

Ohjelmistotuotanto

Luento 9

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/ohtu2014/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/ohtu2014/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>

Komposiitti ja proxy

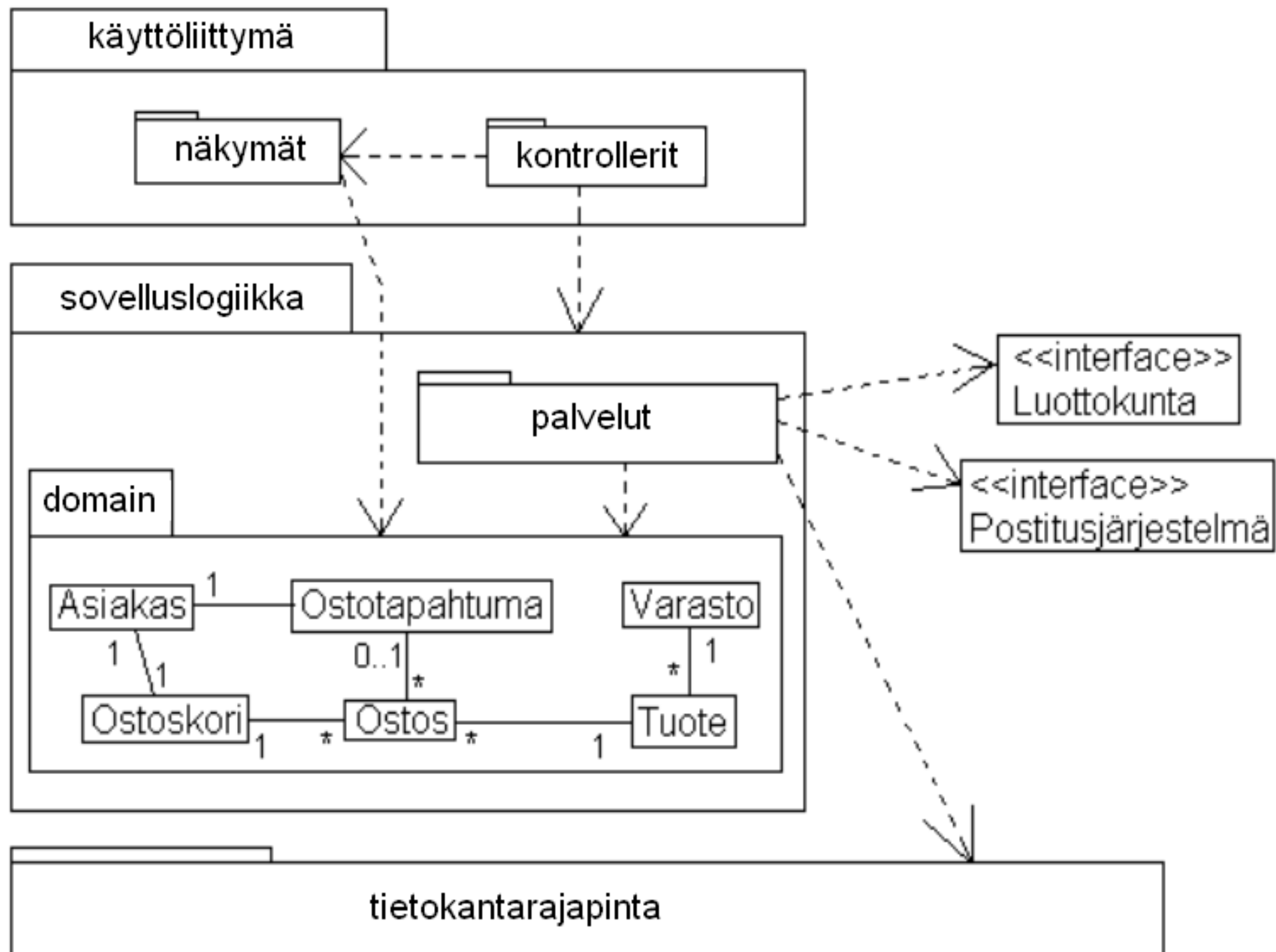
- <https://github.com/mluukkai/ohtu2014/blob/master/web/luento9.md> olevat esimerkit Komposiitti ja Proxy demonstroivat jälleen kahta suunnittelumallia
- Komposiitti on tapa järjestää puu/rekursiomaisesti rakentuvia samankaltaisesti ulospäin käyttäytyviä olioita
 - http://sourcemaking.com/design_patterns/composite
- Komposiitti kapseloi yhden rajapinnan taakse joko yksittäisen olion (kuten esimerkissä erotinelementin) tai mielivaltaisen monimutkaisen elementeistä koostuvan puumaisen rakenteen
 - Käyttäjän eli esimerkissämme dokumentin kannalta yksittäisen elementin sisäisellä rakenteella ei ole merkitystä, elementti osaa tulostaa itsensä ja se riittää dokumentille
- Joskus käytettävä olio voi olla luonteeltaan sellainen, että olion itsensä käyttö on raskasta ja usein riittää että oliaa edustaa joku muu siihen asti kunnes olia itseään todellakin tarvitaan
- Tällaisissa tilanteissa proxy-suunnittelumalli tuo ratkaisun
 - http://sourcemaking.com/design_patterns/proxy
 - Esimerkissämme web-elementti toteutettiin proxyn avulla

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
- Tarkastellaan 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/ohtu2014/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*



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 määrittävät tietokantaan
 - Mäppäyksessä käytetään usein DAO-suunnittelumallia, johon tutustuimme ohimennen laskareissa 3
 - DAO on oleellisesti sama asia jota kutsutaan data mapperiksi:
 - <http://martinfowler.com/eaCatalog/dataMapper.html>
 - DAO:n lisäksi on muitakin mäppäystapoja, kuten Ruby on Railsin käyttämä Active Record
<http://martinfowler.com/eaCatalog/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/eaCatalog/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 oikeastaan **fasaadi**-suunnittelumallin 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

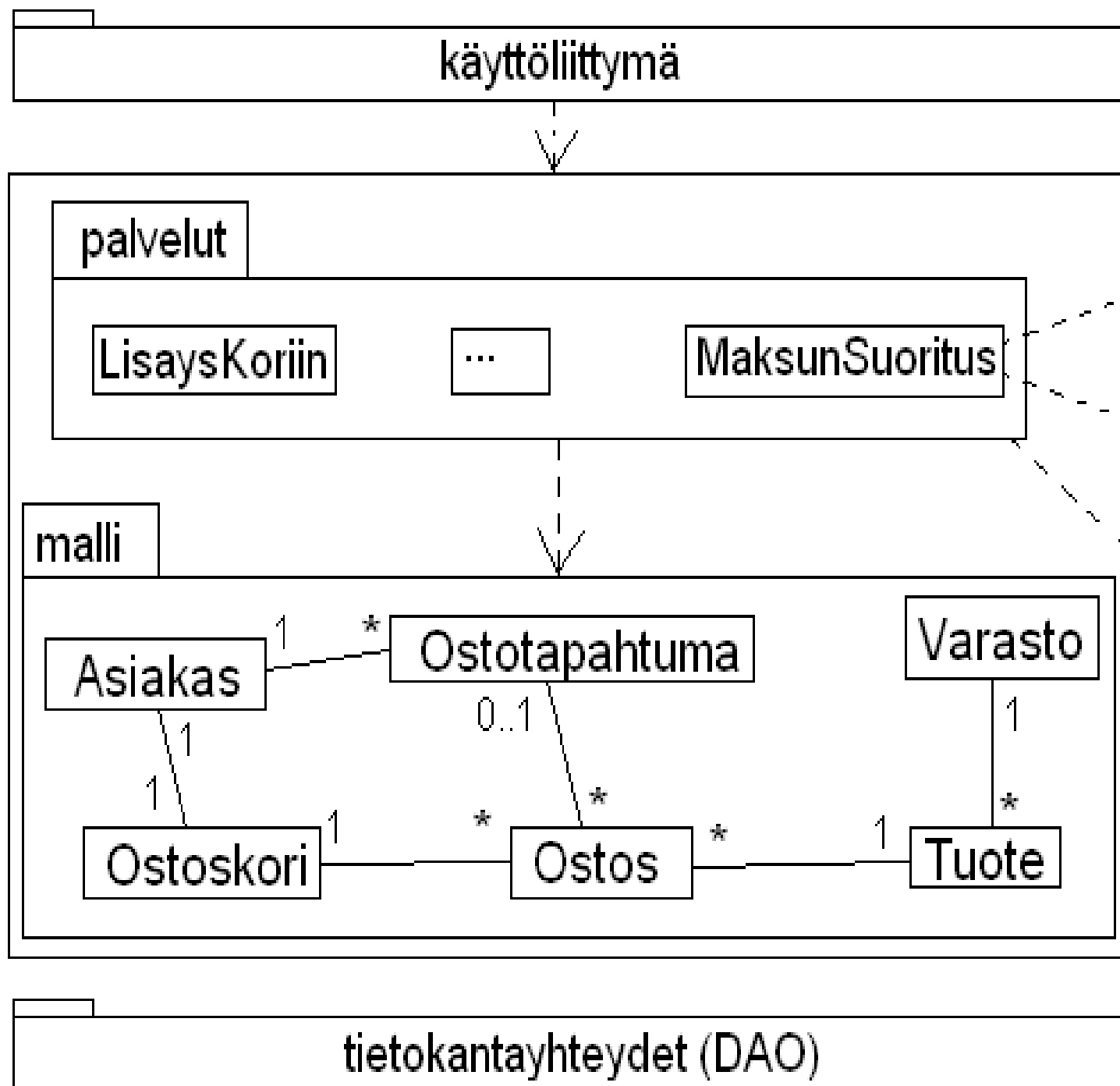
Varasto

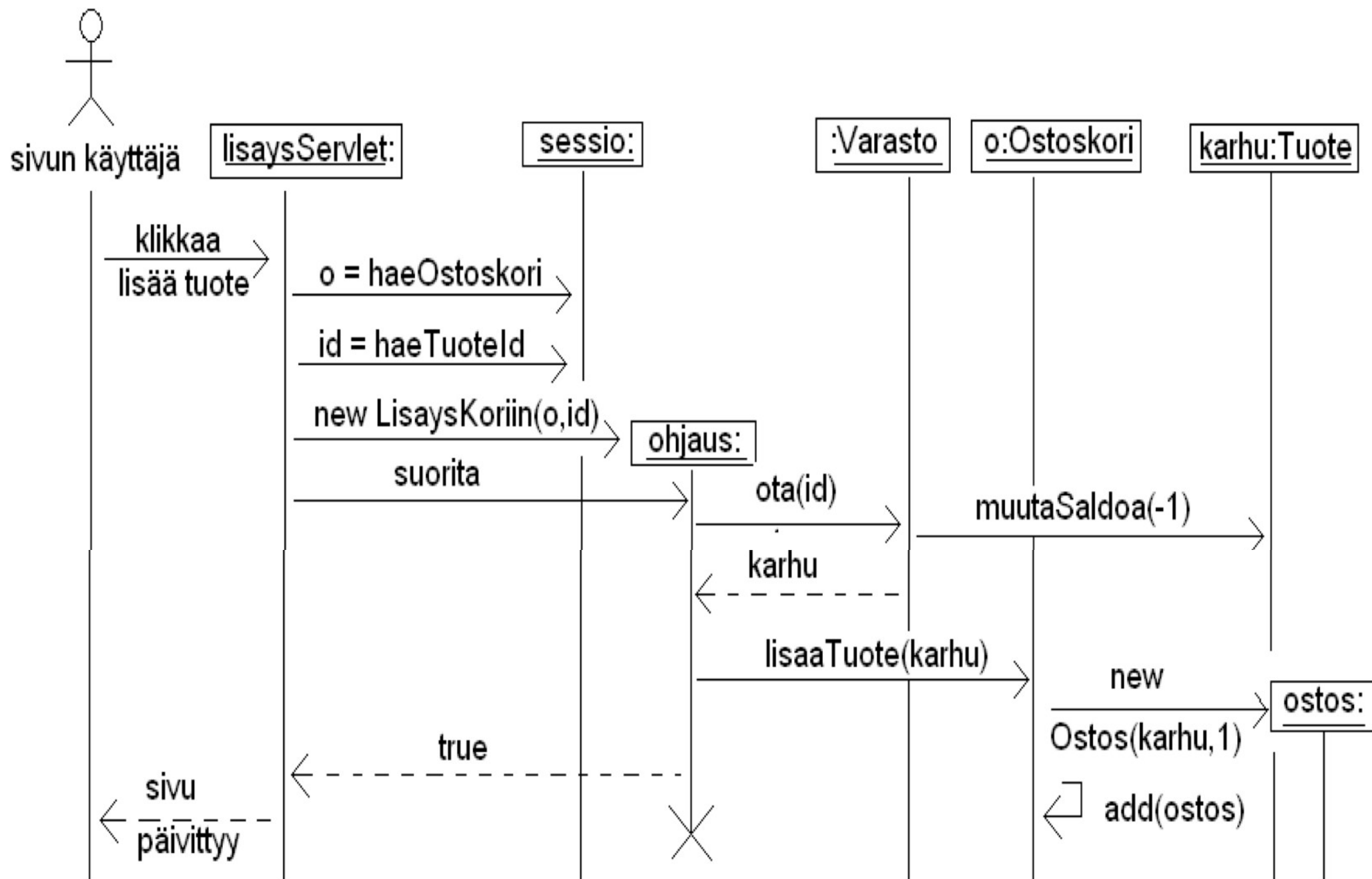
Ostoskori

Ostos

Tuote

tietokantayhteydet (DAO)



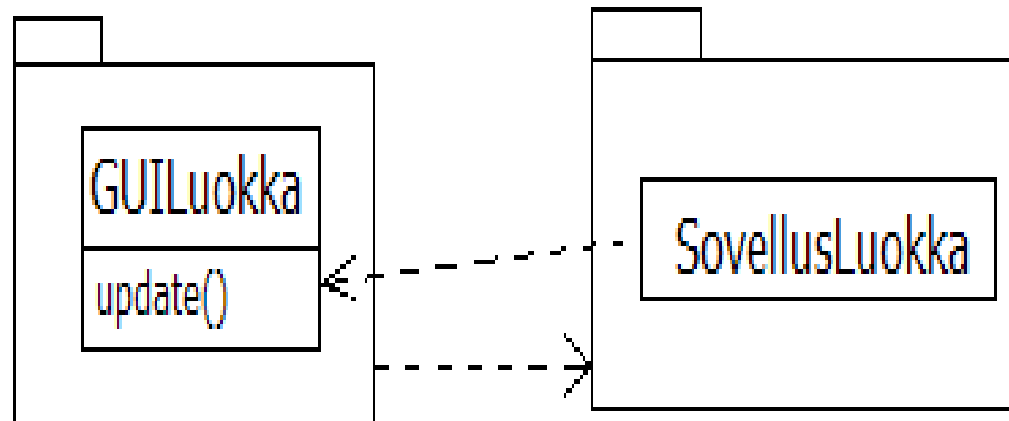


Model View Controller eli MVC -malli

- 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-sovellukseen 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/ohtu2014/blob/master/web/luento9.md> kohta MVC
- Model ei tunne kontrollereja eikä näyttöjä ja samaan modelissa olevaan dataan voikin olla useita näyttöjä

Riippuvuuksien eliminointi

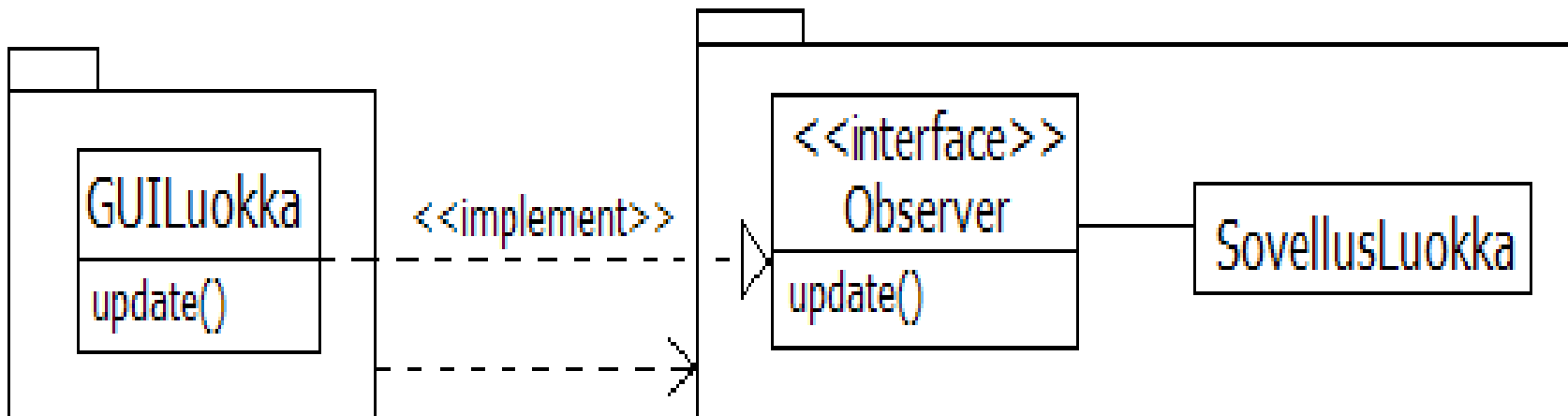
- Kerrosarkkitehtuurissa ja MVC-mallin mukaisissa sovelluksissa törmätään usein tilanteeseen, jossa sovelluslogiikan on kerrottava käyttöliittymälle jonkin sovellusolion tilan muutoksesta, jotta käyttöliittymä näyttäisi koko ajan ajantasaista tietoa
- Tästä muodostuu ikävä riippuvuus sovelluslogiikasta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



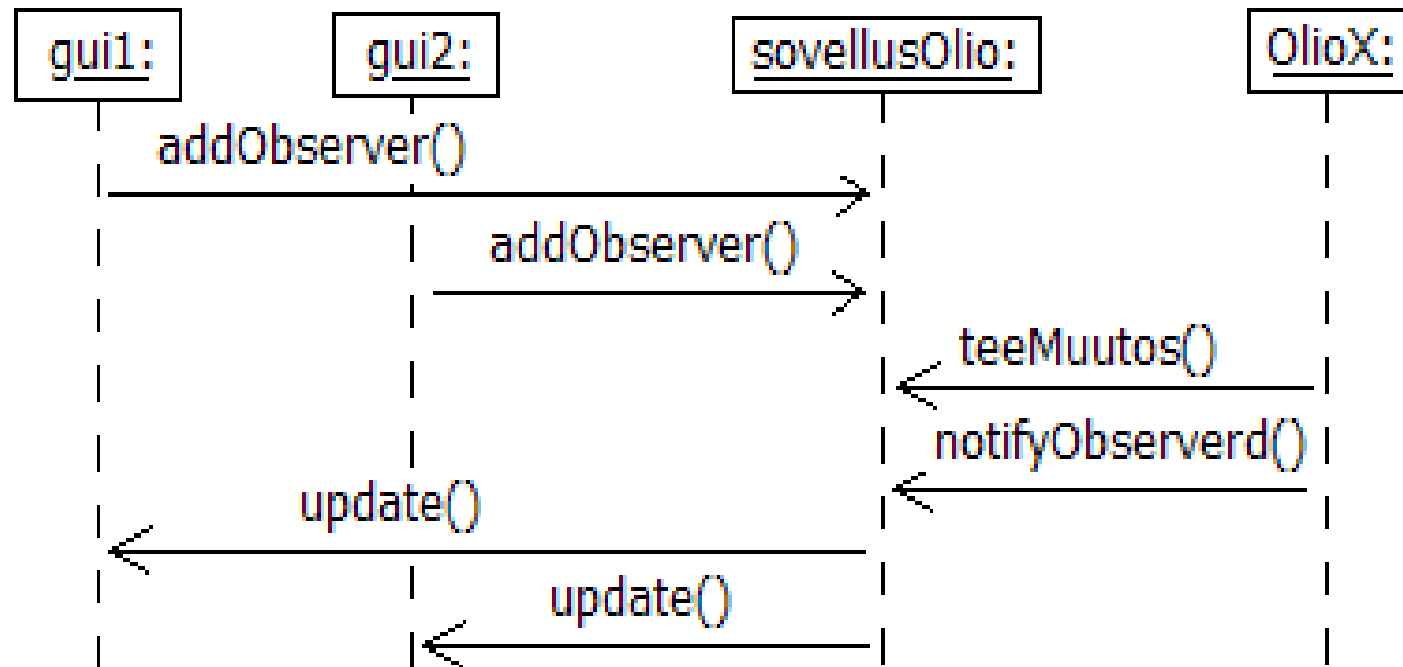
- Riippuvuus saadaan eliminointua **observer**-suunnittelumallilla
 - Ks <https://github.com/mluukkai/ohtu2014/blob/master/web/luento9.md> kohta Observer

Riippuvuuksien eliminointi observer-suunnittelumallilla

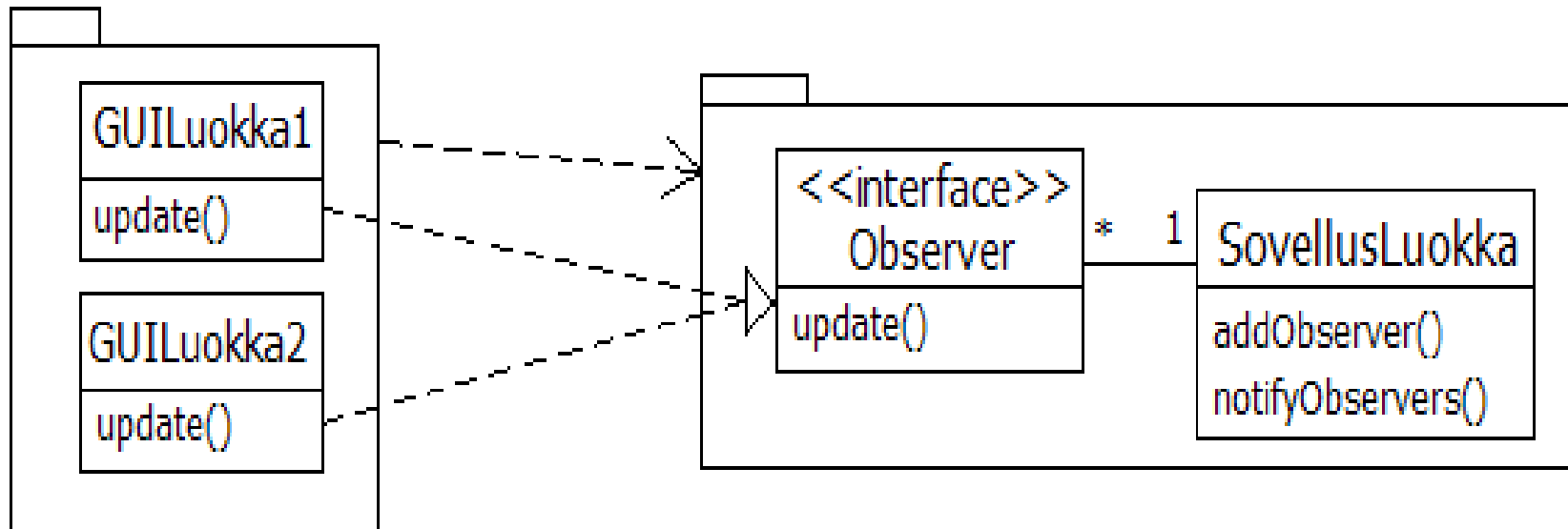
- Määritellään rajapinta, joka sisältää käyttöliittymäluokan päivitysmetodin `update()`, jota sovellusluokka kutsuu
 - Alla rajapinnalle on annettu nimeksi *Observer*
- Käyttöliittymäluokka toteuttaa rajapinnan, eli käytännössä toteuttaa `update()`-metodin haluamallaan tavalla
- Sovellusluokalle riittää 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*/
}
```