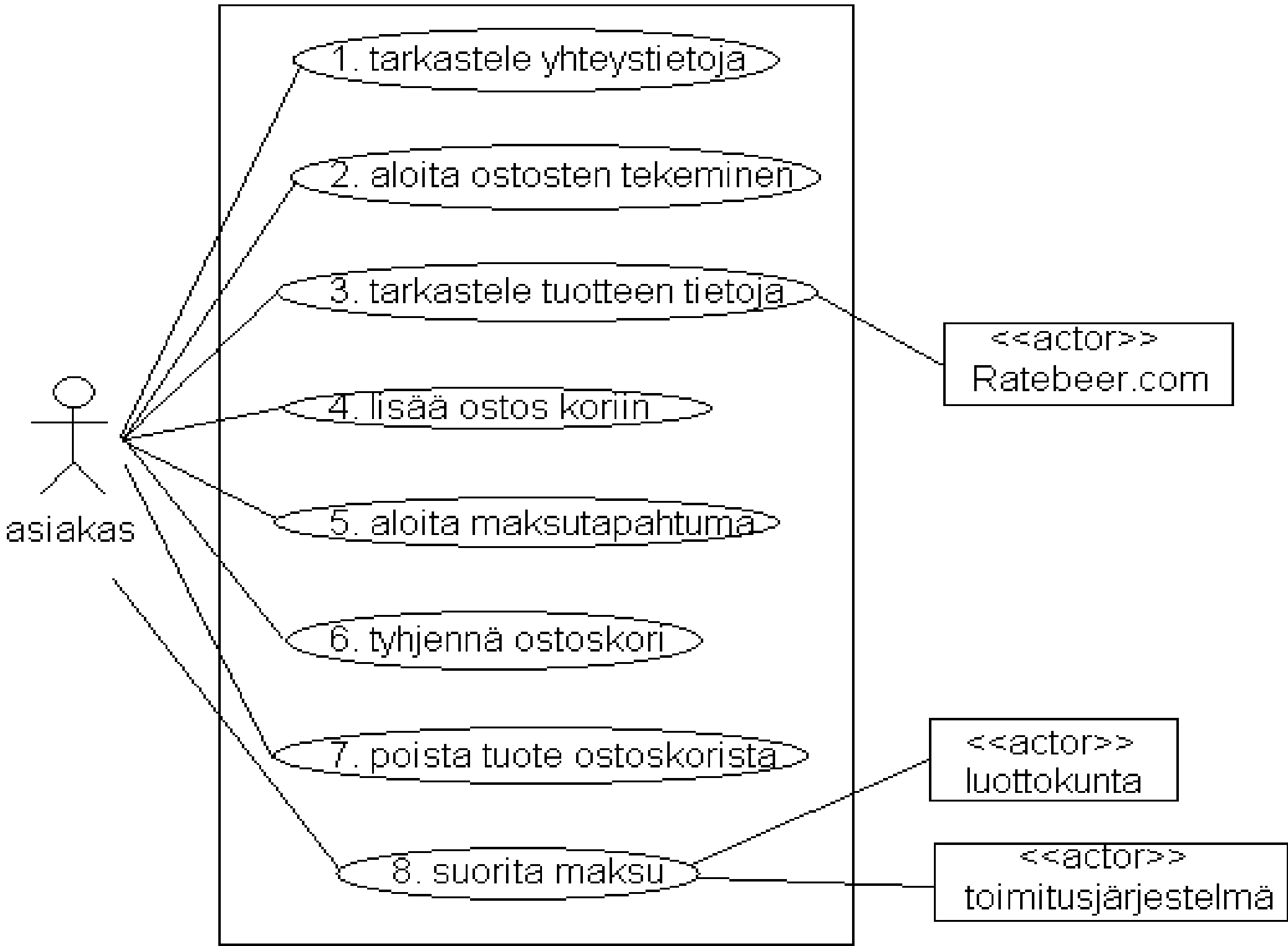
Luento 5, 26.11.

Kumpulabiershop

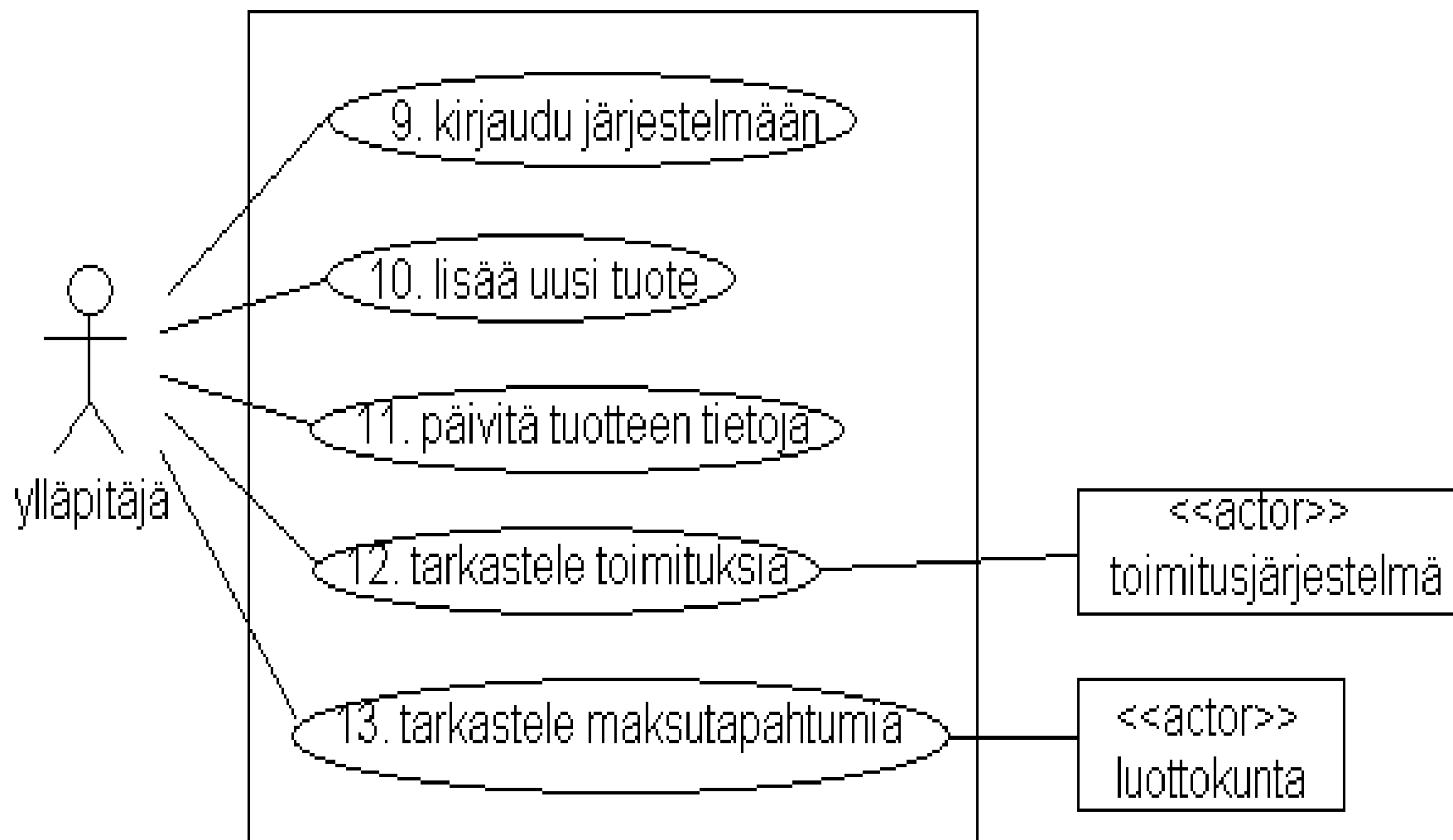
Määrittely ja suunnittelua

Kumpula Biershopin vaatimusmäärittely ja -analyysi

- Kuten kurssilla on useaan kertaan mainittu, sisältää ohjelmistotuotantoprojekti seuraavat vaihteet:
 - Vaatimusmäärittely ja analyysi: *mitä tehdään*
 - Suunnittelu: *miten tehdään*
 - Toteutus: *koodataan*
 - Testaus: *tehtiinkö oikein*
- Olemme laskareiden 2 ja 3 yhteydessä tehneet vaatimusmäärittelyä
 - Ensin määriteltiin tilaajan ohjelmistolle asettamia toiminnallisia vaatimuksia käyttötapausmallin avulla
 - Ja toissa viikolla teimme sovelluksen käsitteistöä jäsentävän määrittelyvaiheen luokkamallin
- Käyttötapausmalli ja määrittelyvaiheen luokkamalli antavat hyvän pohjan järjestelmän suunnittelulle
- Biershopin käyttötapauskaavio seuraavilla sivuilla



Biershopin käyttötapauskaavio jatkuu



Käyttötapausten kuvaus

- Keskitytään tällä luennolla käyttötapausten ***lisää ostos koriin ja suorita maksu*** toiminnallisuuden suunnitteluun
- Käyttötapausten kuvaus seuraavassa
- ***käyttötapaus***: lisää ostos koriin
- ***käyttäjät***: asiakas
- ***esiehdot***: asiakas on tuotelistanäkymässä
- ***käyttötapausten kulku***:
 1. asiakas lisää tuotteen ostoskoriin
 2. ostoskorissa olevien tuotteiden määrä ja niiden yhteishinta päivittyy sivulle

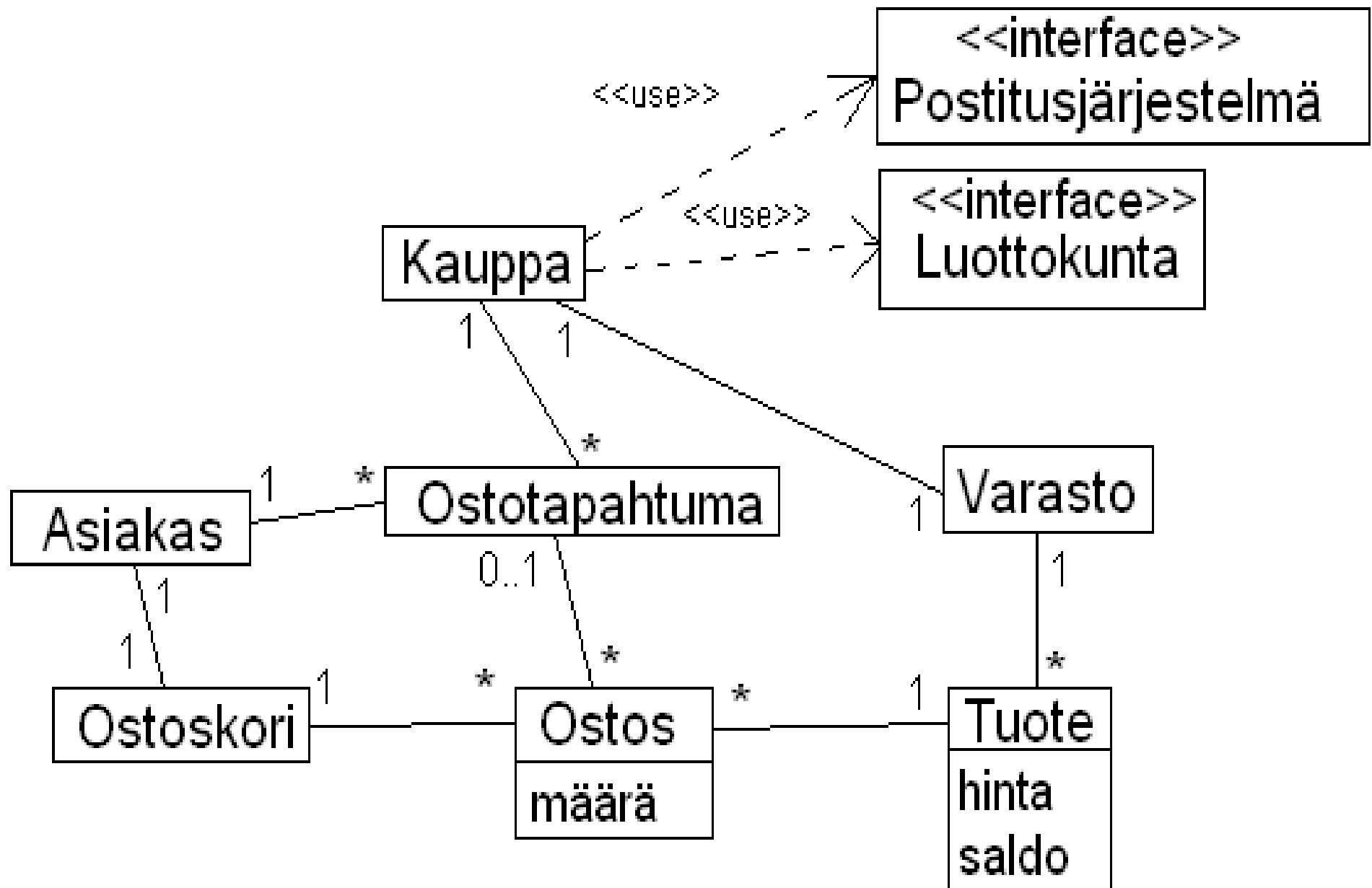
Käyttötapausten kuvaus jatkuu

- ***käyttötapaus***: suorita maksu
- ***käyttäjät***: asiakas, luottokunta, toimitusjärjestelmä
- ***esiehdot***: ollaan maksutapahtumanäkymässä ja korissa on ainakin yksi ostos
- ***käyttötapausten kulku***:
 1. asiakas täyttää tietonsa (nimi, osoite, luottokorttinumero) maksutietolomakkeelle ja valitsee maksu-toiminnon
 2. Jos asiakkaan tiedot ovat kunnossa ja luottokunta hyväksyy maksun, ilmoitetaan maksun onnistumisesta
 3. Hyväksytyn ostotapahtuman tiedot (ostajan nimi ja osoite sekä ostetut tuotteet) ilmoitetaan toimitusjärjestelmälle
- ***poikkeuksellinen toiminta***:
 - 2a. Jos maksua ei hyväksytä, ilmoitetaan maksun epäonnistuneen ja ohjataan asiakas takaisin maksusivulle.

Määrittelyvaiheen luokkamalli

- Seuraavalla sivulla on laskareissa tehty Biershopin määrittelyvaiheen luokkamalli
 - Kuten aiemmissa kalvoissa on todettu, kutsutaan määrittelyvaiheen luokkamallia myös kohdealueen luokkamalliksi sillä siinä esiintyvät oliot ovat ”sovelluksen kohteen” olioita
 - Englanninkielessä on yleisesti käytössä termi **domain model**
- Määrittelyvaiheen luokkamalli siis kuvaa järjestelmän ydinkäsitteitä ja niiden välisiä suhteita
- Määrittelyvaiheen luokkamalli otetaan yleensä suunnittelun ja toteutuksen pohjaksi
- Osa määrittelyvaiheen luokista muuttuu teknisen tason luokiksi jotka sitten toteutetaan ohjelmointikielellä
- Suunnittelu- ja toteutusvaiheessa myös usein hylätään jotain määrittelyvaiheen luokkia
- Mukaan otetaan mukaan uusia luokkia, mm.:
 - Käyttöliittymän toteutukseen ja tietokantayhteyksiin liittyvät luokat
 - Toimintojen suorittamisesta vastaavat sovelluksen ohjausoliot

Biershopin määrittelyvaiheen luokkakaavio



Tarkennuksia luokkamallin käsitteisiin

- Tuote

- Oliot edustavat yhteen myynnissä olevaan tuotteeseen liittyviä tietoja kuten tuotteen nimi, tarkempi kuvaus, varastosaldo, hinta
- Yhtä myynnissä olevaa oluttyyppiä, esim *Weihenstephaner Hefe Weissbier* kohti on olemassa yksi tuote-olio
- Tuote **ei siis edusta** yksittäistä myynnissä olevaa olutpulloa/tölkkiä

- Ostos

- Tuote-olioiden sijaan ostoskoriin viedään Ostos-olioita
- Olio tietää mistä tuotteesta on kysymys ja kuinka monta pulloa/tölkkiä kyseistä tuotetta korissa on

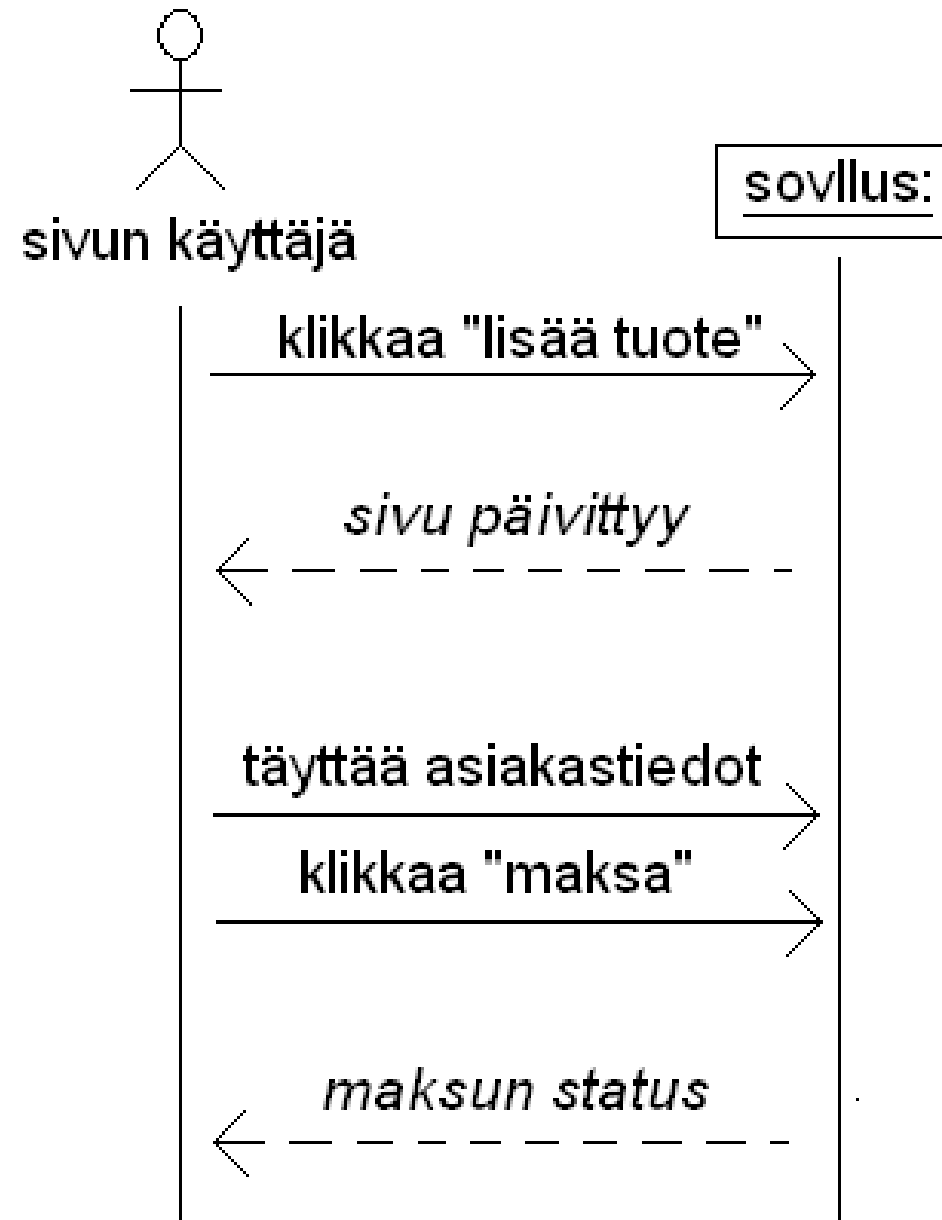
- Ostotapahtuma

- Ostoskori on asiakkaan käytössä vaan ostoksia tehtäessä
- Kun ostokset on maksettu, tallettuu järjestelmään tieto ostoksesta Ostotapahtuma-olioon
- Olio tietää ostotapahtumaan liittyvän asiakkaan tiedot sekä listan tehdyistä ostoksista
- Ostotapahtuma-olioiden tärkein tehtävä on siis ”muistaa” kaupankäyntihistoria

Kohti suunnittelua

Järjestelmätason sekvenssikaaviot

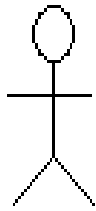
- Toimiakseen halutulla tavalla, on järjestelmän tarjottava ne toiminnot tai operaatiot, jotka käyttötapauksen läpiviemiseen vaaditaan
- Joissain tilanteissa on hyödyllistä dokumentoida tarkasti, mitä yksittäisiä operaatioita käyttötapauksen toiminnallisuuden toteuttamiseksi järjestelmältä vaaditaan
- Dokumentointiin sopivat *järjestelmätason sekvenssikaaviot*, eli sekvenssikaaviot, joissa koko järjestelmä ajatellaan yhtenä oliona



Käyttöliittymän ja sovelluslogiikan eriyttäminen

- Edellisen kalvon järjestelmätason sekvenssikaavio keskittyy käyttäjän ja järjestelmän väliseen interaktioon, eli siihen miten käyttäjä kommunikoi järjestelmän käyttöliittymän kanssa
- Käyttöliittymä ja varsinainen sovelluslogiikka kannattaa monestakin syystä eriyttää toisistaan, ja näin tehdään myös Biershopin toteutuksessa
- Tehdäänkin hieman tarkempi järjestelmätason sekvenssikaavioesitys, jossa järjestelmä kuvataan kahtena ”olioina”, käyttöliittymänä ja sovelluslogiikkana:
 - Käyttöliittymä ottaa vastaan vastaan käyttäjän interaktion (esim. näppäinten painallukset ja syötteen antamisen) ja tulkitsee ne järjestelmän toiminnoiksi
 - Sovelluslogiikka vastaa varsinaisesta toiminnasta
- Nämä kaksi oliota eivät siis ole lopullisen sovelluksen oikeita olioita, ne ainoastaan kuvaavat järjestelmän rakenteen jakautumista erillisiin komponentteihin (joiden sisällä on ”oikeita” ohjelmointikielellä toteutettuja olioita)
- Seuraavalla sivulla käyttötapausten tarkempi järjestelmätason sekvenssikaavio

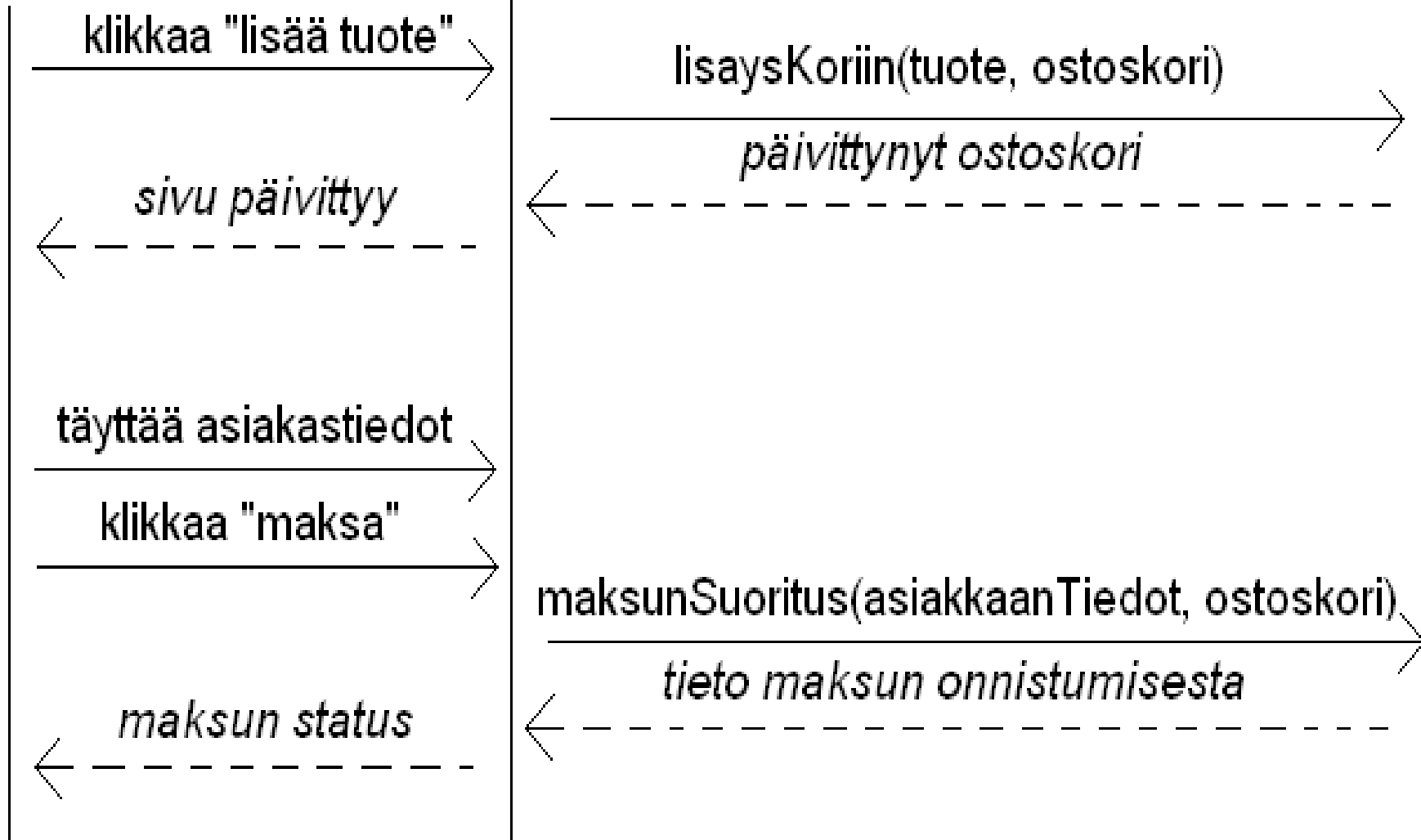
Tarkempi järjestelmätason sekvenssikaavio käyttötapausta lisää ostos koriin ja suorita maksu



sivun käyttäjä

UI:

sovelluslogiikka:



Käyttötapaukset aiheuttavat luokkamallin tasolla tapahtuvia asioita

- Käyttötapausten suorittaminen aiheuttaa käyttäjälle suoraan näkyviä toimenpiteitä, esim:
 - Käyttöliittymänäkymän päivittyminen
 - siirtyminen toiseen näkymään
- Käyttötapausten suorittaminen aiheuttaa myös järjestelmän sisäisiä asioita
 - luodaan, päivitetään, poistetaan järjestelmän kohdealueen olioita ja niiden välisiä suhteita
 - kohdealueen olioita ovat siis määrittelyvaiheen luokkakaavion oliot (tuote, ostos, ostoskori, ...)
- Käyttötapausten suorittaminen saattaa myös edellyttää muiden järjestelmien kanssa tapahtuvaa kommunikointia
 - Esim. luottokortin veloitus luottokunnan verkkorajapinnan avulla
- Seuraavalla kalvolla tarkennetaan seuraavassa tarkasteltavien käyttötapausten **lisää ostoskoriin** ja **suorita maksu** suorittamisen aiheuttamia järjestelmän sisäisiä asioita ja kommunikointia ulkoisten järjestelmien kanssa

Käyttötapausten luokkamallin tasolla aiheuttavat toimenpiteet sekä kommunikointi ulkoisten järjestelmien kanssa

- ***lisää ostos koriin***

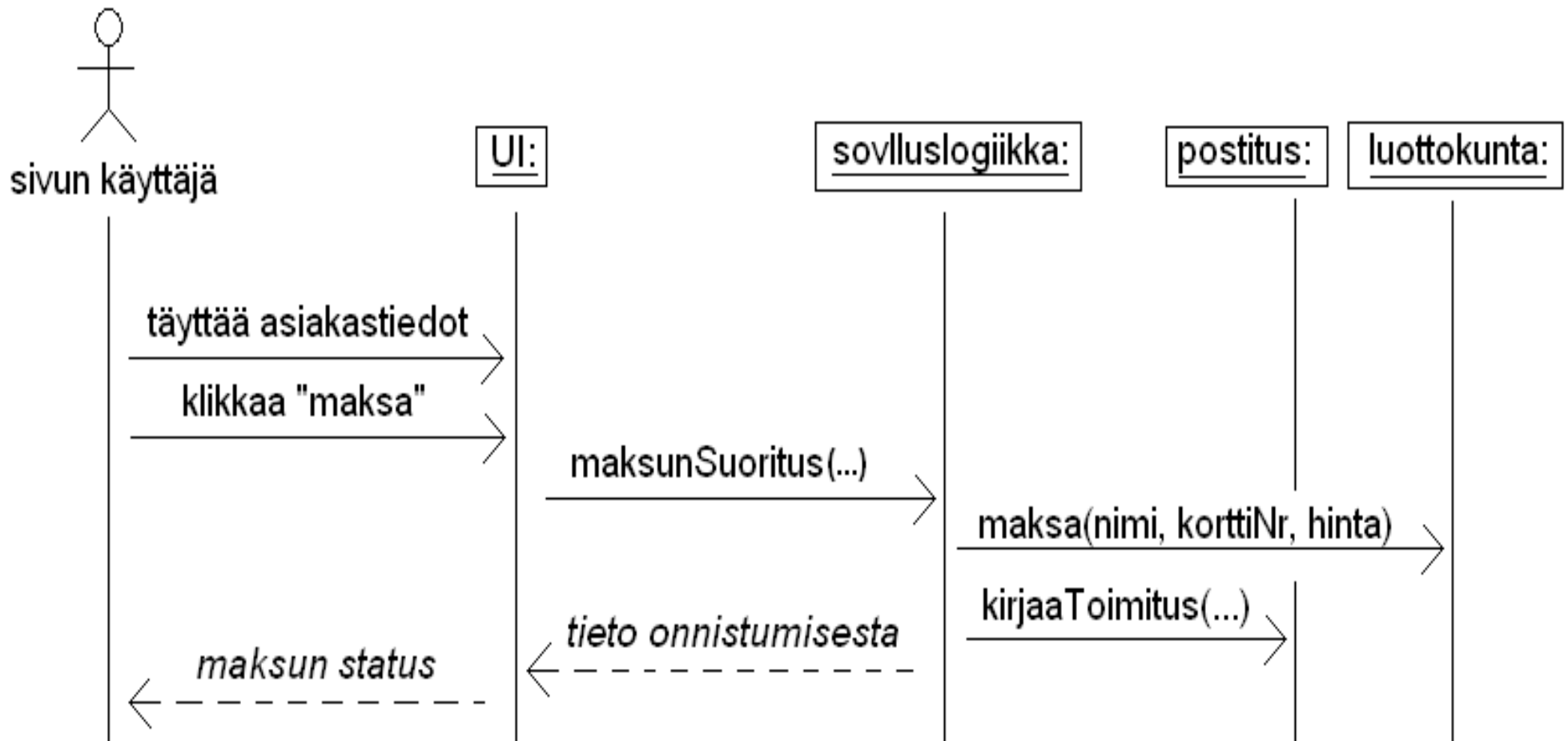
- Ostoskori-olioon lisättävä uusi ostos-olio joka vastaa ostettavaa tuotetta
- tai jos samaa tuotetta on jo korissa, päivitettävä korissa jo olevaa ostos-olioa

- ***suorita maksu***

- Suoritetaan maksu Luottokunnan rajapinnan avulla
- Jos maksu onnistuu
 - Ilmoitetaan ostoksen tiedot ja toimitusosoite postitusjärjestelmän verkkorajapinnalle
 - tyhjennetään ostoskori
 - Luodaan ostotapahtuma
 - Ostotapahtumaan liittyy ostoskorissa olevat tuotteet sekä asiakkaan osoitetiedot

- Seuraavalla sivulla vielä astetta tarkempi järjestelmätason sekvenssikaavio, jossa tuodaan esille ulkoisten järjestelmien kanssa tapahtuva kommunikointi

Käyttötapausten *suorita maksu* tarkennettu järjestelmätason sekvenssikaavio



- Kutsujen yhteydessä välitettäviä tietoja ei ole tilan puutteen takia merkitty täsmällisesti. Toisaalta se ei ole tässä vaiheessa edes tarpeen

Määrittelystä suunnittelun

- Järjestelmätason sekvenssikaaviot siis tuovat selkeästi esiin, mihin toimintoihin järjestelmän on kyettävä toteuttaakseen asiakkaan vaatimukset (jotka siis on kirjattu käyttötapauksina)
- Sekvenssikaavioista ilmi käyvien operaatioiden voi ajatella muodostavat *järjestelmän ulospäin näkyvän rajapinnan*
 - Kyseessä ei siis vielä varsinaisesti ole suunnittelutason asia
 - Nyt alkaa kuitenkin konkretisoitua, mitä järjestelmältä tarkalleen ottaen vaaditaan, eli mitä operaatiota järjestelmällä on ja mitä operaatioiden on tarkoitus saada aikaan
- Tässä vaiheessa siirrymme suunnitteluun
- Järjestelmän sovelluslogiikan operaatiot saattavat vielä tarkentua nimien ja parametrien osalta
 - tämä ei haittaa sillä on täysin ketterien menetelmien hengen mukaista, että astiat tarkentuvat ja muuttuvat sitä mukaa järjestelmän suunnittelu etenee

Ohjelmiston suunnittelu

- *Suunnitteluvaiheessa tarkoituksena on löytää sellaiset oliot, jotka pystyvät yhteistoiminnallaan toteuttamaan järjestelmältä vaadittavat operaatiot*
- Suunnittelu jakautuu karkeasti ottaen kahteen vaiheeseen:
 - Arkkitehtuurisuunnittelu
 - Olios suunnittelu
- Ensimmäinen vaihe on **arkkitehtuurisuunnittelu**, jonka aikana hahmotellaan järjestelmän rakenne karkeammalla tasolla
- Tämän jälkeen suoritetaan **oliosuunnittelu**, eli suunnitellaan oliot, jotka ottavat vastuulleen järjestelmältä vaaditun toiminnallisuuden toteuttamisen
 - Yksittäiset oliot eivät yleensä pysty toteuttamaan kovin paljoa järjestelmän toiminnallisuudesta
 - Erityisesti oliosuunnitteluvaiheessa tärkeäksi seikaksi nouseekin *olioiden välinen yhteistyö*, eli se vuorovaikutus, jolla oliot saavat aikaan halutun toiminnallisuuden

Määrittely- ja suunnittelutason luokkien yhteys

- Ennen kun lähdemme suunnitteluun, on syytä korostaa erästä seikkaa
- Määrittelyvaiheen luokkamallissa esiintyvät luokat edustavat vasta sovellusalueen yleisiä käsitteitä
 - Määrittelyvaiheessa luokille ei edes merkitä vielä mitään metodeja
- Kuten pian tulemme näkemään, monet määrittelyvaiheen luokkamallin luokat tulevat siirtymään myös suunnittelu- ja toteutustasolle
 - Osa luokista saattaa jäädä pois suunnitteluvaiheessa, osa muuttaa muotoaan ja tarkentuu
 - Suunnitteluvaiheessa saatetaan myös löytää uusia tarpeellisia kohdealueen käsitteitä
 - Suunnitteluvaiheessa ohjelmaan tulee lähes aina myös *teknisen tason luokkia*, eli luokkia, joilla ei ole suoraa vastinetta sovelluksen kohdealueen käsitteistössä
 - Teknisen tason luokkien tehtävänä on esim. toimia oliosäiliöinä ja sovelluksen ohjausolioina sekä toteuttaa käyttöliittymä ja huolehtia tietokantayhteyksistä

Ohjelmiston arkkitehtuuri

Ohjelmiston arkkitehtuuri

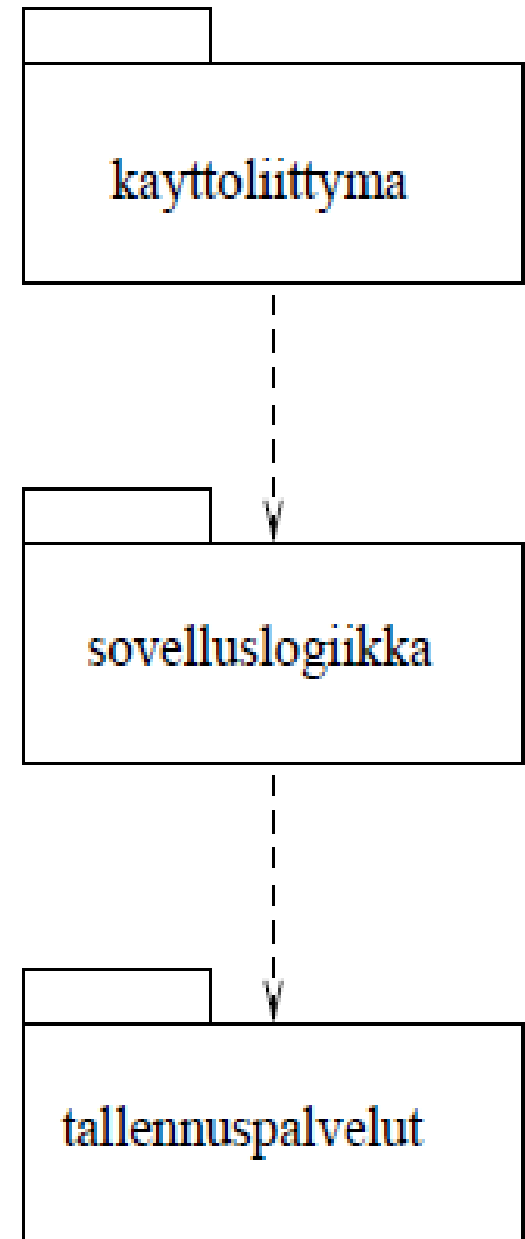
- Ohjelmiston *arkkitehtuurilla* (engl. software architecture) tarkoitetaan ohjelmiston korkean tason rakennetta
 - jakautumista erillisiin *komponentteihin*
 - komponenttien välisiä suhteita
- *Komponentilla* tarkoitetaan yleensä kokoelmaa *toisiinsa liittyviä olioita/luokkia*, jotka suorittavat ohjelmassa jotain tehtäväkokonaisuutta
 - esim. käyttöliittymän voitaisiin ajatella olevan yksi komponentti
- Ison komponentti voi muodostua useista *alikomponenteista*
 - Biershopin sovelluslogiikkakomponentti voisi sisältää komponentin, joka huolehtii sovelluksen alustamisesta ja yhden komponentin kutakin järjestelmän toimintokokonaisuutta varten
 - tai Biershopissa voisi olla omat komponentit ostosten tekoa ja varastotietojen ylläpitoa varten
 - ...

Ohjelmiston arkkitehtuuri

- Jos ajatellaan pelkkää ohjelman jakautumista komponenteiksi, puhutaan oikeastaan loogisesta arkkitehtuurista
- Looginen arkkitehtuuri ei ota kantaa siihen miten eri komponentit sijoitellaan, eli toimiiko esim. käyttöliittymä samassa koneessa kuin sovelluksen käyttämä tietokanta
- Ohjelmistoarkkitehtuurit on laaja aihe jota käsitellään nyt vain pintapuolisesti
 - Aiheesta on olemassa noin 4. vuotena suoritettava syventävien opintojen kurssi *Ohjelmistoarkkitehtuurit*
 - Asiaa käsitellään myös 2. vuoden kurssilla *Ohjelmistotuotanto*
- UML:ssa on muutama kaaviotyyppi jotka sopivat arkkitehtuurin kuvaamiseen
 - *Komponenttikaaviota* ja *sijoittelukaaviota* emme nyt käsittele
 - Komponenttikaavio on erittäin käyttökelpoinen kaaviotyyppi, mutta silti jätämme sen myöhemmille kursseille
 - Nyt käytämme *pakkauskaaviota* (engl. package diagram)

Pakkauskaavio

- Kuvassa karkea hahmotelma Biershopin arkkitehtuurista
- Näemme, että järjestelmä on jakautunut kolmeen komponenttiin
 - Käyttöliittymä
 - Sovelluslogiikka
 - Tallennuspalvelut
- Jokainen komponentti on kuvattu omana pakkauksena, eli isona laatikkona, jonka vasempaan ylälaitaan liittyy pieni laatikko
- Laatikoiden välillä on *riippuvuuksia*
 - Käyttöliittymä riippuu sovelluslogiikasta
 - Sovelluslogiikka riippuu tallennuspalveluista
- Järjestelmä perustuu **kerrosarkkitehtuuriin** (engl. layered architecture)

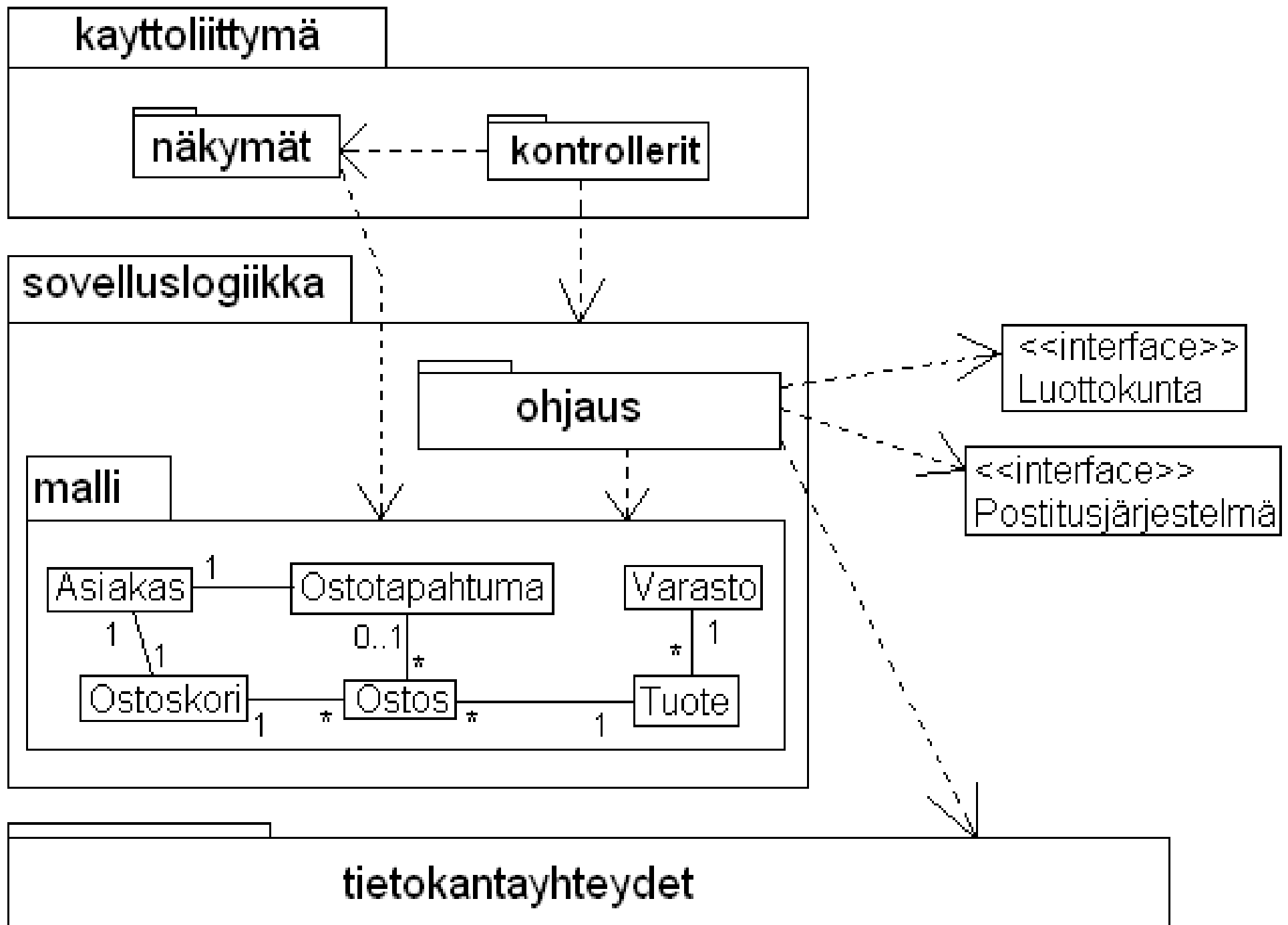


Kerrosarkkitehtuuri

- Järjestelmän rakenne perustuu siis **kerrosarkkitehtuuriin**
 - Kerrosarkkitehtuuri yksi hyvin tunnettu *arkkitehtuurimalli* (engl. architecture pattern), eli periaate, jonka mukaan tietynlaisia ohjelmia kannattaa pyrkiä rakentamaan
 - Olemassa myös muita arkkitehtuurimalleja, joita ei nyt kuitenkaan käsitellä
- Kerros on kokoelma toisiinsa liittyviä olioita tai alikomponentteja, jotka muodostavat esim. toiminnallisuuden suhteen loogisen kokonaisuuden ohjelmistosta
- *Kerrosarkkitehtuurissa on pyrkimyksenä järjestellä komponentit siten, että ylempänä oleva kerros käyttää ainoastaan alempana olevien kerroksien tarjoamia palveluita*
 - Ylimpänä kerroksista on käyttöliittymäkerros
 - sen alapuolella sovelluslogiikka
 - alimpana tallennuspalveluiden kerros, eli esim. tietokanta, jonne sovelluksen olioita voidaan tarvittaessa tallentaa.
- Palaamme kerrosarkkitehtuurin hyötyihin pian, ensin hieman lisää pakkauskaavioista

Pakkauskaavio

- Pakkauskaaviossa yksi komponentti kuvataan pakkaussymbolilla
 - Pakkauksen nimi joko keskellä symbolia tai ylänurkan laatikossa
- Pakkausten välillä olevat *riippuvuudet* ilmaistaan *katkoviivanuolena*, joka suuntautuu pakkaukseen, johon riippuvuus kohdistuu
- Kerrosarkkitehtuurissa siis pyrkimyksenä, että riippuvuuksia on ainoastaan alapuolella oleviin kerroksiin: Biershopin *käyttöliittymäkerros riippuu sovelluslogiikkakerroksesta mutta ei päinvastoin*
 - Riippuvuus tarkoittaa käytännössä sitä, että *käyttöliittymän oliot kutsuvat sovelluslogiikan olioiden metodeja*
 - Sovelluslogiikkakerros taas on riippuvainen tallennuspalvelukerroksesta
- Pakkauksen sisältö on mahdollista piirtää pakkaussymbolin sisään kuten seuraavalla kalvolla olevassa Biershopin tarkennetussa arkkitehtuurikuvauksessa
 - Pakkauksen sisällä voi olla alipakkauksia tai luokkia
- Riippuvuudet voivat olla myös alipakkausten välisiä
- Palaamme seuraavalla sivulla olevaan pakkauskaavioon tarkemmin myöhemmin



Kerrosarkkitehtuurin etuja

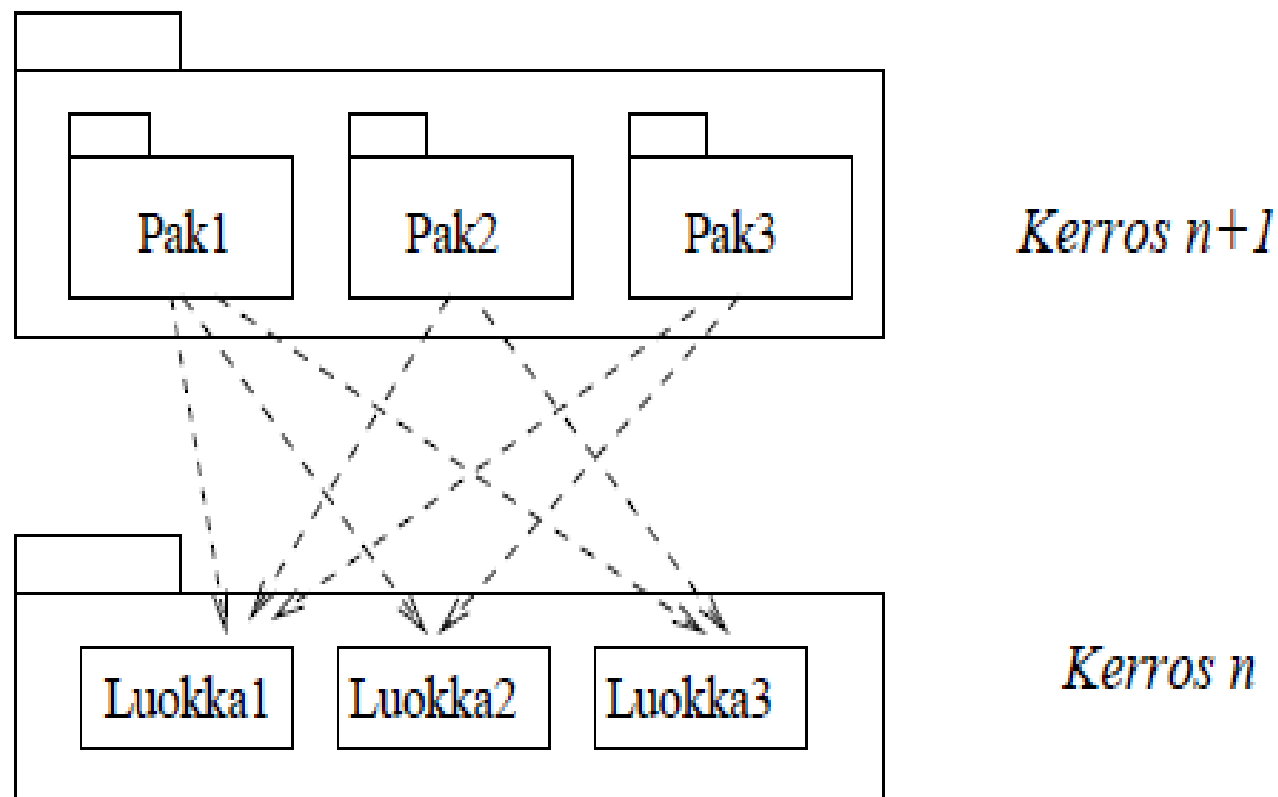
- Kerroksittaisuus *helpottaa ylläpitoa*
 - Kerroksen sisältöä voi muuttaa vapaasti jos sen palvelurajapinta eli muille kerroksille näkyvät osat säilyvät muuttumattomina
 - Sama pätee tietysti mihin tahansa komponenttiin
- Jos kerroksen palvelurajapintaan tehdään muutoksia, aiheuttavat muutokset *ylläpitotoimenpiteitä ainoastaan ylemmän kerroksen* riippuvuuksia omaavin osiin
 - Esim. käyttöliittymän muutokset eivät vaikuta sovelluslogiikkaan tai tallennuspalveluihin
- Sovelluslogiikan riippumattomuus käyttöliittymästä helpottaa ohjelman siirtämistä uusille alustoille
- Alimpien kerroksien palveluja, kuten esim. tallennuspalvelukerrosta voidaan ehkä uusiokäyttää myös muissa sovelluksissa
- *Ylemmät kerrokset* voivat toimia *korkeammalla abstraktiotasolla*
 - Esim. hyvin tehty tallennuspalvelukerros kätkee tietokannan käsittelyn muilta kerroksilta: sovelluslogiikan tasolla voidaan ajatella kaikki olioina
 - Kaikkien ohjelmoijien ei tarvitse ymmärtää kaikkia detaljeja, osa voi keskittyä tietokantaan, osa käyttöliittymiin, osa sovelluslogiikkaan

Ei pelkkiä kerroksia...

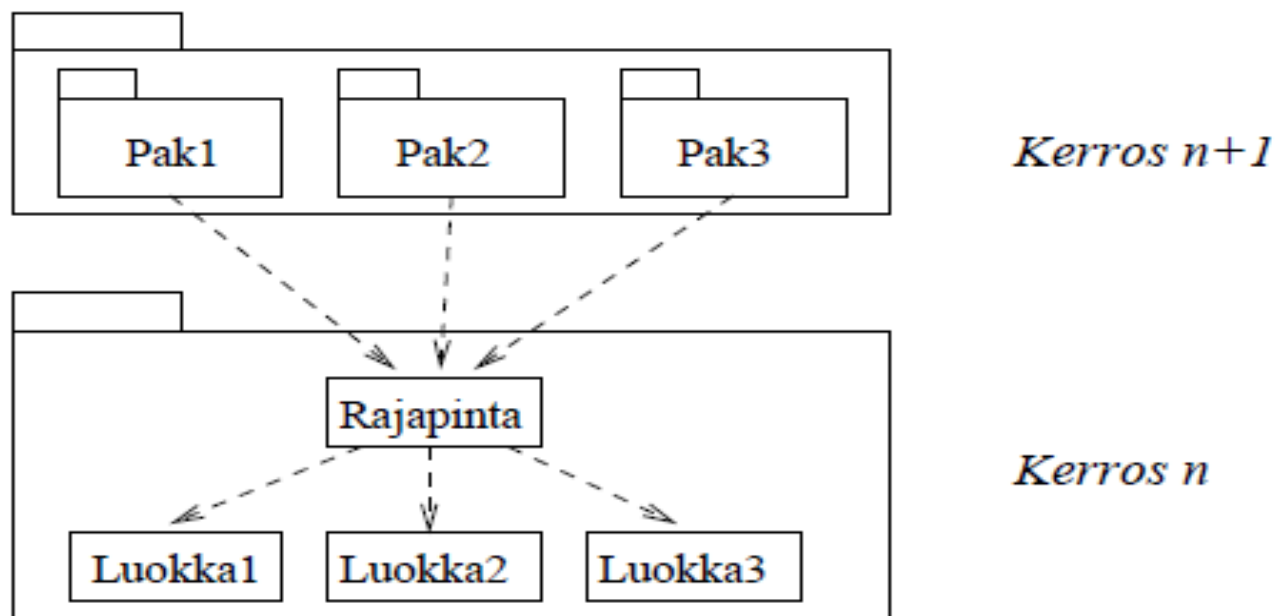
- Myös kerrosten sisällä ohjelman loogisesti toisiinsa liittyvät komponentit kannattaa ryhmitellä omiksi kokonaisuuksiksi joka voidaan UML:ssa kuvata pakkauksena
- Yksittäisistä komponenteista kannattaa tehdä mahdollisimman *yhtenäisiä* toiminnallisuudeltaan
 - eli sellaisia, joiden osat kytkeytyvät tiiviisti toisiinsa ja palvelevat ainoastaan yhtä selkeästi eroteltua tehtäväkokonaisuutta
- Samalla pyrkimyksenä on, että erilliset komponentit ovat mahdollisimman *löyhästi kytkettyjä* (engl. loosely coupled) toisiinsa
 - komponenttien välisiä riippuvuuksia pyritään minimoimaan
- Ohjelman jakautuminen mahdollisimman riippumattomiin komponentteihin eristää koodiin ja suunnitelmaan tehtävien muutosten vaikutukset mahdollisimman pienelle alueelle, eli ainoastaan riippuvuuden omaaviin komponentteihin
- Tämä helpottaa ohjelman testausta sekä ylläpitoa ja tekee sen laajentamisen helpommaksi
- Selkeä jakautuminen komponentteihin myös helpottaa työn jakamista suunnittelu- ja ohjelmointivaiheessa

Kerroksellisuus ei riitä

- Pelkkä kerroksittaisuus ei tee ohjelman arkkitehtuurista automaattisesti hyvää.
- Alla tilanne, missä kerroksen $n+1$ kolmella alipaketilla on kullakin paljon riippuvuuksia kerroksen n sisäisiin komponenttiin
- Esim. muutos kerroksen n luokkaan 1 aiheuttaa nyt muutoksen hyvin moneen ylemmän kerroksen pakkaukseen



- **Kerrosten välille** kannattaa määritellä selkeä **rajapinta**
- Yksi tapa toteuttaa rajapinta on luoda kerroksen sisälle erillinen *rajapintaolio*, jonka kautta ulkoiset yhteydet tapahtuvat
 - Tätä periaatetta sanotaan *fasaadiksi* (engl. facade pattern)
 - Huom: rajapinnalla ei nyt tarkoiteta pelkästään Javan rajapintaa, kyseessä voi olla myös usean rajapinnan kokonaisuus jonka ylempi kerros näkee
- Alla luotu rajapintaolio kerrokselle n . Kommunikointi kerroksen kanssa tapahtuu rajapintaolion kautta
 - ylemmän kerroksen riippuvuudet kohdistuvat rajapintaolioon
 - muutos esim. luokkaan 1 ei vaikuta kerroksen $n+1$ komponentteihin
 - ainoat muutokset on tehtävä rajapintaolion sisäiseen toteutukseen



Käyttöliittymän ja sovelluslogiikan erottaminen

- Kerrosarkkitehtuurin ylimpänä kerroksena on yleensä käyttöliittymä
- Pidetään järkevänä, että **ohjelman sovelluslogiikka on täysin erotettu käyttöliittymästä**
 - Asia tietysti riippuu myös sovelluksen luonteesta ja oletetusta käyttöajasta
 - Sovelluslogiikan erottaminen lisää koodin määrää, joten jos kyseessä ”kertakäyttösovellus”, ei ylimääräinen vaiva ehkä kannata
- Käytännössä tämä tarkoittaa kahta asiaa:
 - **Sovelluksen palveluja toteuttavilla olioilla** (mitkä suunnitellaan seuraavassa luvussa) **ei ole suoraan yhteyttä käyttöliittymän olioihin**, joita ovat esim. Java-ohjelmissa Swing-komponentit, kuten menut, painikkeet ja tekstikentät tai verkossa toimivissa ohjelmissa ns. Servletit
 - Eli sovelluslogiikan oliot eivät esim. suoraan kirjoita mitään ruudulle
 - **Käyttöliittymän toteuttavat oliot eivät sisällä ollenkaan ohjelman sovelluslogiikkaa**
 - Käyttöliittymäoliot ainoastaan piirtävät käyttöliittymäkomponentit ruudulle, välittävät käyttäjän komennot eteenpäin sovelluslogiikalle ja heijastavat sovellusolioiden tilaa käyttäjille

Käyttöliittymän ja sovelluslogiikan erottaminen

- Käytännössä erottelu tehdään liittämällä käyttöliittymän ja sovellusalueen olioiden väliin erillisiä komponentteja, jotka koordinoivat käyttäjän komentojen aiheuttamien toimenpiteiden suoritusta sovelluslogiikassa
- Erottelun pohjana on Ivar Jacosonin kehittämä idea oliotyyppien jaoittelusta kolmeen osaan, rajapintaolioihin (boundary), ohjausolioihin (control) ja sisältöolioihin (entity)
- Käyttöliittymän (eli rajapintaolioiden) ja sovelluslogiikan (eli sisältöolioiden) yhdistävät **ohjausoliot**
- *Käyttöliittymä ei siis itse tee mitään sovelluslogiikan kannalta oleellisia toimivia, vaan ainoastaan välittää käyttäjien komentoja ohjausolioille, jotka huolehtivat sovelluslogiikan olioiden manipuloimisesta*
- Huom: idea ohjausolioista on hiukan sukua ns. *MVC-periaatteelle*, tarkalleen ottaen kyse ei kuitenkaan ole täysin samasta asiasta

Hieman oliosuunnittelu

Oliosuunnittelu

- Ohjelman on siis toteutettava käyttötapauksista johdetut operaatiot toimiakseen vaatimustensa mukaisesti
- Ohjelmalta vaadittavien operaatioiden voidaan ajatella olevan *ohjelman vastuita* (engl. responsibilities).
 - hoitamalla vastuunsa ohjelma toimii kuten sen odotetaan toimivan
- Ohjelma toteuttaa vastuunsa olioiden yhteistyön avulla
 - **Haasteena oliosuunnittelussa siis on löytää sopivat oliot, joille ohjelman vastuut jaetaan**
- Tyypillisesti mikään yksittäisen olio ei toteuta yhtä ohjelman vastuuta itse, vaan *jakaa vastuun pienemmiksi alivastuiksi ja delegoi alivastuiden hoitamisen muille olioille*
- **Oliosuunnittelun lähtökohdaksi otetaan yleensä määrittelyvaiheen luokkamalli**
 - todennäköisesti Biershapiin tulee suunnittelu/toteutustasolle mukaan ainakin luokat Tuote, Ostos, Ostotkori, Asiakas, Varasto ja Ostotapahtuma

Oliosuunnittelun periaatteita

- Oliosuunnittelua tehdessä kannattaa pitää mielessään hyvän oliosuunnittelun periaatteet, kerrataan ne vielä seuraavassa
 - Itseasiassa hyvällä ohjelmistosuunnittelijalla nämä periaatteet pitää olla jo selkärangassa
- Olemme jo maininneet kurssilla kolme tärkeää periaatetta
 - **Single responsibility**
 - Oliosta kannattaa tehdä pieniä ja selkeitä, hyvin yhden asian osaavia oliota
 - **Favour composition over inheritance**
 - Älä liikakäytä perintää
 - Koosta mielummin toiminnallisuus useasta pienestä yhdessä toimivasta yhden vastuun olioista jotka hoitavat vastuunsa jakamalla sen osavastuusiin joiden hoitaminen delegoidaan muille olioille

Oliosuunnittelun periaatteita

- Kolmas jo mainittu periaate oli
 - **Program to an interface, not to an Implementation**
 - Tee luokat mahdollisimman riippumattomiksi toisistaan
 - Tunne vain rajapinta tai abstrakti luokka
- Jo kauan ennen olio-ohjelmoinnin keksimistä on korostettu järjestelmän eri komponenttien **riippuvuuksien vähäisyyden** (engl. low coupling) periaatetta
 - Järjestelmän erilaisten komponenttien välisten riippuvuuksien minimointia perusteltiin jo kerrosarkkitehtuurin yhteydessä
 - Erityisesti riippuvuuksia konkreettisiin luokkiin kannattaa välttää (periaate program to interfaces, not to concrete implementations sanoo juuri tämän)
 - Riippuvuuksien vähentämistä ei kuitenkaan saa tehdä single responsibility -periaatetta rikkoen
- Muutamia muitakin periaatteita on olemassa
- Kaikkien oliosuunnittelun periaatteiden motivaationa on ohjelman ylläpidettävyyden, muokattavuuden ja testattavuuden maksimoiminen

Uusien luokkien mukaanottaminen

- Jos suunnittelutasolle ei oteta muita luokkia kuin määritelyvaiheen luokkamallissa olevat luokat, joudutaan helposti huonoihin ratkaisuihin
 - Seurauksena olioiden sekavat vastuut ja liialliset riippuvuudet ja näinollen mm. single responsibility -periaate rikkoutuu
- Tällaisissa tilanteissa otetaan mukaan uusia, ”keinotekoisia” luokkia, joita vastaavia käsitteitä ei välttämättä sovelluksen kohdealueella ole
- Uudet mukaan tuotavat luokat ovat usein *teknisen tason luokkia*, joilla voi olla monia erilaisia käyttötarkoituksia, esim.
 - Käyttöliittymän toteutus
 - Tietokantayhteyksien hoitaminen
 - Sovellusolioiden yhteyksien toteuttaminen
 - Osajärjestelmien piilottaminen (fasadioliot)
 - Sovelluksen toiminnan ohjaus ja koordinointi (ks. seuraava sivu)
 - Abstraktit yliluokat ja rajapinnat

Käyttöliittymän ja sovelluslogiikan erottaminen ohjausolioiden avulla

- Kuten jo mainittiin, käyttöliittymä ja sovelluslogiikka on hyvä erottaa, eli
 - **Käyttöliittymän toteuttaviin olioihin ei sisällytetä ollenkaan ohjelman sovelluslogiikkaa**
 - Käyttötapaukset toteuttavat operaatiot suoritetaan kokonaisuudessaan sovelluslogiikan puolella
 - Käyttöliittymä ainoastaan kutsuu näitä operaatioita kun käyttäjä suorittaa käyttötapaukseen liittyvää toiminnallisuutta
- Yksi tapa tehdä erottelu on lisätä käyttöliittymän ja varsinaisten sovellusolioiden väliin **ohjausolioita**, jotka ottavat vastaan käyttöliittymästä tulevat operaatiokutsut ja kutsuvat edelleen sovelluslogiikan olioiden metodeja
- Ohjausolio voi olla ohjelman **kaikkien operaatioiden yhteinen** ohjausolio tai vaihtoehtoisesti voidaan käyttää **käyttötapauskohtaisia** ohjausolioita
 - Välimuodotkin ovat mahdollisia eli joillain käyttötapauksella voi olla oma ohjausolio ja jotkut taas käyttävät yhteistä ohjausolioa
- Liian monesta asiasta huolehtivat ohjausoliot kuitenkin rikkovat single responsibility -periaatetta ja vaikeuttavat ohjelman ylläpidettävyyttä ja testattavuutta

Oliosuunnittelun vaikeus

- Todettakoon edellisiin kalvoihin liittyen, että sopivien teknisten apuluokkien keksiminen on äärimmäisen haastavaa
 - Kokemus, luovuus ja tieto auttavat
- Monissa *suunnittelumalleista* (engl. design patterns) on kyse juuri sopivien teknisten ratkaisujen ja abstraktioiden mukaantuomisesta
 - Luennoilla ja materiaalissa olemme törmänneet jo pariin suunnittelumalliin (mm. fasaadi, komposiitti)
- Valitettavasti suunnittelumalleja ja muuta haasteellista ja mielenkiintoisia oliosuunnitteluun liittyvää ei tällä kurssilla juuri ehditä käsittelemään
 - Asiaan tutustutaan tarkemmin toisen vuoden kurssilla Ohjelmistotuotanto
 - Aiheesta löytyy runsaasti kirjallisuutta
 - Robert Martin: Agile and iterative development
 - Larman: Applying UML and Patterns
 - Freeman et. All.: Head first design patterns
 - Gamma et all.: Design patterns

Suunnittelun eteneminen: käyttötapaus kerrallaan

- Yksi tapa tehdä suunnittelua on edetä käyttötapauksittain
 - Otetaan yksi käyttötapaus kerrallaan tarkasteluun
 - Suunnitellaan oliot tai mukautetaan jo suunniteltujen olioiden vastuita ja yhteistyötä siten, että tarkastelussa olevan käyttötapauksen tarvitsemat operaatiot saadaan toteutetuksi
 - Usein käy niin, että uuden toiminnallisuuden lisääminen aiheuttaa jo olemassa olevaan suunnitelmaan pieniä muutoksia
- Aloitamme nyt olutkaupan oliosuunnittelun käyttötapauksesta **Lisää ostos koriin**
- Koska oliosuunnittelussa on kyse olioiden välisestä yhteistyöstä, on **oliosuunnittelun yksi tärkeimmistä työvälineistä sekvenssikaavio**
 - Luokkakaavioon merkittyjen metodinimien avulla oliosuunnittelua ei kannata tehdä
- Todellisuudessa suunnittelu ei yleensä etene ollenkaan näin suoraviivaisesti kuten nämä kalvot antavat ymmärtää
- Oikeastaan nämä kalvot vaan näyttävät suunnittelun lopputuloksen. Luentomonisteesta löytyvä Kirjasto-esimerkki valottaa suunnitteluprosessia hieman tarkemmin

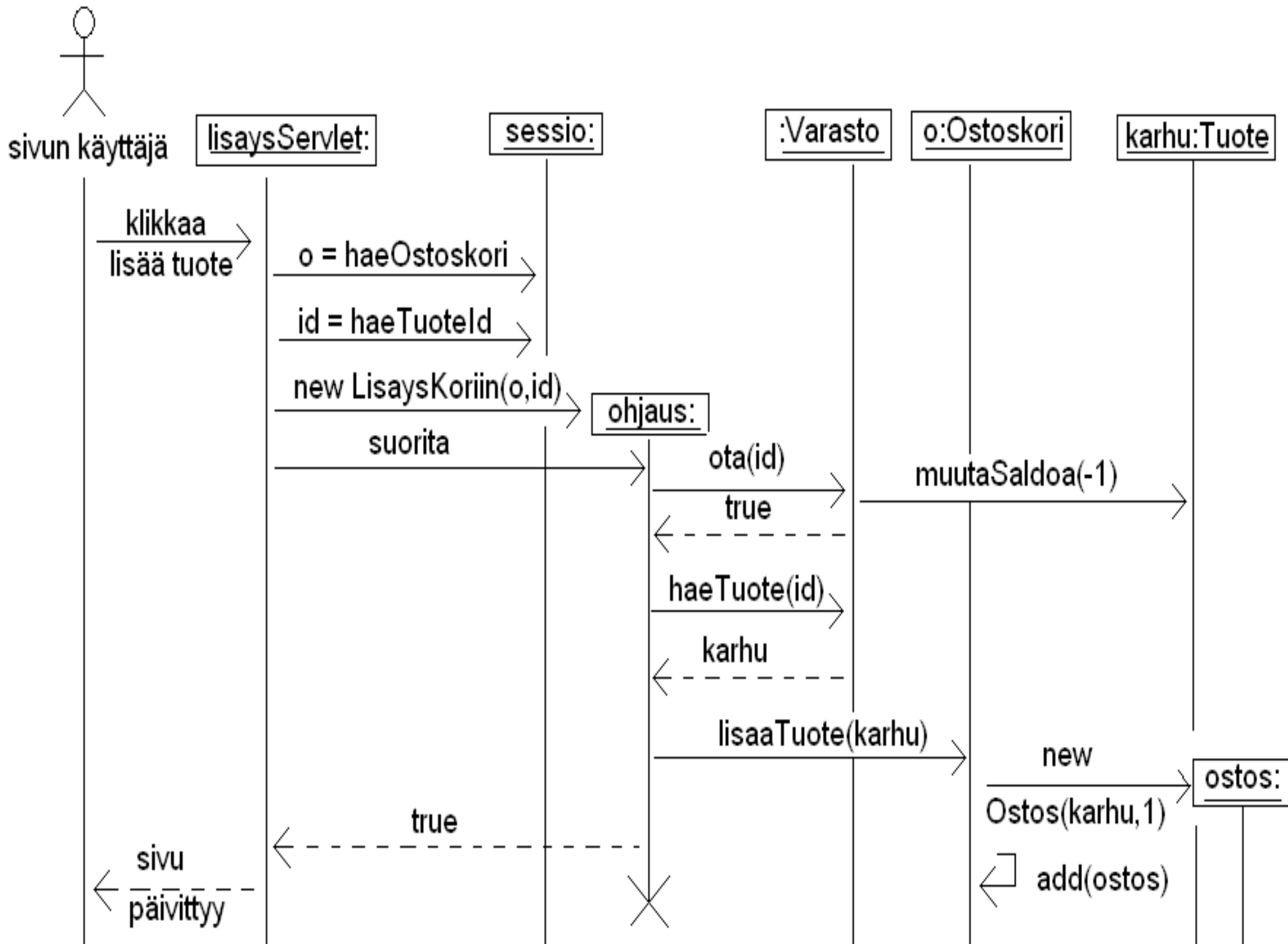
**Käyttötapausten *Lisää ostos koriin*
toiminnallisuuden suunnittelu**

Lisää ostos koriin toiminnallisuuden suunnittelu

- Lähtökohdaksi otetaan kalvolla 9 esitetty määrittelyvaiheen luokkamalli
 - Katso tätä ja seuraavaa kalvoa lukiessasi kahden kalvon päästä löytyvää toiminnallisuuden kuvaavaa sekvenssikaaviota
- Kalvolla 16 todettiin että käyttötapauksen suorituksen vastuulla on seuraavien toimenpiteiden tekeminen
 - Ostoskori-olioon lisättävä uusi ostos-olio, joka vastaa ostettavaa tuotetta
 - tai jos samaa tuotetta on jo korissa, päivitettävä korissa jo olevaa ostos-olioa
- Määrittelemme käyttötapausta varten ohjausolioluokan LisaaKoriin
 - Kun käyttäjä lisää tuotteen ostoskoriin, käyttöliittymä luo ohjausolion sekä käynnistää sen
- Ohjausolio tulee suorittamaan pääosan käyttötapauksen vastuista
 - Teemme ohjausoliosta kertakäyttöisen, eli jokaisen ostoksen lisäyksen koriin hoitaa oma ohjausolionsa
- Päätämme myös lisätä käyttötapaukselle uuden vastuun
 - Kun ostos lisätään koriin, tulee tuotteen saldoa vähentää yhdellä
 - Näin varastossa olevien tuotteiden saldo kuvaa koko ajan myymättömien ja korissa olemattomien tuotteiden määrää

Lisää ostos koriin toiminnallisuuden suunnittelu

- Toteutustekniikka vaikuttaa jossain määrin ohjelmiston suunnitteluun
- Biershop on toteutettu Java-servlet-tekniikalla
 - Servleteillä toteutetussa web-sovelluksessa jokaiseen käyttäjään liittyy *sessio-olio*, jonka avulla käyttöliittymä saa tietoonsa käyttäjän ostoskorin
 - Käyttöliittymä saa tietoonsa sessiolta (tosiasiassa muualta, mutta yksinkertaistetaan hieman) myös sen tuotteen id:n jonka käyttäjä haluaa lisätä ostoskoriin
- Käyttöliittymä antaa luotavalle ohjausolion ostoskorin ja lisättävää tuotetta vastaavan id:n ja käynnistää ohjausolion kutsumalla sen metodia *suorita*
- Ohjausolio saa id:tä vastaavan tuotteen selville Varasto-oliolta
 - Varaston vastuulla on hoitaa tuotteisiin liittyviä asioita
- Ohjausolio ottaa varastosta tuotteen, tämä saa aikaan tuotteen saldon pienenemisen
- Ohjausolio pyytää Ostoskoria lisäämään tuotteen
- Ostoskori luo tuotetta vastaavan Ostos-olion
 - Jos tuote olisi jo korissa, päivitettäisiin olemassaolevaa ostos-olioa
- Ohjausolio palauttaa käyttöliittymälle tiedon onnistumisesta ja käyttöliittymä päivittää käyttäjän sivunäkymän. Ohjausolio tuhoutuu lopussa



Huomioita

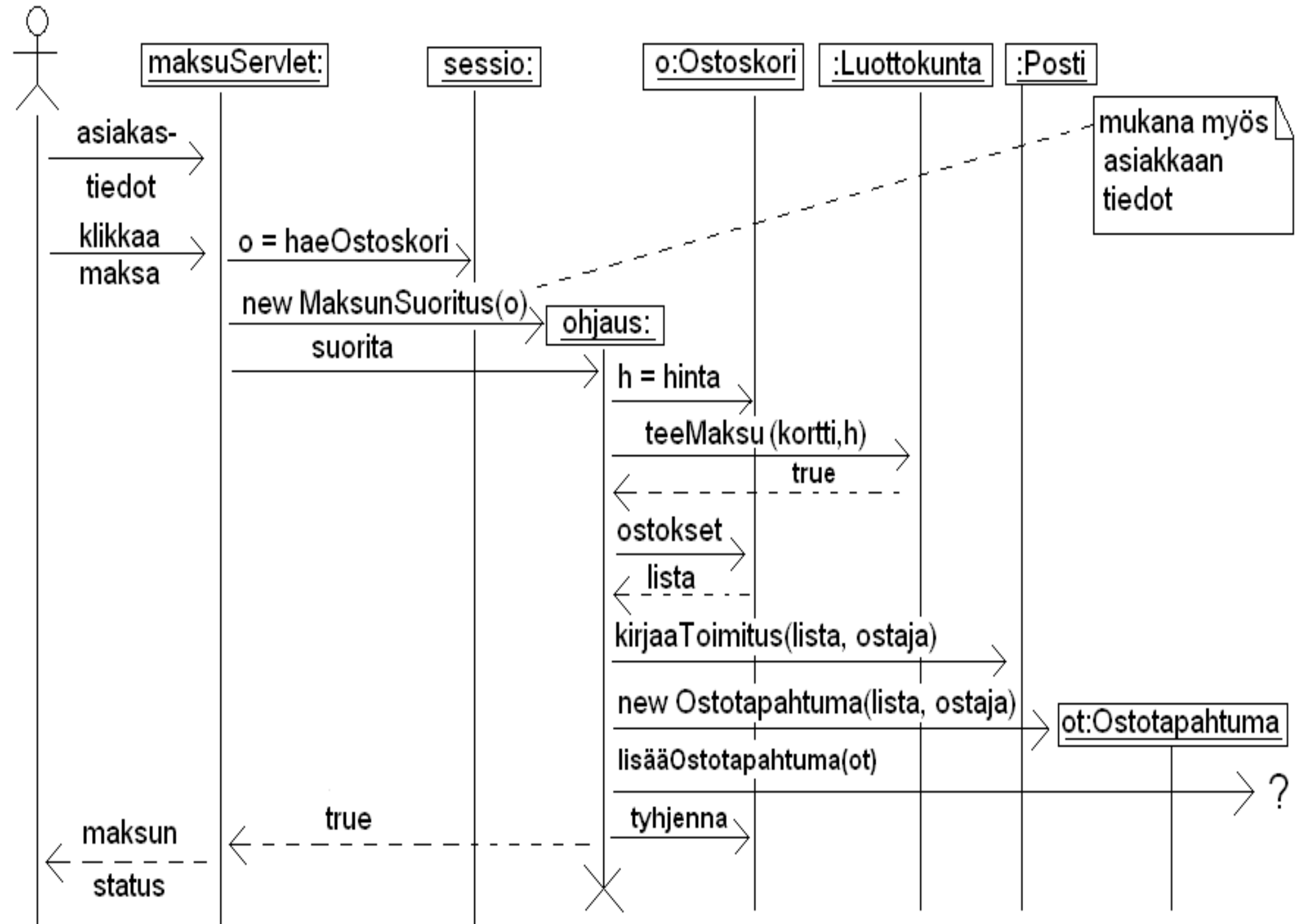
- Edellisen kalvon sekvenssikaavio kuvaa käyttötapauksen suorituksen aikana tapahtuvan toiminnallisuuden
- Alussa siis käyttöliittymä eli `lisaysServlet` selvittää `Sessio`-oliolta käyttäjän ostoskorin ja lisättävän tuotteen tunnisteen
 - Jos käytössä olisi jokin muu käyttöliittymätoteutustekniikka esim. Java Swing, olisi tämä vaihe hieman erilainen
- Ohjausolio on täysin riippumaton käytettävästä käyttöliittymätekniikasta
- Ohjausolion käytön ansiosta toiminnallisuutta on helppo testata (esim. JUnit-testien avulla) täysin käyttöliittymästä irrallaan
- Osan vastuistaan ohjausolio delegoi:
 - Varasto-olion vastuulla varastosaldon pienennys ja oikean Tuote-olion etsiminen varastosta
 - Ostoskori-olion vastuulla sen sisäisten Ostos-olioiden luonti
- Ohjausolio ei tee nyt liikaa asioita vaan lähinnä koordinoi käyttötapausta vastaavan toiminnallisuuden suorittamista
 - Tämä helpottaa varaston ja ostoskorin testausta

**Käyttötapauksen *Suorita maksu*
toiminnallisuuden suunnitleminen**

Suorita maksu toiminnallisuuden suunnittelu

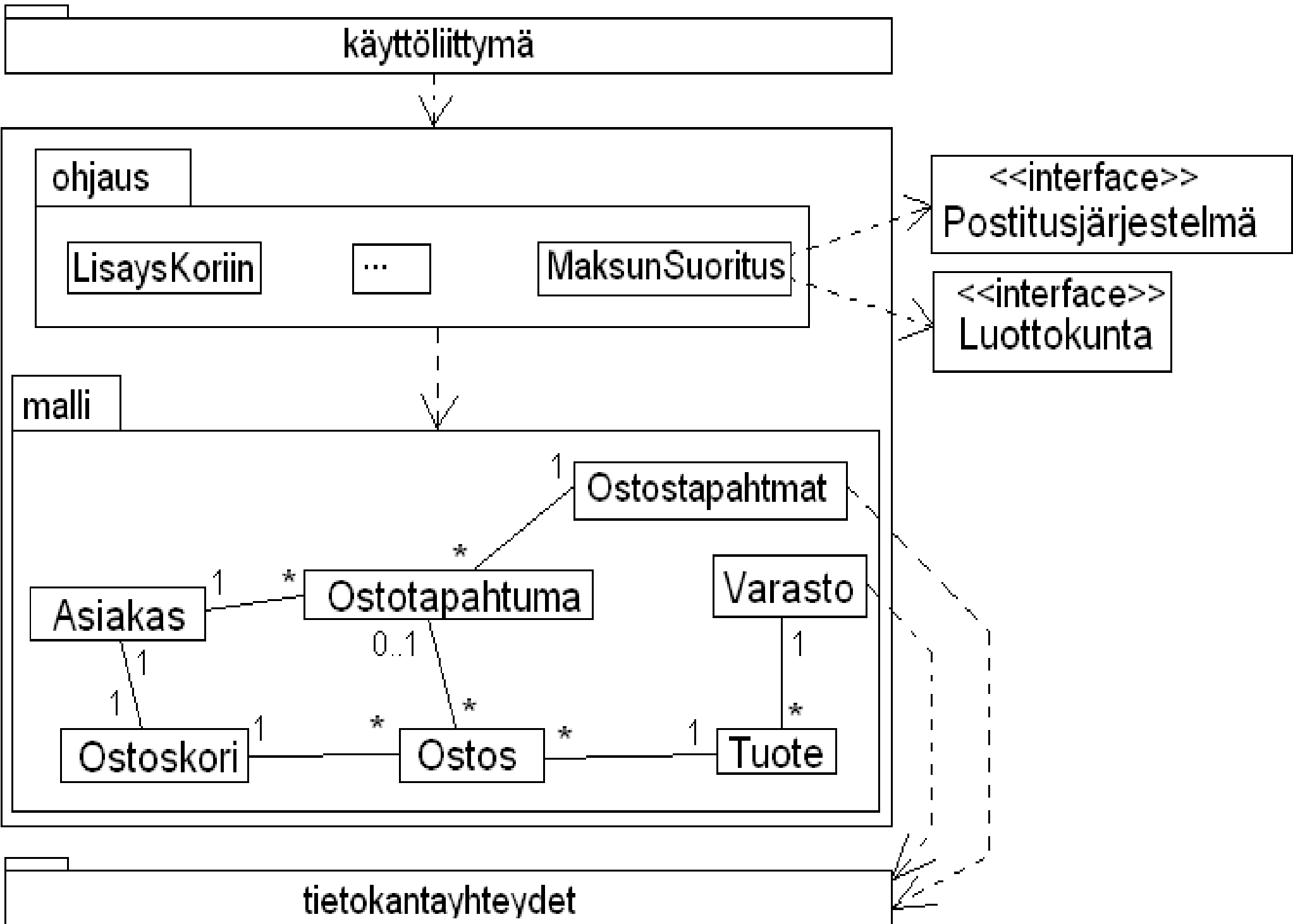
- Käyttöliittymä hakee asiakkaan ostoskorin sessio-oliolta ja luo ohjausolion jolle annetaan konstruktorissa ostoskori sekä asiakkaan web-sivulle täyttämät tiedot, sekä käynnistää ohjausolion
- Ohjausolio kysyy ostoskorilta sen hintaa
 - Ostoskorin vastuulla on siis tietää sisältämiensä ostosten hinta
- Ohjausolio veloittaa asiakkaan luottokortilta ostosten hinnan kutsumalla luottokunnan rajapintaa
- Ostoskorilta kysymällä saadaan lista ostoksista
- Ostosten lista yhdessä asiakkaan osoitetietojen kanssa lähetetään postitusjärjestelmän rajapinnalle
- Ohjausolio luo Ostostapahtuma-olion joka sisältää ostosten listan sekä asiakkaan tiedot
- Huomataan, että mallissa ei ole sopivaa luokkaa joka vastaisi Ostostapahtuma-olioiden käsittelystä
 - Päätetään lisätä luokka jonka oliolle luotu Ostostapahtuma-olio annetaan, uusi olio on merkattu sekvenssikaaviossa kysymysmerkillä
- Lopussa ohjausolio tyhjentää ostoskorin

Käyttötapausten *suorita maksu* suunnittelu



Suunnittelutason luokkakaavio käyttötapauksen toiminnallisuuden suunnittelun jälkeen

- Otimme pohjaksi kalvon 9 määrittelyvaiheen luokkakaavion
- Huomamme, että määrittelyvaiheen luokalla Kauppa ei oikeastaan ole käyttöä
- Olemme lisänneet Ohjausolioita varten luokat LisaysKoriin ja MaksunSuoritus
 - Näiden vastuulla on käyttötapaukseen liittyvän toiminnallisuuden suoritus
- Lisäsimme myös luokan Ostostapahtumat
 - Luokan ainoan olion vastuulla on huolehtia yksittäisistä Ostostapahtuma-olioista
- Varaston rooli on selkeytynyt:
 - Varasto-olion (joita järjestelmässä on yksi) rooli on huolehtia tuotteista
- Varasto ja Ostostapahtumat käyttävät Tietokantayhteydet-pakkauksen tarjoamia palveluita tallettamaan huolehtimansa oliot tietokantaan
- Seuraavalla sivulla suunnitteluvaiheen osittainen luokkakaavio



Huomioita oliosuunnittelusta

- Korostettakoon vielä toistamiseen: koska oliosuunnittelussa on kyse olioiden välisestä yhteistyöstä, on **oliosuunnittelun yksi tärkeimmistä työvälineistä sekvenssikaavio**
 - Luokkakaavioon merkittyjen metodinimien avulla oliosuunnittelua ei kannata tehdä
- Todellisuudessa suunnittelu ei etene näin suoraviivaisesti
- Suunnittelu ja toteutus voivat myös edetä osittain rinnakkain, erityisesti jos käytetään TDD:tä eli testivetoista kehitystä (josta kohta hieman lisää)
- On hyvin tyypillistä että suunniteltuja ratkaisuja joudutaan muuttamaan (eli ohjelman sisäistä rakennetta refaktoroimaan)
- Koska oliosuunnittelu ei ole suoraviivainen prosessi, ei kaavioita kannata tehdä kaavioeditorilla
- Ohjelmistosuunnittelu on erittäin haastava aihe, tässä olemme tehneet vaan pienen pintaraapausun aihepiiriin

**Muutama olionsuunnitteluun ja toteutukseen
liittyvä seikka**

Kertaus: oliosuunnittelun periaatteita

- **Single responsibility**
 - Jokaisella luokalla vain yksi selkeä vastuu
- **Favour composition over inheritance**
 - Älä liiakäytä perintää vaan suosi yhteistoiminnallisia oliota
- **Program to an interface, not to an Implementation**
 - Tee luokat mahdollisimman riippumattomiksi toisistaan
 - Tunne vain rajapinta
- **Riippuvuuksien minimointi**
 - Älä tee spagettikoodia jossa kaikki oliot tuntevat toisensa
- ”ikiaikaisia periaatteita”, motivaationa ohjelman muokattavuuden, uusiokäytettävyyden ja testattavuuden parantaminen
- Huonoa oliosuunnittelua on verrattu *velan* (engl. design debt tai technical debt) ottamiseen
 - Piittaamattomalla ja laiskalla ohjelmoinnilla/suunnittelulla saadaan ehkä nopeasti aikaan jotain, mutta hätäinen ratkaisu tullaan maksamaan korkoineen takaisin myöhemmin kun ohjelmaa on tarkoitus laajentaa tai muuttaa

Oliosuunnittelun kovia nimiä

- Oliosuunnittelun periaatteet siis ikiaikaisia, periaatteita ovat systematoisoineet mm. seuraavat henkilöt



Erich Gamma



Robert "uncle bob" Martin



Martin Fowler



Kent Beck

Koodi haisee: merkki huonosta suunnittelusta

- Seuraavassa alan ehdoton asiantuntija Martin Fowler selittää mistä on kysymys **koodin hajuista**:
 - **A code smell is a surface indication that usually corresponds to a deeper problem in the system.** The term was first coined by Kent Beck while helping me with my Refactoring book.
 - The quick definition above contains a couple of subtle points. Firstly **a smell is by definition something that's quick to spot** - or sniffable as I've recently put it. *A long method is a good example of this - just looking at the code and my nose twitches if I see more than a dozen lines of java.*
 - The second is that smells don't always indicate a problem. Some long methods are just fine. You have to look deeper to see if there is an underlying problem there - smells aren't inherently bad on their own - they **are often an indicator of a problem rather than the problem themselves.**
 - One of the nice things about smells is that **it's easy for inexperienced people to spot them**, even if they don't know enough to evaluate if there's a real problem or to correct them. I've heard of lead developers who will pick a "smell of the week" and ask people to look for the smell and bring it up with the senior members of the team. Doing it one smell at a time is a good way of gradually teaching people on the team to be better programmers.

Koodihajuja

- Koodihajuja on hyvin monenlaisia ja monentasoisia
- Aloittelijankin on hyvä oppia tunnistamaan ja välttämään tavanomaisimpia
- Internetistä löytyy paljon hajulistoja, esim:
 - <http://sourcemaking.com/refactoring/bad-smells-in-code>
 - <http://c2.com/xp/CodeSmell.html>
 - <http://wiki.java.net/bin/view/People/SmellsToRefactorings>
 - <http://www.codinghorror.com/blog/2006/05/code-smells.html>
- Muutamia esimerkkejä aloittelijallekin helposti tunnistettavista hajuista:
 - Duplicated code (eli koodissa copy pastea...)
 - Methods too big
 - Classes with too many instance variables
 - Classes with too much code
 - Uncommunicative name
 - Comments
- Lääke koodihajuun on *refaktorointi* eli muutos koodin rakenteeseen joka kuitenkin pitää koodin toiminnan ennallaan

Koodin refaktorointi

- Erilaisia koodin rakennetta parantavia refaktorointeja on lukuisia
 - ks esim. <http://sourcemaking.com/refactoring>
- Pari hyvin käyttökelpoista ja nykyaikaisessa kehitysympäristössä (esim NetBeans, Eclipse, IntelliJ) automatisoitua refaktorointia:
 - **Rename method** (rename variable, rename class)
 - Eli uudelleennimetään huonosti nimetty asia
 - **Extract method**
 - Jaetaan liian pitkä metodi erottamalla siitä omia apumetodejaan
- Seuraavassa esimerkki vanhasta Ohjelmoinnin jatkokurssin tehtävästä
 - Koodi lisää luvun suuruusjärjestyksessä olevaan taulukkoon jos luku ei ole ennestään taulukossa
 - Ensin sotkuinen kaiken tekevä metodi
 - Seuraavaksi refaktoroitu versio jossa jokainen lisäämiseen liittyvä vaihe on erotettu omaksi selkeästi nimetyksi metodikseen
 - Osa näin tunnistetuista metodeista tulee käyttöön muidenkin metodien kuin lisäyksen yhteydessä
 - Lopputuloksena koodin rakenteen selkeys ja copy-pasten eliminointi joiden seurauksena ylläpidettävyys paranee

```
public boolean lisaa(int lisattava) {  
    boolean loytyiko = false;  
    for ( int i=0; i<alkioidenLkm; i++ )  
        if ( taulukko[i]==lisattava ) loytyiko = true;  
    if ( !loytyiko ) {  
        if (alkioidenLkm == taulukko.length) {  
            int[] uusi = new int[taulukko.length + kasvatuskoko];  
            for (int i = 0; i < alkioidenLkm; i++) uusi[i] = taulukko[i];  
            taulukko = uusi;  
        }  
        for( int i=0; i<alkioidenLkm; i++ )  
            if ( i==alkioidenLkm || taulukko[i]>=lisattava ) {  
                for ( int j=alkioidenLkm-1; i>=i; j-- ) taulukko[j+1] = taulukko[j];  
                taulukko[i] = lisattava;  
                alkioidenLkm++;  
            }  
        return true;  
    }  
    else return false;  
}
```

```

public boolean lisaa(int lisattava) {
    if ( kuuluuJoukkoon(lisattava) ) {
        return false;
    }

    if ( alkiodenLkm == taulukko.length ) {
        kasvataTaulukkoa();
    }

    int lisayskohta = etsiLisayskohta(lisattava);
    siirraEteenpainAlkaenKohdasta(lisayskohta);

    taulukko[lisayskohta] = lisattava;
    alkiodenLkm++;

    return true;
}

```

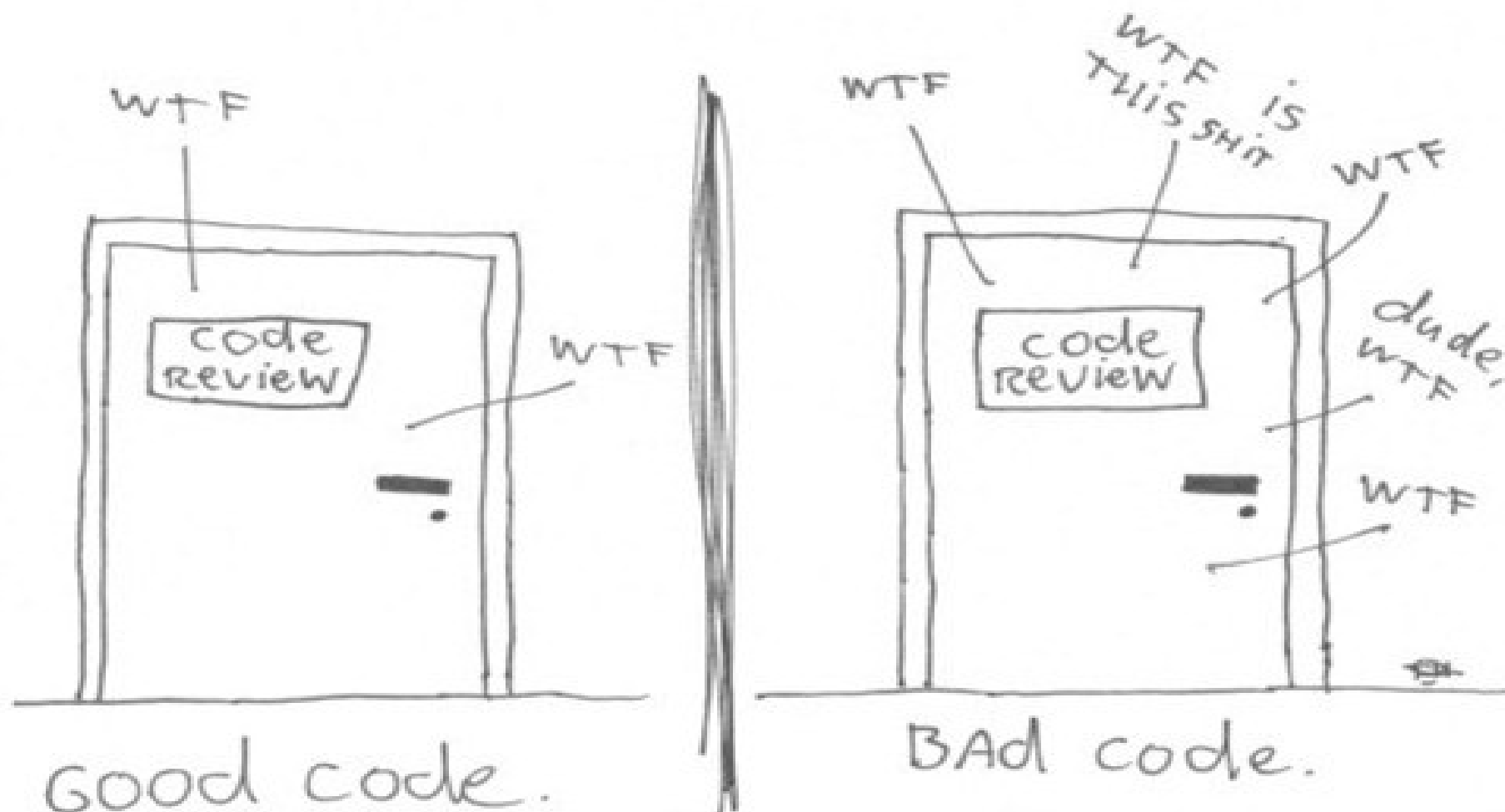
- Jokainen tässä kutsuttava metodi tekee kukin yhden pienen asian
- Apumetodit ovat lyhyitä, helppoja ymmärtää ja tehdä
- Apumetodit on helppo testata oikein toimivaksi
- Koodi kannattaa kirjoittaa osin jo alusta asti suoraan ”puhtaaksi ja hajuttomaksi”
 - Helpompaa tehdä ja saada toimimaan oikein
- Toisen vuoden kurssilla *Ohjelmistotuotanto* palataan tarkemmin refaktorointiin ja puhtaan koodin kirjoittamiseen



Miten refaktorointi kannattaa tehdä

- Refaktoroinnin melkein ehdoton edellytys on kattavien yksikkötestien olemassaolo
 - Refaktoroinninhan on tarkoitus ainoastaan parantaa luokan tai komponentin sisäistä rakennetta, ulospäin näkyvän toiminnallisuuden pitäisi pysyä muuttumattomana
- Kannattaa ehdottomasti edetä pienin askelin
 - Yksi hallittu muutos kerrallaan
 - Testit on ajettava mahdollisimman usein ja varmistettava että mikään ei mennyt rikki
- Refaktorointia kannattaa suorittaa lähes jatkuvasti
 - Koodin ei kannata antaa ”rapistua” pitkiä aikoja, refaktorointi muuttuu vaikeammaksi
 - Lähes jatkuva refaktorointi on helppoa, pitää koodin rakenteen selkeänä ja helpottaa sekä nopeuttaa koodin laajentamista
- Osa refaktoroinneista, esim. metodien tai luokkien uudelleennimentä tai pitkien metodien jakaminen osametodeiksi on helppoa, aina ei näin ole
 - Joskus on tarve tehdä isoja refaktorointeja joissa ohjelman rakenne eli arkkitehtuuri muuttuu

The ONLY valid MEASUREMENT OF Code QUALITY: WTFs/minute



Test Driven Development

Yksikkötestien kirjoittaminen valmiille koodille on työlästä ja tylsää

- Kirjoittamalla testejä ainoastaan valmiille koodille jää huomattava osa yksikkötestien hyödyistä saavuttamatta
 - Esim. refaktorointi edellyttäisi testejä
- JUnit ei ole alunperin tarkoitettu jälkikäteen tehtävien testien kirjoittamiseen, JUnitin kehittäjällä Kent Beckillä oli alusta asti mielessä jotain paljon järkevämpää ja mielenkiintoisempaa



Test Drive it!

TDD eli Test Driven Development

- TDD:ssä ohjelmoija (eikä siis erillinen testaaja) kirjoittaa testikoodin
- Testit laaditaan ennen koodattavan luokan toteutusta, yleensä jo ennen lopullista suunnittelua
- Sovelluskoodi kirjoitetaan täyttämään testien asettamat vaatimukset
 - Testit määrittelevät miten ohjelmoitavan luokan tulisi toimia
 - Testit toimivatkin osin koodin dokumentaationa, sillä testit myös näyttävät miten testattavaa koodia käytetään
- Testaus ohjaa kehitystyötä eikä ole erillinen toteutuksen jälkeinen laadunvarmistusvaihe
 - Oikeastaan TDD ei ole testausmenetelmä vaan ohjelmiston kehitysmenetelmä, joka tuottaa sivutuotteenaan automaattisesti ajettavat testit
- Testien on ennen toteutuksen valmistumista epäonnistuttava
 - Näin pyritään varmistamaan, että testit todella testaavat haluttua asiaa
- Toiminnallisuus katsotaan toteutetuksi, kun testit menevät läpi

Oikeaoppinen TDD-sykli

(1) tehdään **yksi** testitapaus

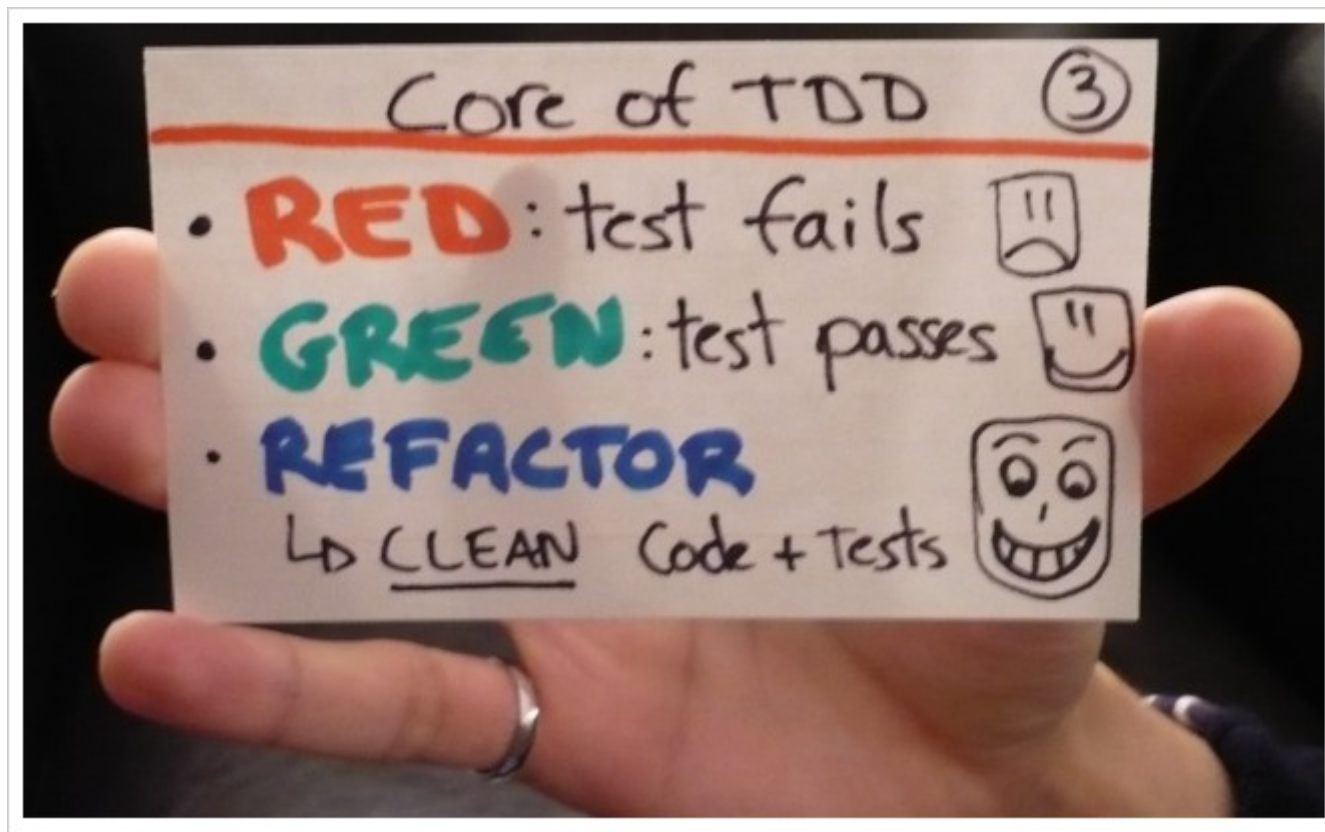
- testitapaus testaa ainoastaan yhden ”pienen” asian

(2) Tehdään koodi joka läpäisee testitapauksen

(3) **refaktoroidaan** koodia, eli parannellaan koodin laatua ja struktuuria

- Testit varmistavat koko ajan ettei mitään mene rikki

Kun koodin rakenne on kunnossa, palataan vaiheeseen (1)



TDD

- Automaattinen testaus ja TDD ovat usein osana ketterää ohjelmistokehitystä
- Mahdollistaa turvallisen refaktoroinnin
 - Koodi ei rupea haisemaan
 - Ohjelman rakenne säilyy laajennukset mahdollistavana
- Tämän viikon laskareiden paikanpäällä tehtävässä tehtävässä pääsemme itse kokeilemaan TDD:tä

