Ohjelmistotuotanto

Luento 9

13.4.2014

Ohjelmiston suunnittelu, kestausta

- Suunnittelun ajatellaan yleensä jakautuvan kahteen vaiheeseen:
  - **Arkkitehtuurisuunnittelu**
    - Ohjelman rakenne karkealla tasolla
    - Mistä suuremmista rakennekomponenteista ohjelma koostuu?
    - Miten komponentit yhdistetään, eli komponenttien väliset rajapinnat
    - Useimmiten ohjelma noudattaa jotain hyvin tunnettua **arkkitehtuurista mallia**, kuten kerrosarkkitehtuuria, MVC:tä jne
  - **Oliosuunnittelu**
    - yksittäisten komponenttien suunnittelu
- Suunnittelun ajoittuminen riippuu käytettävästä tuotantoprosessista:
  - Ketterissä menetelmissä suunnittelua tehdään tarvittava määrä jokaisessa iteraatiossa **(inkrementaalinen design ja arkkitehtuuri**), tarkkaa suunnitteludokumenttia ei yleensä ole
- Jos ei olla tekemässä ”kertakäyttökoodia” tai ottamassa tietoisesti teknistä velkaa, on oliosuunnittelussa tärkeää pitää mielessä ohjelman ylläpidettävyys ja laajennettavuus

Helposti ylläpidettävän koodin tunnusmerkit

- Ylläpidettävyyden ja laajennettavuuden kannalta tärkeitä seikkoja
  - Koodin tulee olla luettavuudeltaan selkeää, eli koodin tulee kertoa esim. nimennällään mahdollisimman selkeästi mitä koodi tekee, eli tuoda esiin koodin alla oleva ”design”
  - Yhtä paikkaa pitää pystyä muuttamaan siten, ettei muutoksesta aiheudu sivuvaikutuksia sellaisiin kohtiin koodia, jota muuttaja ei pysty ennakoimaan
  - Jos ohjelmaan tulee tehdä laajennus tai bugikorjaus, tulee olla helppo selvittää mihin kohtaan koodia muutos tulee tehdä
  - Jos ohjelmasta muutetaan ”yhtä asiaa”, tulee kaikkien muutosten tapahtua vain yhteen kohtaan koodia (metodiin tai luokkaan)
  - Muutosten ja laajennusten jälkeen tulee olla helposti tarkastettavissa ettei muutos aiheuta sivuvaikutuksia muualle järjestelmään
- Näin määritelty koodin *sisäinen laatu* on erityisen tärkeää ketterissä menetelmissä, joissa koodia laajennetaan iteraatio iteraatiolta
- Jos koodin sisäiseen laatuun ei kiinnitetä huomiota, on väistämätöntä että pidemmässä projektissa kehitystiimin velositeetti alkaa tippua ja eteneminen alkaa vaikeutua iteraatio iteraatiolta
  - Koodin sisäinen laatu on siis usein myös asiakkaan etu

Koodin laatuattribuutteja

- Edellä lueteltuihin hyvän koodin tunnusmerkkeihin päästään kiinnittämällä huomio seuraaviin *laatuattribuutteihin*
  - Kapselointi
  - Koheesio
  - Riippuvuuksien vähäisyys
  - Toisteettomuus
  - Testattavuus
  - Selkeys
- Jatketaan laatuattribuutteihin ja niitä tukeviin ”ikiaikaisiin” hyvän suunnittelun periaatteisiin sekä erilaisissa tilanteissa toimiviksi todettuihin geneerisiä suunnitteluratkaisuja dokumentoiviin *suunnittelumalleihin* tutustumista

Lisää suunnittelumalleja

Olion rikastaminen dekoraattorilla

- Joskus eteen tulee tarve lisätä olioon jotain ekstraominaisuuksia, pitäen kuitenkin olio sellaisena, että sitä käyttäviin ohjelmanosiin ei tarvitse tehdä muutoksia
- **Dekoraattori** (decorator) -suunnittelumalli tuo avun
  - http://sourcemaking.com/design_patterns/decorator
- Dekoraattorissa muodostetaan ”rikastettu” olio, jolla on täysin sama rajapinta kuin oliolla, johon lisäominaisuuksia halutaan
  - Dekoraattoriolio yleensä delegoi varsinaisen tehtävän, eli olion vanhan vastuun suorittamisen alkuperäiselle oliolle
- Katsotaan ensin hieman yksinkertaisempaa tapausta
- ks https://github.com/mluukkai/ohtu2015/blob/master/web/luento9.md Dekoroitu Random
- Esimerkissä tehdään dekoroitu Random-olio, jonka avulla on mahdollista testata satunnaislukuja käyttävää ohjelmaa
  - Dekoroitu Random ottaa talteen kaikki arvotut luvut
  - Testissä käytetään dekoroitua versiota normaalin Randomin sijaan
  - Testi pääsee kysymään dekoroidulta randomilta arvotut numerot

Dekoroitu pino, pinotehdas ja rakentaja

- Tarkastellaan esimerkkejä Dekoroitu Pino ja Pinotehdas osoitteessa https://github.com/mluukkai/ohtu2015/blob/master/web/luento9.md
- Saamme dekoraattorin avulla hienosti tehtyä monen eri ominaisuuskombinaation omaavia pinoja
- Dekoroitujen pinojen luominen on monimutkaista, mutta Factoryn avulla saamme peitettyä monimutkaisuuden pinon käyttäjältä
- Factorystä muodostuu kuitenkin ongelma...
- **Rakentaja** (engl builder) -suunnittelumalli kuitenkin ratkaisee ongelman!
- Rakentajassa on kiinnitetty erityinen huomio metodien nimeämiseen:

```
Pinorakentaja rakenna = new Pinorakentaja();

Pino pino = rakenna.kryptattu().prepaid(10).pino();
```

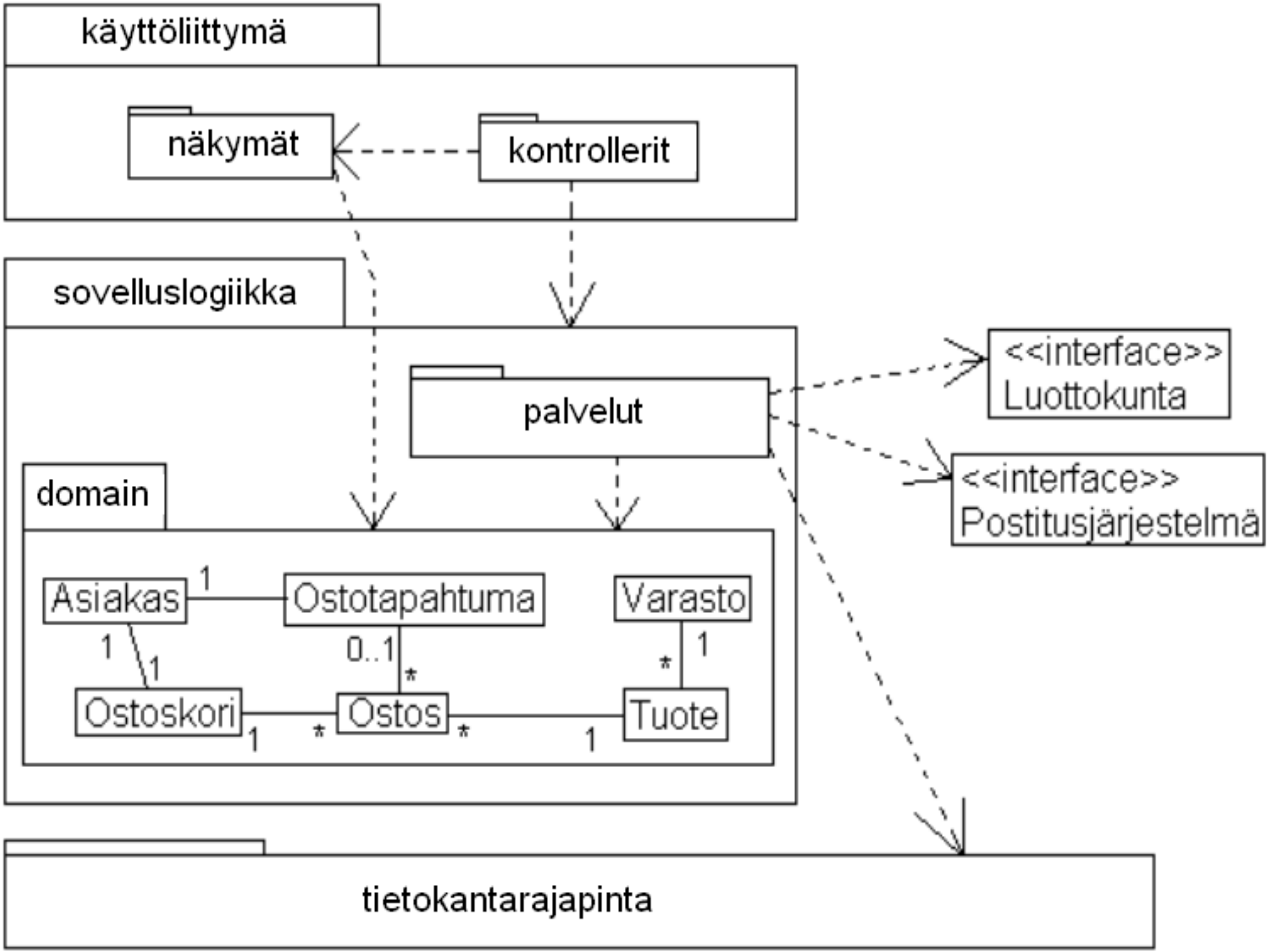
- On haettu mahdollisimman luonnollista kieltä muistuttavaa luettavuutta
- Muodostettiin **DSL (domain specific language)** pinojen luomiseen
  - http://martinfowler.com/bliki/FluentInterface.html
  - http://www.infoq.com/articles/internal-dsls-java

Luokan rajapinnan muuttaminen adapterilla

- Äsken käsiteltyjen suunnittelumallien, dekoraattorin, komposiitin ja proxyn yhteinen puoli on, että saman ulkokuoren eli rajapinnan takana voi olla yhä monimutkaisempaa toiminnallisuutta, joka on kuitenkin täysin kapseloitu käyttäjältä
- Tarkastellan nyt tilannetta, jossa käytettävissä on luokka, joka oleellisesti ottaen tarjoaa halutun toiminnallisuuden, mutta sen rajapinta on hieman vääränlainen esim. metodien nimien tai parametrien osalta
  - Perintä ei siis sovi ratkaisumenetelmäksi
- Alkuperäistä luokkaa ei kuitenkaan haluta tai voida muuttaa sillä muutos rikkoisi luokan muut käyttäjät
- **Adapteri-suunnittelumalli** sopii tällaisiin tilanteisiin
  - http://sourcemaking.com/design_patterns/adapter
- Tutkitaan esimerkkiä ”adapteri” sivulta https://github.com/mluukkai/ohtu2015/blob/master/web/luento9.md
  - Pino adaptoidaan sopimaan rajapinnaltaan paremmin uuteen käyttötilanteeseen

Paluu suuriin linjoihin

- Arkkitehtuurin yhteydessä mainitsimme kerrosarkkitehtuurin, josta esimerkkinä oli Kumpula biershopin arkkitehtuuri
- Kerroksittaisuudessa periaate on sama kuin useiden suunnittelumallien ja hyvän oliosuunnittelussa yleensäkin **kapseloidaan monimutkaisuutta ja detaljeja rajapintojen taakse**
- Tarkoituksena ylläpidettävyyden parantaminen ja kompleksisuuden hallinnan helpottaminen
  - Kerroksen N käyttäjää on turha vaivata N:n sisäisellä rakenteella
  - Eikä sitä edes kannata paljastaa koska näin muodostuisi eksplisiittinen riippuvuus käyttäjän ja N:n välille
- Pyrkimys siihen että *kerrokset ovat mahdollisimman korkean koheesion omaavia*, eli ”yhteen asiaan” keskittyvä
  - Käyttöliittymä
  - Tietokantayhteydet
  - Liiketoimintalogiikka
- Kerrokset taas ovat keskenään mahdollisimman *löyhästi kytkettyjä*


käyttöliittymä
näkymät
kontrollerit
sovelluslogiikka
palvelut
<<interface>>
Luottokunta
<<interface>>
Postitusjärjestelmä
domain
Asiakas
Ostotapahtuma
Varasto
Ostoskori
Ostos
Tuote
1
1
1
1
0..1
*
*
1
*
*
1
tietokantarajapinta

Domain Driven Design

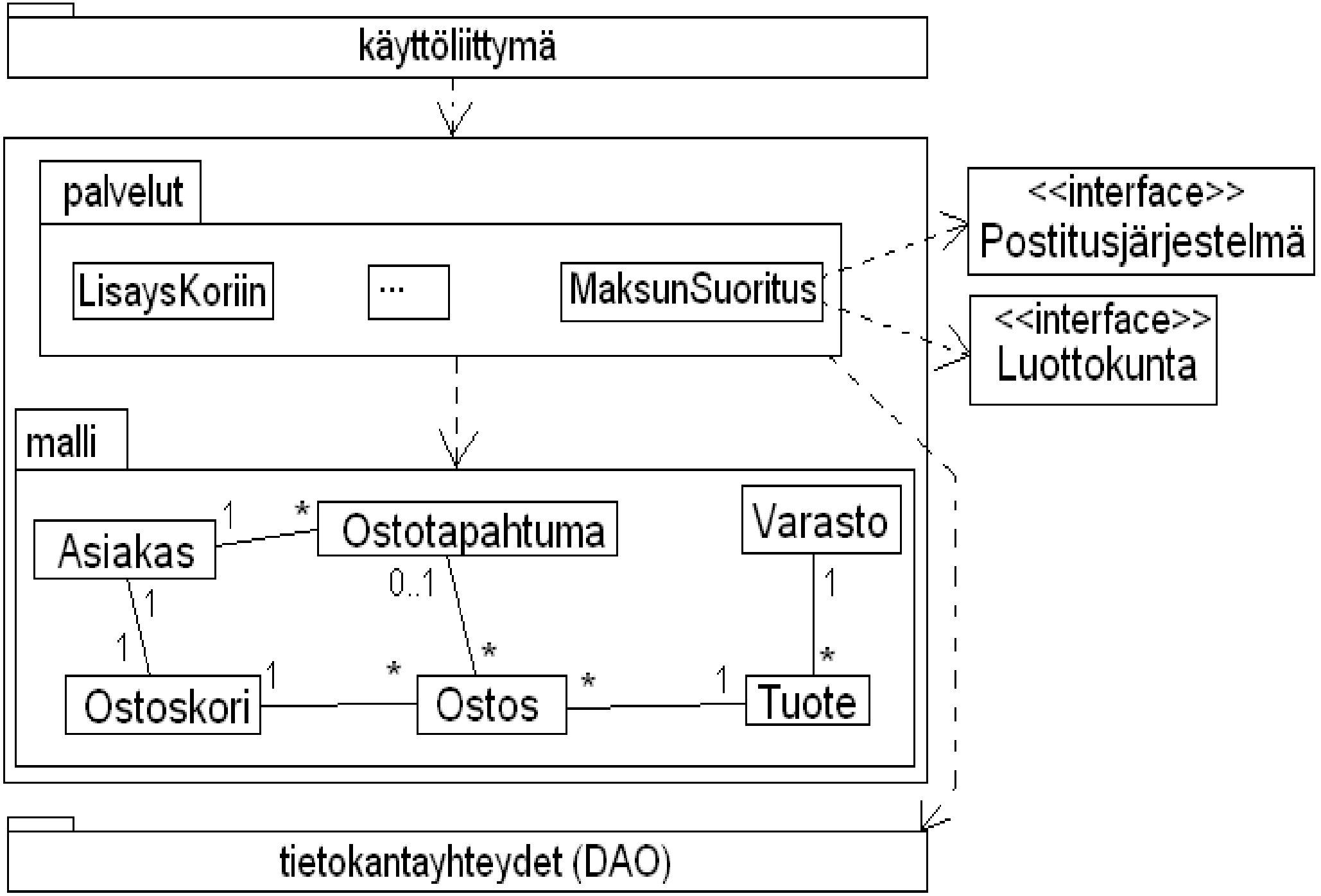
- Viimeaikaisena voimakkaasti nousevana trendinä on käyttää sovelluksen koodin tasolla nimentää, joka vastaa liiketoiminta-alueen eli ”bisnesdomainin” terminologiaa
  - Yleisnimike tälle tyylille on Domain Driven Design, DDD
  - ks esim. http://www.infoq.com/articles/ddd-evolving-architecture
- Ohjelmiston arkkitehtuurissa on DDD:tä sovellettaessa (ja muutenkin kerrosarkkitehtuuria sovellettaessa) on kerros joka kuvaa domainin, eli sisältää liiketoimintaoliot
- Esim. Kumpula Biershopin domain-oliot:
  - Tuote
  - Varasto
  - Ostos
  - Ostoskori
  - Asiakas
  - Ostostapahtuma

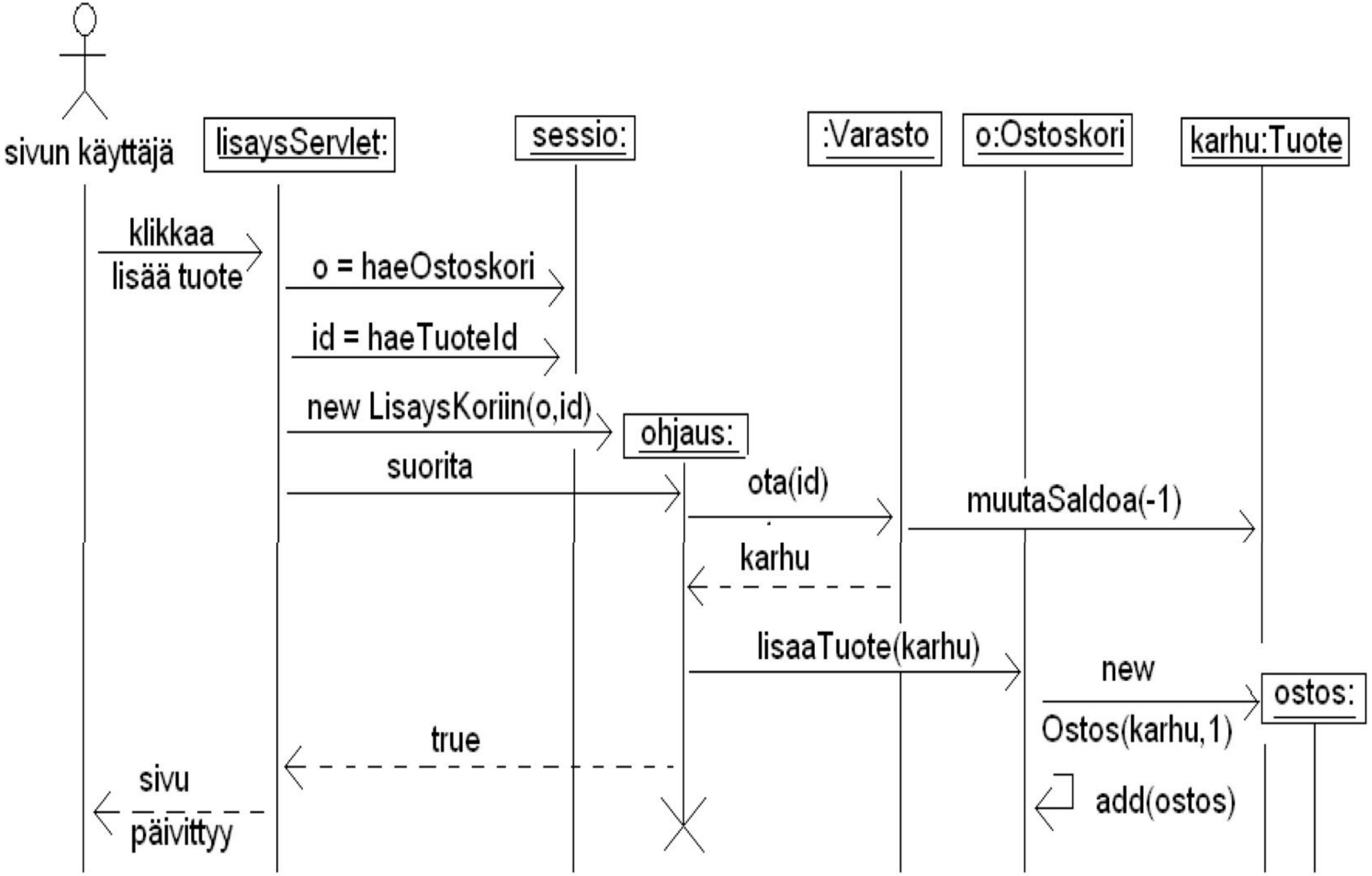
Domain Driven Design

- Domain-oliot tai osa niistä yleensä mäpätään tietokantaan
  - Mäppäyksessä käytetään usen DAO-suunnittelumallia, johon tutustuimme ohimennen laskareissa 3
  - DAO on oleellisesti sama asia jota kutsutaan data mapperiksi:
    - http://martinfowler.com/eaaCatalog/dataMapper.html
  - DAO:n lisäksi on muitakin mäppäystapoja, kuten Ruby on Railsin käyttämä Active Record http://martinfowler.com/eaaCatalog/activeRecord.html
- Domain-oliot tietokantaan mäppäävät komponentit muodostavat oman kerroksen kerrosarkkitehtuurissa
- Joissain suunnittelutyyleissä Domain-olioiden ja sovelluksen käyttöliittymän välissä on vielä erillinen palveluiden kerros
  - http://martinfowler.com/eaaCatalog/serviceLayer.html
- Palvelut koordinoivat domain-olioille suoritettavaa toiminnallisuutta, esim. *ostoksen laitto ostoskoriin* tai *ostosten maksaminen*
- Ideana on eristää palveluiden avulla kaikki sovelluslogiikka käyttöliittymältä

Palvelukerros Kumpula Biershopissa

- Palvelukerroksessa on jokaisen käyttöliittymätason toiminnallisuuden toteutus omana **command**-suunnittelumallin mukaisena oliona
  - Parin sivun päästä havainnollistavana esimerkkinä LisäysKoriin-olion luonti ja kutsu
  - LisäysKoriin-olio suorittaa kaiken interaktion domain-olioiden kanssa
  - Käyttöliittymä käyttää domain-olioita ainoastaan web-sivulla näytettävän datan renderöintiin
- Komento-oliot muodostavat oikestaan **fasaadi**-suunnitelumallin mukaisen eristävän kerroksen käyttöliittymän ja alempien kerrosten välille
  - Tarjoaa hyvin rajatun rajapinnan jonka kautta kerrosta käytetään, eristää kerroksen toiminnallisuuden täysin
  - http://sourcemaking.com/design_patterns/facade
- Sovelluslogiikan testaaminen ilman käyttöliittymää onnistuu helposti yksikkötesteillä testaamalla command-olioiden ja domain-olioiden interaktiota


käyttöliittymä
palvelut
LisaysKoriin
...
MaksunSuoritus
<<interface>>
Postitusjärjestelmä
<<interface>>
Luottokunta
malli
Asiakas
Ostotapahtuma
Varasto
Ostoskori
Ostos
Tuote
1
*
0..1
tietokantayhteydet (DAO)

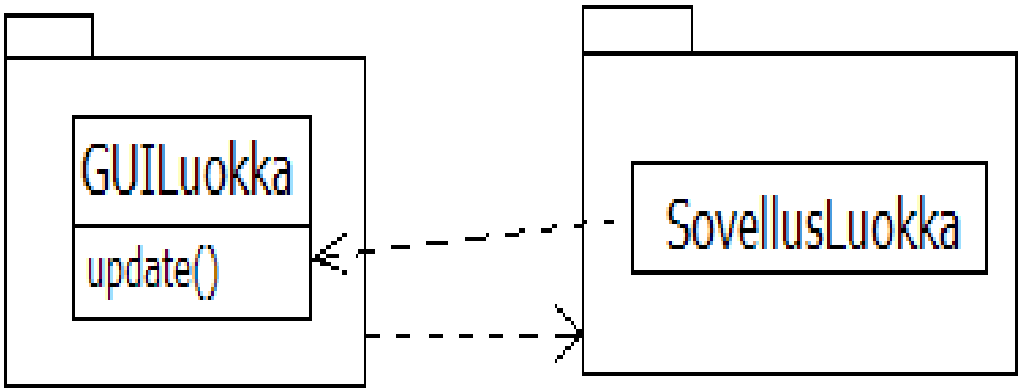

sivun käyttäjä
lisaysServlet:
sessio:
:Varasto
o:Ostoskori
karhu:Tuote
klikkaa
lisää tuote
o = haeOstoskori
id = haeTuoteId
new LisaysKoriin(o,id)
ohjaus:
suorita
ota(id)
muutaSaldoa(-1)
karhu
lisaaTuote(karhu)
new
Ostos(karhu,1)
ostos:
true
sivu
päivittyy
add(ostos)

Model View Controller el MVC -malli

- MVC-mallilla tarkoitetaan periaatetta, jonka avulla **malli** (model) eli liiketoimintalogiikan sisältävät oliot (esim. domain-oliot) eristetään käyttöliittymän **näytöt** (view) generoivasta koodista
  - Kumpula Biershopissa on oikeastaan sovellettu WebMVC:tä, eli MVC:n www-sovelluksiin sopivaa varianttia
- Ideana on laittaa näytön/näytöt generoivan koodin ja sovelluslogiikasta huolehtivien olioiden väliin **kontrolleri** (controller)
- Kontrolleri huolehtii esim. nappien klikkaamisen tai web-sovelluksissa osoitteisiin navigoinnin tai lomakkeiden lähettämisen edellyttävän toiminnallisuuden suorittamisesta kutsumalla sopivia modelin olioita
- Näytöt generoivat käyttäjälle näytettävän käyttöliittymän käyttäen joko suoraan malleissa olevaa dataa tai saamalla datan kontrollerin välityksellä (kuten WebMVC:ssä tapahtuu)
  - ks. https://github.com/mluukkai/ohtu2015/blob/master/web/luento9.md kohta MVC
- Model ei tunne kontrollereja eikä näyttöjä ja samaan modelissa olevaan dataan voikin olla useita näyttöjä

Riippuvuuksien eliminointi

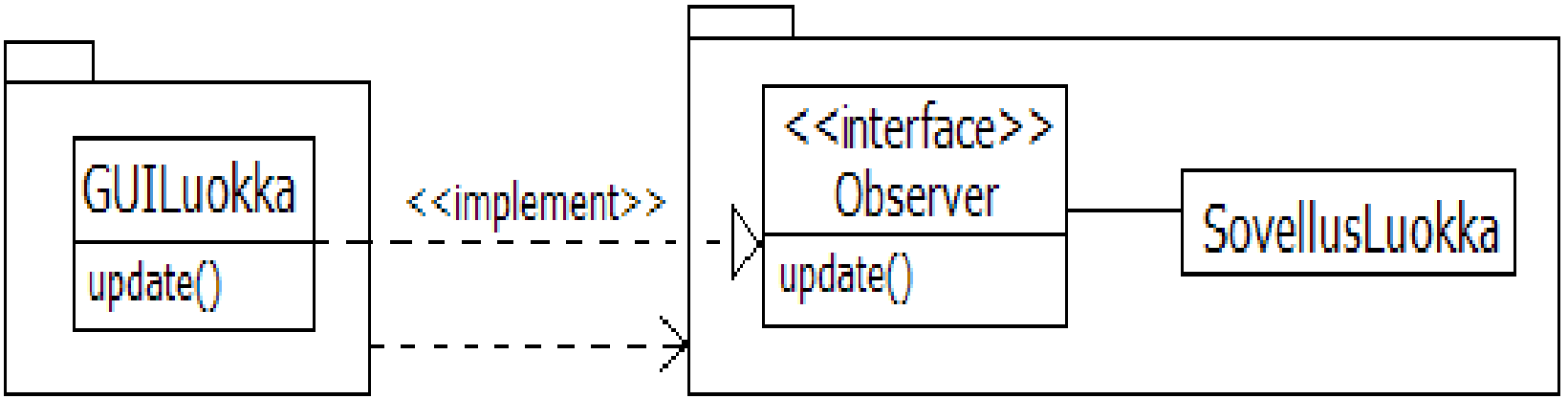
- Kerrosarkkitehtuurissa ja MVC-mallin mukaisissa sovelluksissa törmätään usein tilanteeseen, jossa sovelluslogiikan on kerrottava käyttöliittymälle jonkin sovellusolion tilan muutoksesta, jotta käyttöliittymä näyttäisi koko ajan ajantasaista tietoa
- Tästä muodostuu ikävä riippuvuus sovelluslogiikasta käyttöliittymään
- Kuvitellaan, että sovelluslogiikka ilmoittaa muuttuneesta tilasta kutsumalla jonkin käyttöliittymän luokan toteuttamaa metodia *update()*
  - Parametrina voidaan esim. kertoa muuttunut tieto



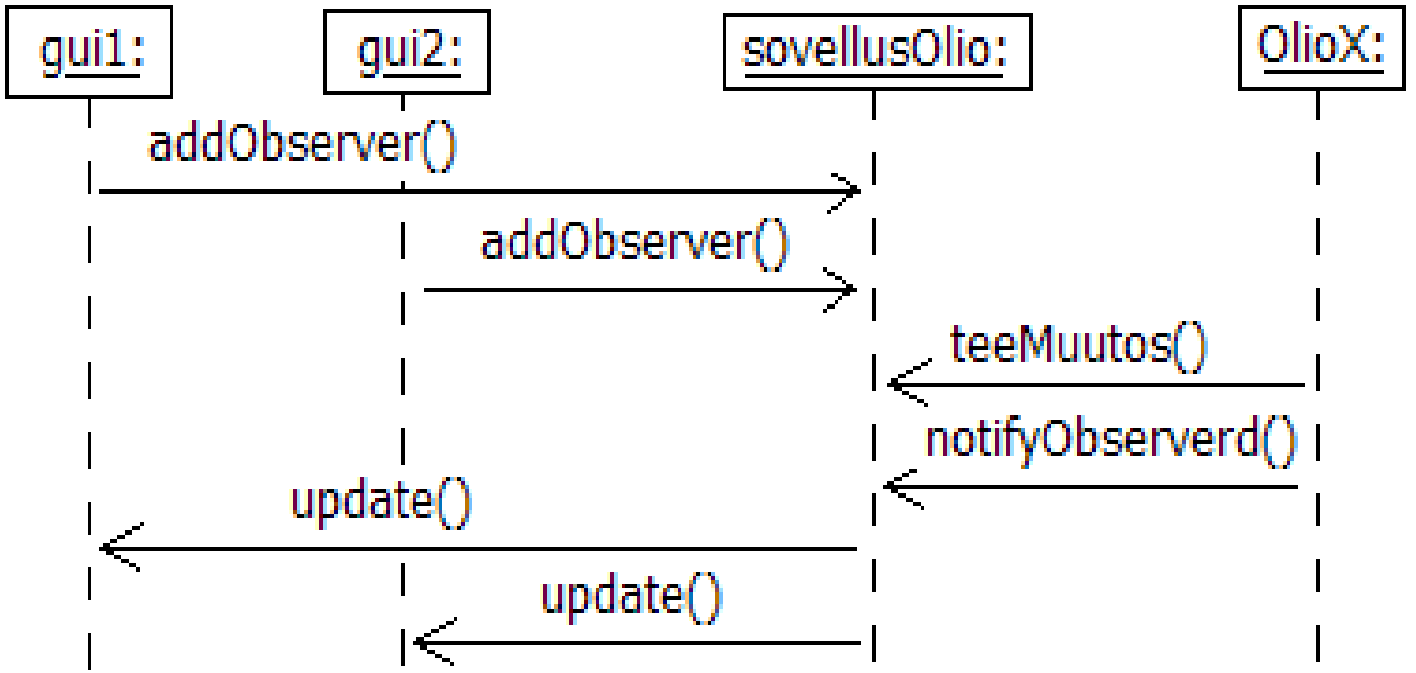
- Riippuvuus saadaan eliminoitua **observer**-suunnittelumallilla
  - Ks https://github.com/mluukkai/ohtu2015/blob/master/web/luento9.md kohta Observer

Riippuvuuksien eliminointi observer-suunnittelumallilla

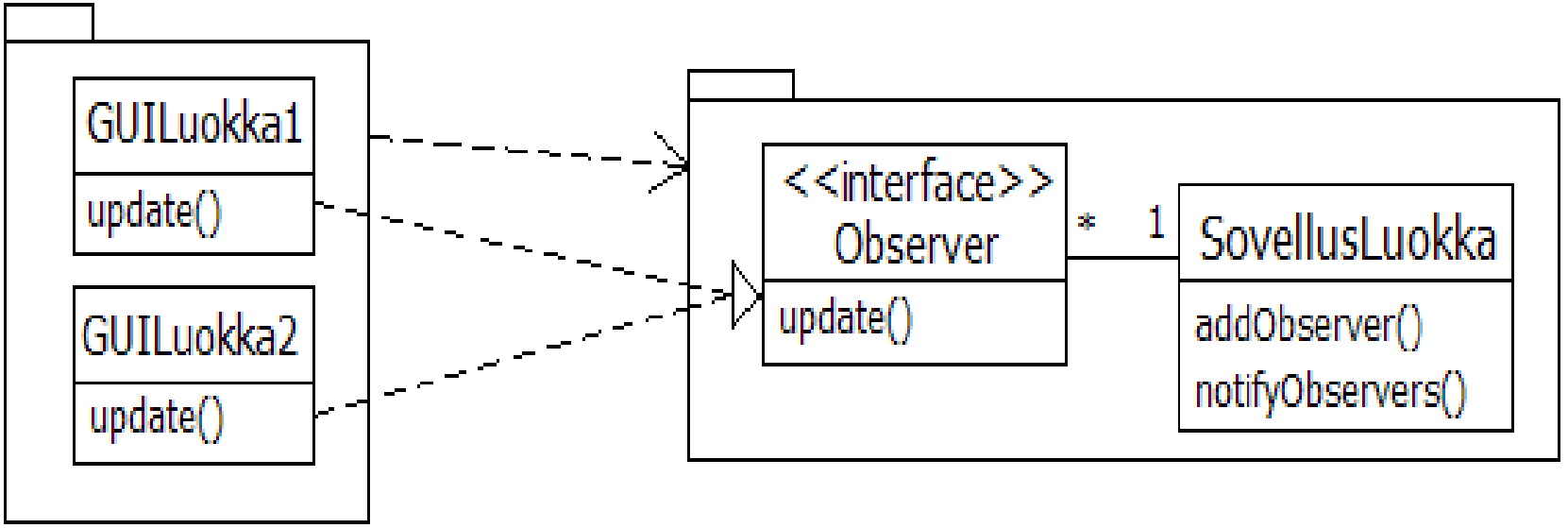
- Määritellään rajapinta, joka sisältää käyttöliittymäluokan päivitysmetodin update(), jota sovellusluokka kutsuu
  - Alla rajapinnalle on annettu nimeksi *Observer*
- Käyttöliittymäluokka toteuttaa rajapinnan, eli käytännössä toteuttaa update()-metodin haluamallaan tavalla
- Sovellusluokalle riittää nyt tuntea ainoastaan rajapinta, jonka metodia update() se tarvittaessa kutsuu
- Nyt kaikki menee siististi, sovelluslogiikasta ei enää ole riippuvuutta käyttöliittymään ja silti sovelluslogiikka voi kutsua käyttöliittymän metodia
  - Sovellusluokka tuntee siis vain rajapinnan, joka on määritelty sovelluslogiikkapakkauksessa



- Kyseessä on **observer**- eli tarkkailijasuunnittelumalli
  - http://sourcemaking.com/design_patterns/observer
- Jos käyttöliittymäolio haluaa tarkkailla jonkun sovellusolion tilaa, se toteuttaa Observer-rajapinnan ja rekisteröi rajapintansa tarkkailtavalle sovellusoliolle
  - Sovellusoliolla metodi addObserver()
  - Näin sovellusolio tuntee kaikki sitä tarkkailevat rajapinnat
- Kun joku muuttaa sovellusolion tilaa, kutsuu se sovellusolion metodia notifyObservers(), joka taas kutsuu kaikkien tarkkailijoiden update()- metodeja, joiden parametrina voidaan tarvittaessa välittää muutostieto



Observer-suunnittelumalli




```
class Sovellusluokka{
    ArrayList<Observer> tarkkailijat;
    void addObserver(Observer o){
        tarkkailijat.add(o);
    }
    void notifyObservers(){
        for ( Observer o : tarkkailijat) o.update();
    }
    /* muu koodi */
}
```

```
Interface Observer{
    void update();
}
```

```
GUILuokka implements Observe {
    void update(){
            /* päivitetään näyttöä */
    }
    /* muu koodi*/
}
```