Ohjelmistojen mallintaminen

Luento 12, 10.12.

Tänään luvassa kertausta ja hieman uuttakin asiaa

- Kurssilla on puhuttu erinäisistä ohjelmistojen tuottamiseen liittyvistä asioista
- Ohjelmiston tuotantoon liittyy erilaisia *vaiheita*
    - Vaatimusmäärittely
    - Suunnittelu
    - Toteutus
    - Testaus
- Joskus (perinteisesti) vaiheet tehdään peräkkäin
    - Tällöin puhutaan *vesiputousmallin* mukaan etenevästä ohjelmistotuotantoprosessista
- Vaihtoehtoinen tapa on työskennellä iteratiivisesti, eli toistaa vaiheita useaan kertaan peräkkäin ja limittäin
    - *Ketterä ja iteratiivinen* ohjelmistotuotanto
- Ketteryys on valtaamassa alaa mutta perinteiset tavat myös käytössä
- Tehdään ohjelmia miten tahansa, **määrittelyn ja suunnittelun tukena tarvitaan malleja**

UML, mallit ja menetelmät

- UML tarjoaa runsaasti erilaisia kaavioita mallinnuksen tueksi, esim.
  - Käyttötapauskaaviot
  - Luokka- ja oliokaaviot
  - Sekvenssikaaviot
  - Pakkauskaaviot
- Kaikki nämä perustuvat *olioparadigmaan*, jossa ohjelman nähdään koostuvan joukosta keskenään kommunikoivia olioita
  - Useita eri kaaviotyyppejä: *mallinnuksessa* tarvitaan erilaisia *näkökulmia*
- Milloin ja miten UML:ää tulisi käyttää?
  - **Menetelmät** vastaavat tähän
  - Oliomenetelmiäkin useita, yleiset linjat selvät, hieman eri koulukuntia erityisesti oliosuunnittelun suhteen
- Oliosuunnittelun tehtävä on löytää/keksiä oliot ja niiden operaatiot siten, että ohjelma toimii niinkuin vaatimuksissa halutaan
  - Erittäin haastavaa: enemmän ”art” kuin ”science”

Oliosuunnittelusta

- Kurssilla on tarkasteltu muutamia yleisiä oliosuunnittelun periaatteita
  - **Single responsibility** eli luokilla vain yksi vastuu
  - **Program to an interface, not to concrete implementation**, eli suosi rajapintoja
  - **Favor composition over inheritance**, eli älä väärinkäytä perintää
- Merkki huonosta suunnittelusta: **koodihaju (engl. code smell)**
- Lääke huonoon suunnitteluun/koodihajuun: **refaktorointi**
  - Muutetaan koodin rakennetta parempaan suuntaan muuttamatta toiminnallisuutta
- Mietittiin Kirjasto-esimerikin yhteydessä katsottiin hieman **vastuupohjaista (responsibility driven)** oliosuunnittelutekniikka
  - Noudattaa kaikkia ym. yleisiä oliosuunnitteluperiaatteita
- Viime tiistaina demottiin **Test Driven Development** -menetelmää, joka yhdistää testauksen, ohjelmoinnin ja suunnittelun
  - TDD:tä lisää syksyn 2011 kurssilla *Ohjelmistokehitys*

Mikä on tärkeintä kurssilla/kokeessa?

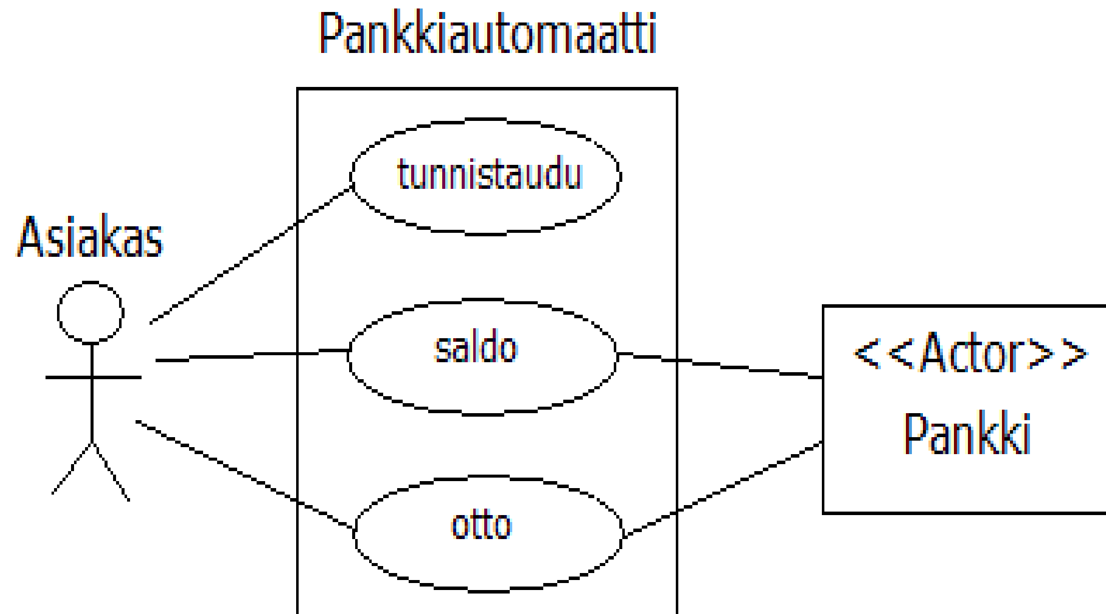
- Kokonaiskuva: *Mitä? Miksi? Milloin? Missä? Miten? (esseepotentiaalia)*
- UML:sta ylivoimaisesti tärkeimpiä ovat
  - Luokkakaaviot ja edellisten tukena oliokaaviot
  - Sekvenssikaaviot
  - Käyttötapauskaaviot
- Käyttötapausmallinnus myös tärkeää, ks. seuraavat kalvot
- *Käsiteanalyysi*, eli määrittelytason luokkamallin muodostaminen tekstistä
- Määrittelystä suunnitteluun eli *oliosuunnittelu*:
  - Luokkamallin tarkentuminen, uudet teknisen tason luokat
  - Toiminnan kuvaaminen sekvenssikaaviona
- *Takaisinmallinnus*, eli valmiista koodista luokka- ja sekvenssikaavioiden teko
- Kokeeseen ei tule Kirjastoesimerkin kaltaista isoa olisuunnitteluesimerkkiä, yleiset oliosuunnitteluperiaatteet ja motivaatio niiden takana on osattava
- Nippelitason detaljien sijasta kokeessa enemmän arvostetaan sovellusosaamista
- Kokeeseen saa tuoda mukanaan ***itse tehdyn, käsin tehdyn*** yhden A4 arkin (saa olla 2-puolinen) lunttilapun. Lunttilapun teossa siis ei saa käyttää kopiokonetta tai tietokonetta

Käyttötapausmalli

- Kertausta muutamasta käyttötapausmallin ydinkohdasta

Käyttötapaukset ja käyttötapauskaaviot

- Pankkiautomaatin käyttötapaukset ovat *tunnistaudu, saldo* ja *otto*
- Käyttötapausten *käyttäjät* (engl. actor) eli toimintaan osallistuvat tahot ovat *Asiakas* ja *Pankki*
- Käyttötapausten ja käyttäjien suhdetta kuvaa UML:n **käyttötapauskaavio**
  - Tikku-ukolle vaihtoehtoinen tapa merkitä käyttäjä on laatikko, jossa stereotyyppi eli tarkenne <<actor>>
  - Käyttötapausellipsiin yhdistetään viivalla kaikki sen käyttäjät
  - Kuvaan ei siis piirretä nuolia!



Käyttötapaukset ja käyttötapauskaaviot

- Huomattavaa on, että koska käyttötapausmallia käytetään vaatimusmäärittelyssä, *ei ole syytä puuttua järjestelmän sisäisiin yksityiskohtiin*
  - Pankkiautomaatin sisäiseen toimintaan ei oteta kantaa
  - Mitään automaatin sisäistä (esim. tietokantaa tai tiedostoja) ei merkitä kaavioon
  - *Tarkastellaan ainoastaan, miten automaatin toiminta näkyy ulospäin*
- Itse käyttötapauksen sisältö kuvataan *tekstuaalisesti*, käyttäen esim. Alistair Cockburnin käyttötapauspohjaa
- Käyttötapauskaavio toimii hyvänä yleiskuvana, mutta vasta tekstuaalinen kuvaus määrittelee toiminnan tarkemmin
  - Seuraavalla sivulla käyttötapauksen *Otto* tarkennus
- Seuraavan sivun käyttötapauskuvaus ei ota kantaa käyttöliittymään
- Seuraavana vaiheena saattaisi olla tarkemman, käyttöliittymäspesifisen käyttötapauksen kirjoittaminen

Käyttötapaus 1: otto

Tavoite: asiakas nostaa tililtään haluamansa määrän rahaa

Käyttäjät: asiakas, pankki

Esiehto: kortti syötetty ja asiakas tunnistautunut

Jälkiehto: käyttäjä saa tililtään haluamansa määrän rahaa

Jos saldo ei riitä, tiliä ei veloiteta

Käyttötapauksen kulku:

1 asiakas valitsee otto-toiminnon

2 automaatti kysyy nostettavaa summaa

3 asiakas syöttää haluamansa summan

4 pankilta tarkistetaan riittääkö asiakkaan saldo

5 summa veloitetaan asiakkaan tililtä

6 kuitti tulostetaan ja annetaan asiakkaalle

7 rahat annetaan asiakkaalle

8 pankkikortti palautetaan asiakkaalle

Poikkeuksellinen toiminta:

4a asiakkaan tilillä ei tarpeeksi rahaa, palautetaan kortti asiakkaalle

Luokkakaaviot

- Selvennystä muutamaan hienoista hämmennystä herättäneeseen luokkakaavioihin liittyvään aiheeseen

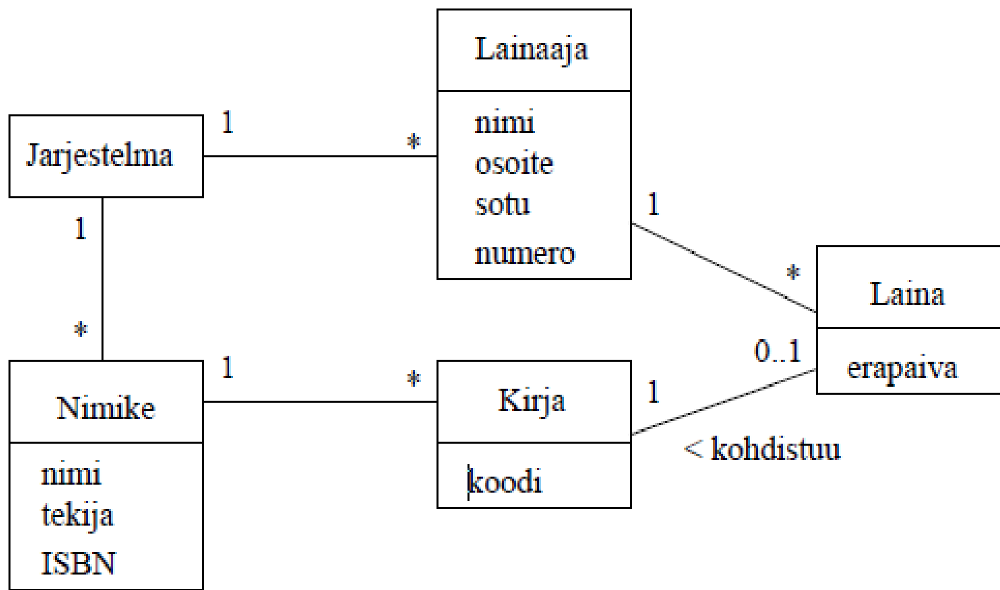
Kahden abstraktiotason luokkamalleja

- Olemme huomanneet, että luokkamallia käytetään kahdella hieman erilaisella abstraktiotasolla
- *Määrittelytasolla luokkamalli kuvailee sovellusalueen käsitteitä ja niiden suhteita*
  - Muodostetaan usein käsiteanalyysin avulla
  - Käytetään nimitystä *kohdealueen luokkamalli* (engl. domain model)
- *Suunnittelutasolla määrittelyvaiheen luokkamalli tarkentuu*
  - Luokat ja yhteydet tarkentuvat
  - Mukaan tulee uusia teknisen tason luokkia
- Suunnittelutason luokkamalli muodostuu *oliosuunnittelun* myötä
  - Oliosuunnittelussa kannattaa noudattaa hyväksi havaittuja oliosuunnittelun periaatteita (mm. single responsibility...)
  - *Suunnittelumallit* (engl. design patterns) voivat tarjota vihjeitä suunnittelun etenemiseen, näitä ei kurssilla juurikaan käsitelty
  - Luovuus, kokemus, tieto ja järki tärkeitä suunnittelussa

Kahden abstraktiotason luokkamalleja

- Malleista tulee hieman erilaisia käsitteellisellä- ja suunnittelutasolla
  - Esim. onko Monopolin Pelaaja-luokka ”ihmisen malli” vai Javalla toteutettava olio?
  - Tämä aiheuttaa joskus hämminkiä
- Eli se, minkälainen mallista tulee, riippuu mallintajan näkökulmasta
- Tämän takia yleensä pelkkä kaavio ei riitä: tarvitaan myös tekstiä (tai keskustelua), joka selvittää mallinnuksen taustaoletuksia
  - Tärkeintä, että kaikki asianosaiset tietävät taustaoletukset
  - Kun taustaoletukset tiedetään ja rajataan, löytyy myös helpommin ”oikea” tapa (tai joku oikeista tavoista) tehdä malli
- Mallinnustapaan vaikuttaa myös mallintajan kokemus ja ”orientaatio”
  - Kokenut ohjelmoija ”näkee kaiken koodina” ja ajattelee väistämättä teknisesti myös abstraktissa mallinnuksessa
  - Tietokantaihminen taas näkee kaiken tietokannan tauluina jne.

Määrittelytaso vs. suunnittelutaso



- Viime viikolta:
- Kirjasto tarkentui abstraktista määrittelytason luokkamallista oliosuunnittelun seurauksena suunnittelutason luokkamalliksi

kayttoliittyma

<<ohjaus>> LisaaKirja: lisaaKirja, luoTunniste

<<ohjaus>> LisaaNimike: lisaaNimike

<<ohjaus>> PalautaKirja: poistaLaina

<<ohjaus>> LainaaKirja: tunnistaLainaaja, tunnistaKirja, kirjaaLaina

<<ohjaus>> LisaaLainaaja: lisaaLainaaja

Nimikkeet

Nimike

Kirja

Lainaajatiedot

Lainaaja

Laina

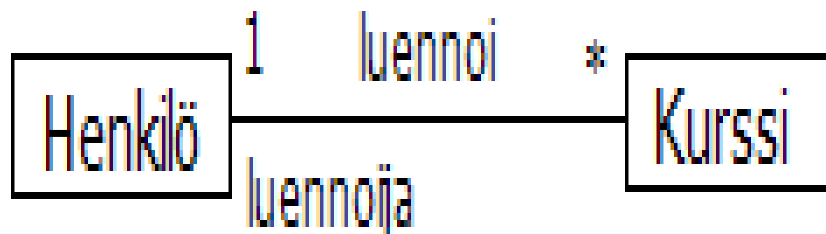
Yhteys ja kompositio

- Erilaiset yhteystyypit ovat herättäneet monissa epäselvyyttä
- Yhteydellähän tarkoitetaan ”rakenteista” eli jollakin tavalla pitempiaikaista suhdetta kahden olion välillä
- Jos kahden olion välillä on tällainen suhde, merkitään luokkakaavioon yhteys (ja yhteyteen liittyvät nimet, roolit, kytkentärajoitteet ym.)
- Toteutusläheisissä malleissa voidaan ajatella, että luokkien välillä on yhteys *jos luokalla on olioarvoinen attribuutti*

```
class Henkilö {
 ...
}
```

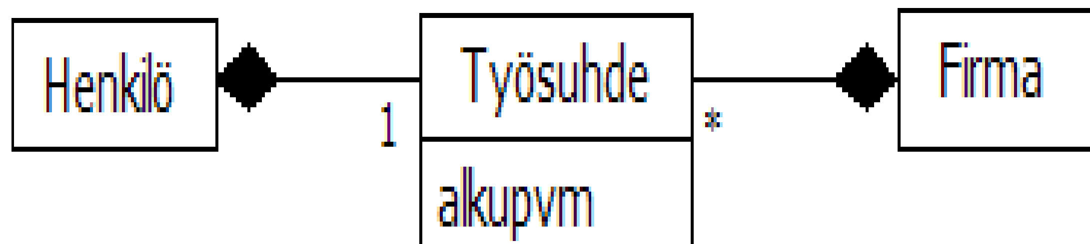
```
class Kurssi{
    Henkilo luennoija;
    …
}
```

- Jokaisella Kurssi-oliolla on yhteys yhteen Henkilö-olioon, joka on Kurssin suhteen roolissa luennoija



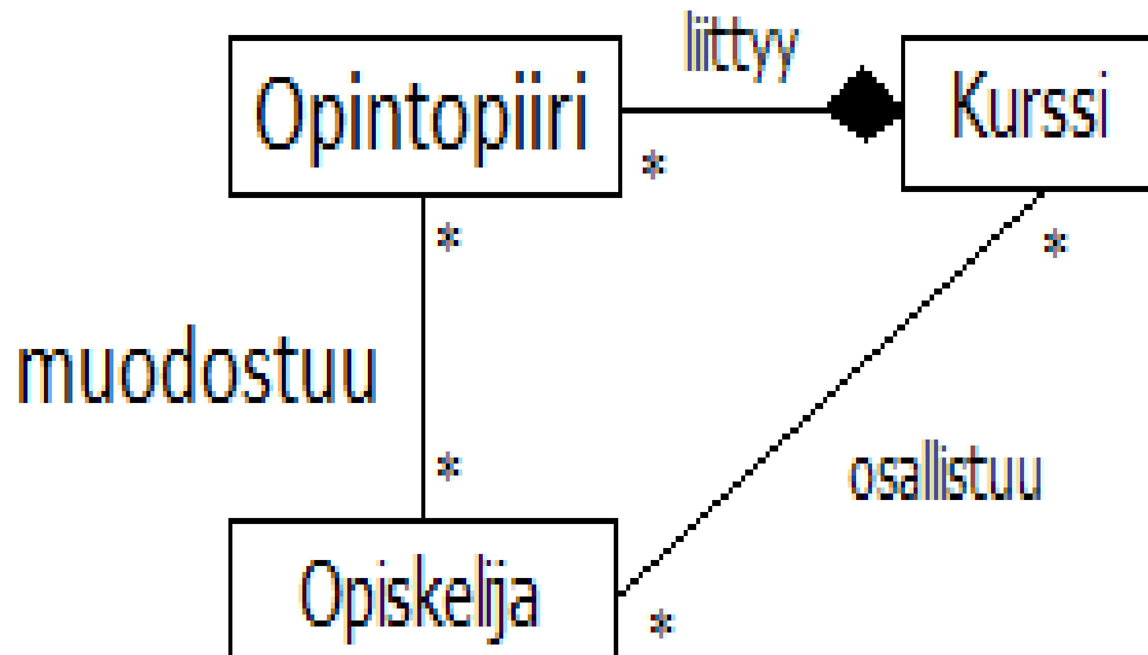
Yhteys ja kompositio

- Normaali yhteys, eli viiva on varma valinta sillä se ei voi olla ”väärin”
- Usein määrittelyvaiheen luokkamalleissa käytetäänkin vain normaaliyhteyksiä
- Yhteyteen voidaan laittaa tarvittaessa *navigointisuunta* eli nuoli toiseen päähän, tällöin vain toinen pää tietää yhteydestä
- Jos joku yhteyden osapuolista on *olemassaoloriippuva* yhteyden toisesta osapuolesta, voidaan käyttää *kompositioita* eli mustaa salmiakkia
  - Työsuhde liittyy koko olemassaolonsa tiettyyn Henkilö-olioon ja tiettyyn Firma-olioon
  - Jos Firma tai Henkilö häviävät, niin häviää myöskin Työsuhde
  - *Merkitään salmiakki niihin olioihin, joista olemassaolo riippuu*
  - Ei olisi väärin ilmaista yhteyksiä normaaleina viivoina mutta musta salmiakki tarjoaa hyödyllistä lisätietoa yhteyden laadusta



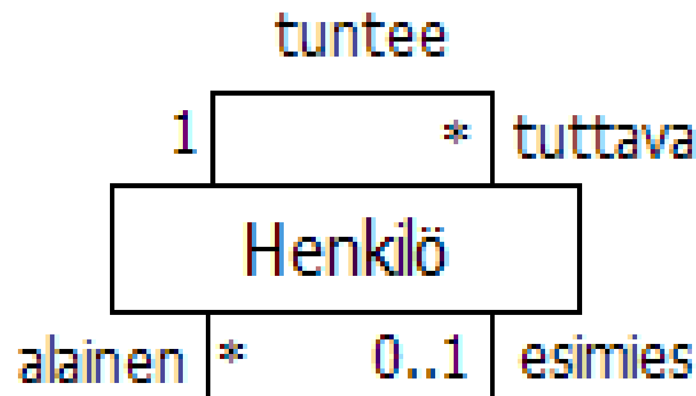
Yhteys ja kompositio

- Opintopiiri liittyy tiettyyn kurssiin
  - Kurssin loputtua myös opintopiiri lakkaa olemasta
  - Olemassaoloriippuvuus, eli voidaan käyttää kompositiota
- Joukko opiskelijoita muodostavat opintopiirin
  - Opiskelija voi kuulua myös useampaan opintopiiriin
  - Normaali yhteys



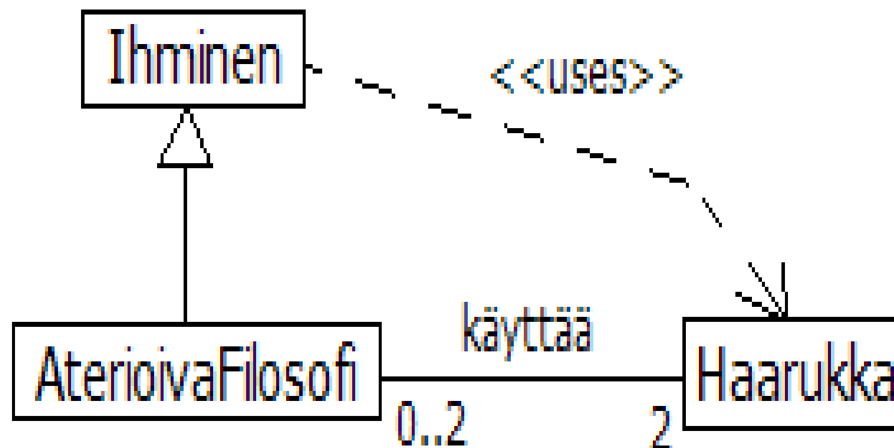
Rekursiivinen yhteys

- Saman luokan olioita yhdistävät yhteydet ovat joskus haasteellisia
- Henkilö tuntee useita henkilöitä
- Henkilöllä on mahdollisesti yksi esimies ja ehkä myös alaisia
- Yhteys *tuntee* liittää Henkilö-olioon useita Henkilö-olioita *roolissa tuttava*
- Henkilö-oliolla voi olla yhteys Henkilö-olioon, joka on *roolissa esimies*
  - Eli Henkilöön voi liittyä esimies
- Henkilö-oliolla voi olla yhteys useaan Henkilö-olioon, jotka ovat *roolissa alainen*
  - Eli Henkilöllä voi olla useita alaisia
- Kuvan alempi yhteys siis sisältääkin kaksi yhteyttä: esimieheen ja alaisiin
- **Rekursiivisten yhteyksien kanssa on syytä käyttää roolinimiä**, pelkkä yhteyden nimi ei yleensä riitä ilmaisemaan selkeästi yhteyden laatua



Yhteys vai riippuvuus?

- Useimmat normaalit ihmiset käyttävät haarukkaa syödessään
  - Ihmisen ja haarukan suhdetta ei kannata ehkä mallintaa yhteytenä
  - Jos suhde ylipäätään on syytä mallintaa, *riippuvuus* riittää
  - Riippuvuus merkitään katkoviivanuolena siihen luokkaan mistä ollaan riippuvaisia
  - Riippuvuuden laatu voidaan tarkentaa *stereotyypillä*
- Aterioiva filosofi on myös ihminen mutta tavoiltaan hieman erikoinen
  - Aterioiva filosofi käyttää tasan kahta haarukkaa
  - Filosofin ja haarukoiden suhteen voisi mallintaa yhteyden avulla

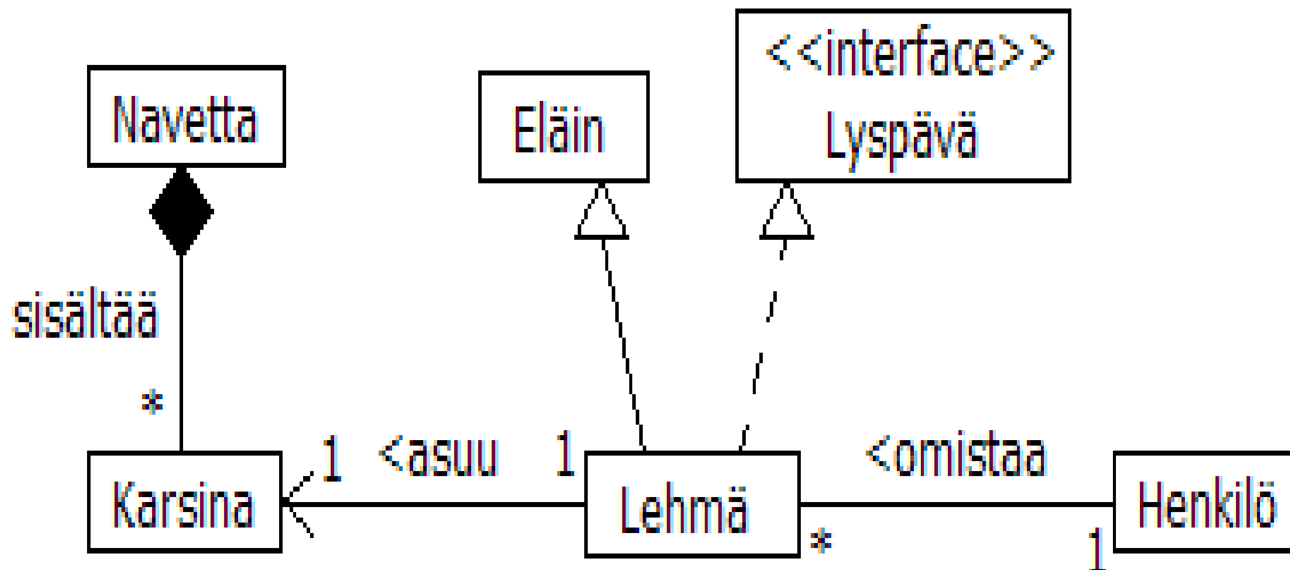


Yhteys vai riippuvuus?

- Riippuvuus on jotain *normaalia yhteyttä heikompaa*, esim. jos luokan jollakin metodilla on olioarvoinen parametri, voidaan käyttää riippuvuutta (aiemmin luennolla esimerkki AutotonHenkilö riippu luokasta Auto)
  - Riippuvuus ei siis ole rakenteinen vaan ajallisesti hetkellinen, esim. yhden metodikutsun ajan kestävä suhde
- Riippuvuuden ja yhteyden välinen valinta on joskus näkökulmakysymys
  - Miten kannattaisi esim. mallintaa Henkilön ja Vuokra-auton suhde?
  - Vaikka kyseessä voi olla lyhytaikainen suhde, lienee se järkevintä kuvata esim. autovuokraamon tietojärjestelmän kannalta normaalina yhteytenä
- Riippuvuuksia ei kannata välttämättä edes merkitä, ainakaan kaikkia
- Jotkut mallintajat eivät merkitse riippuvuuksia ollenkaan
- **Käytä siis riippuvuuksia hyvin harkiten**
- Jos luokkien välillä on yhteys, ei niille tarvitse enää merkitä riippuvuutta sillä yhteyden olemassaolo tarkoittaa että luokat ovat myös riippuvaisia toisistaan

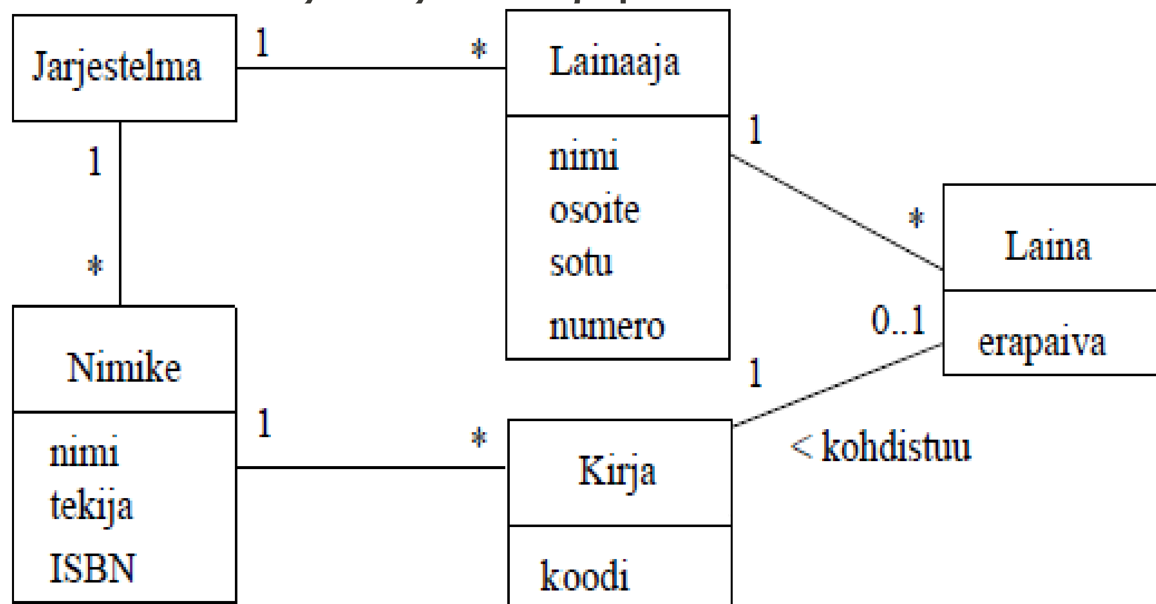
Nuolenpäät luokkakaaviossa

- Lehmä on Eläin-luokan *aliluokka:* valkoinen kolmio yliluokan päässä, normaali viiva
- Lehmä *toteuttaa* Lypsävä-*rajapinnan:* valkoinen kolmio rajapinnan päässä, katkoviiva
- Navetta sisältää karsinoita jotka ovat olemassaoloriippuvaisia navetasta: musta salmiakki olemassaolon takaavan luokan päässä
- Lehmä asuu karsinassa, lehmä tuntee karsinansa, mutta karsina ei lehmää: yhteydessä navigointinuoli lehmästä karsinaan päin
- Lehmä ja omistaja (joka on Henkilö-olio) tuntevat toisensa: normaali yhteys ilman nuolia

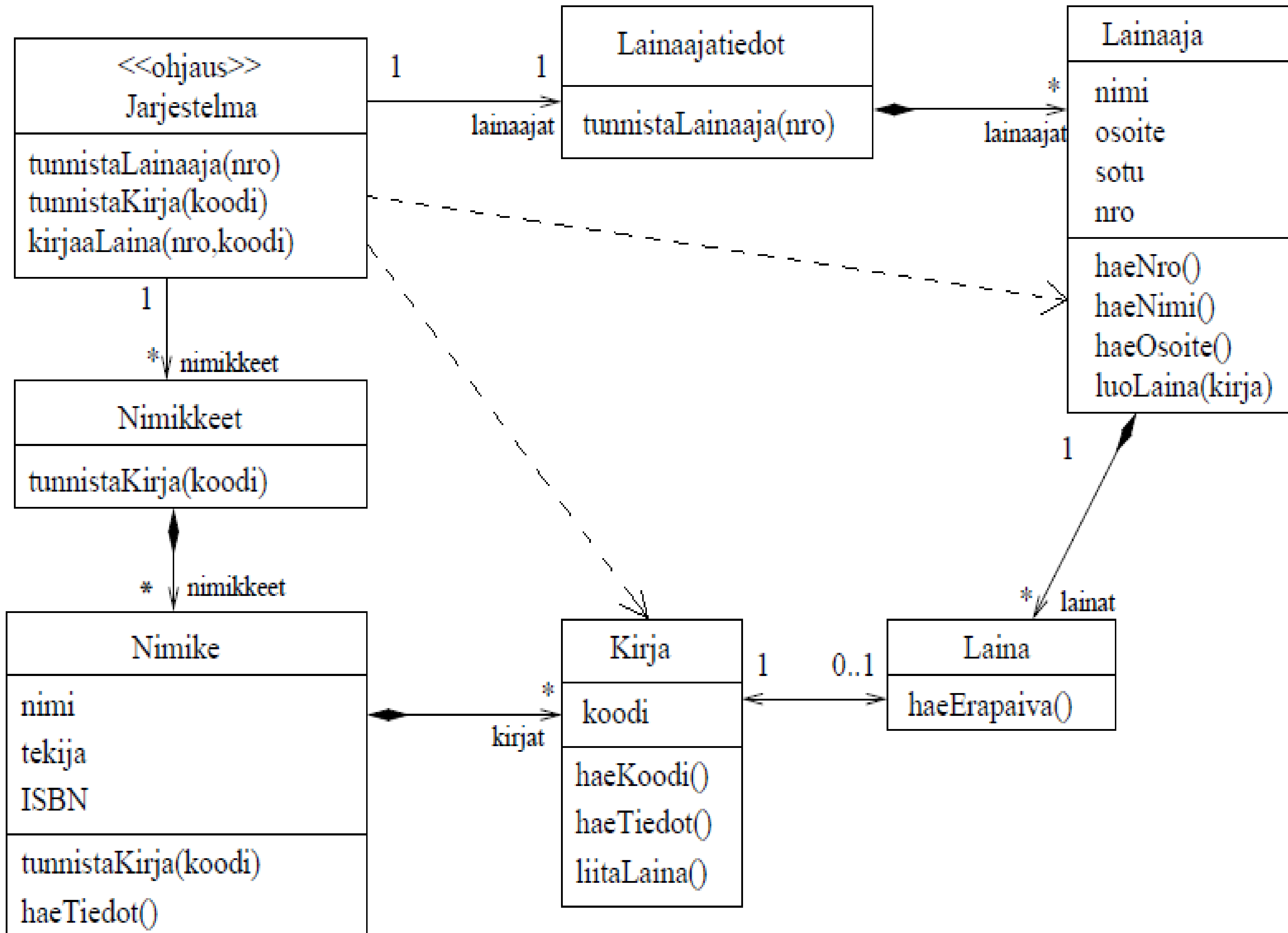


Navigointisuuntien merkitseminen

- Edellisellä sivulla merkitsimme Karsinan ja Lehmän väliseen yhteyteen navigointisuunnan
- Navigointisuuntia ei välttämättä merkitä ollenkaan, monisteesta:
  - *Navigointimäärittelyt kuuluvat matalan tason ohjelmointiläheiseen kuvaukseen. Korkeamman abstraktiotason kuvauksissa ne jätetään teknisen toteutuksen yksityiskohtina usein kokonaan kuvaamatta*
- Kirjastoesimerkin määrittelyvaiheen luokkakaavioon ei merkitty navigointisuuntia (ks. alla)
- Toteutustason luokkakaaviossa taas navigointisuunnat merkittiin kaikkiin yhteyksiin (ks. seuraava sivu)
  - Kaksisuuntaisiin yhteyksiin jopa merkittiin nuolet molempiin päihin



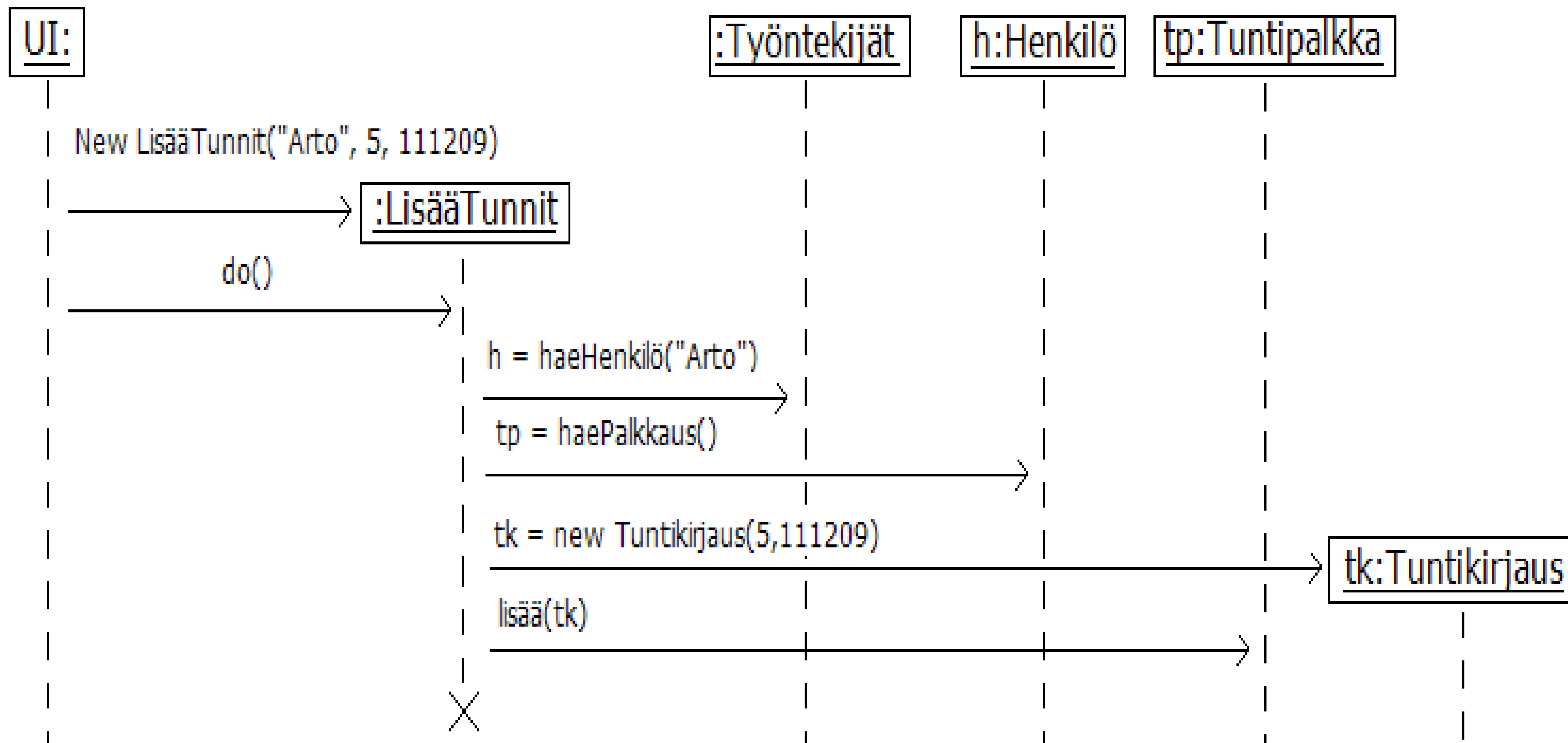
Tarkassa suunnittelutason luokkakaaviossa kaikkien yhteyksien navigointisuunnat merkittynä



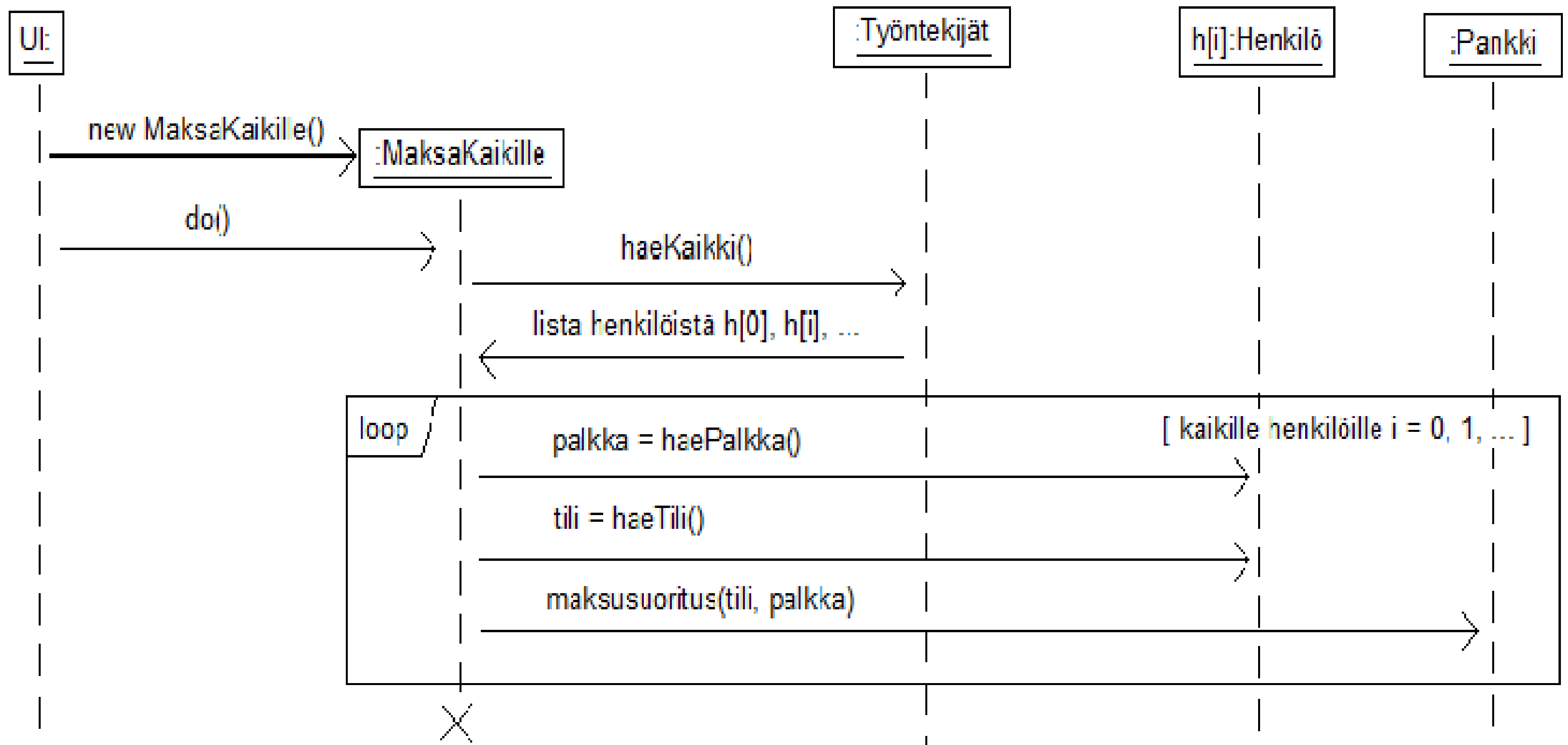
Sekvenssikaavioista

- Luokkakaavio ei kerro mitään olioiden välisestä vuorovaikutuksesta
- Vuorovaikutuksen kuvaamiseen paras väline lienee sekvenssikaavio
  - Kurssilla vilkaistiin myös kommunikaatiokaavioita, niitä ei kuitenkaan kokeessa tarvitse osata
- Sekvenssikaavioiden peruskäyttö on melko suoraviivaista
- Sekvenssikaavioiden ”vaikeudet” liittyvät toisto-, ehdollisuus- ja valinnaisuuslohkoihin
  - Näiden käyttökelpoisuus on kuitenkin rajallinen, ja niitä kannattaa käyttää harkiten
- Pari esimerkkiä, joista käy ilmi oleellinen sekvenssikaavioihin liittyvä
  - Molemmat ovat monisteen liitteenä A olevasta kurssille varsinaisesti kuulumattomasta osasta *”Oliosuunnitteluesimerkki: Yrityksen palkanlaskentajärjestelmä”*

- Olio UI luo ensin luokan LisääTunnit-olion ja kutsuu sen do()-metodia
- do():n suoritus kutsuu Työntekijät-luokan metodia haeHenkilö
- Paluuarvona olevan Henkilö-olion *h* metodia haePalkkaus kutsutaan
- Luodaan uusi Tuntikirjaus-olio *tk*, joka lisätään haePalkkaus()-metodin palauttamalle oliolle *tp* metodikutsulla lisää()
- Lopuksi LisääTunnit-luokan olio tuhoutuu



- Olio UI luo ensin luokan MaksaKaikille-olion ja kutsuu sen do()-metodia
- do():n suoritus kutsuu Työntekijät-luokan metodia haeKaikki() joka palauttaa listan Henkilö-olioita
    - Meritään palautettua oliolistaa h[0], h[1], ..., eli lyhyesti h[i], i=0,1,2, ...
- Loopissa jokaiselle Henkilö-oliolle h[i] kutsutaan metodeja haePalkka() ja haeTili()
    - Sekä tehdään Pankki-oliolle metodikutsu maksusuoritus(), jonka parametreina on henkilöltä kysytyt palkka ja tilinumero



Oliosuunnittelu ja koe

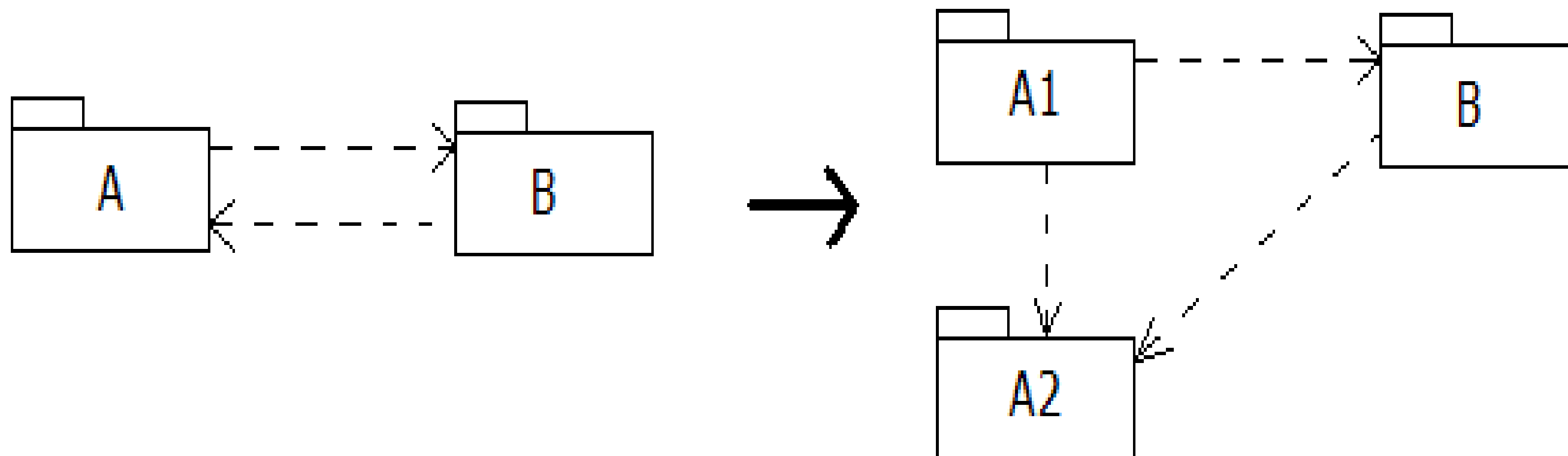
- **Eli mitä aiheesta pitää osata kokeessa?**
  - Yleiset periaatteet
  - Periaatteiden soveltaminen yksinkertaisessa tilanteessa
    - Esim. laskareiden tapaan osattava tunnistaa, että tietty periaate rikkoutuu
  - Pintatason ymmärrys miksi periaatteet ovat olemassa ja milloin niitä saa rikkoa

Hieman lisää suunnittelutason asiaa

- Kertauksen ja täsmennyksien jälkeen, *jos aikaa vielä jää*, katsotaan erästä oliosuunnitteluun liittyvää uutta asiaa, syklisten riippuvuuksien purkamista
  - Erityishuomio sovelluslogiikan eristämisessä käyttöliittymästä
- Monisteen liitteenä A on laajempi oliosuunnitteluesimerkki, johon tutustumista suosittelen kaikille oliosuunnittelusta viehättyneille

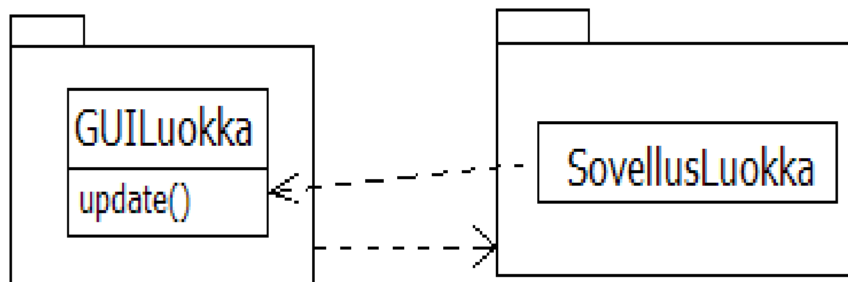
Ohjelmiston arkkitehtuuri ja riippuvuudet

- Ohjelmiston arkkitehtuurin yhteydessä mainittiin, että ohjelman osat on syytä jakaa mahdollisimman riippumattomiin pakkauksiin
  - Esim. kerrosarkkitehtuurin pakkauksissa riippuvuuksia vain yhteen suuntaan
- Pahin mahdollinen tilanne on *riippuvuussykli*
  - Eli Pakkaus A riippuu pakkauksesta B ja päinvastoin
- Riippuvuussyklistä voidaan *joissain* tapauksissa päästä eroon erittelemällä toinen pakkauksista kahdeksi pakkaukseksi
  - Eli erotetaan pakkauksesta A ne osat joista B riippuu omaksi pakkaukseksi A2
  - Loput pakkauksesta A muodostavat pakkauksen A1 joka riippuu B:stä ja uudesta pakkauksesta



Riippuvuuksien eliminointi

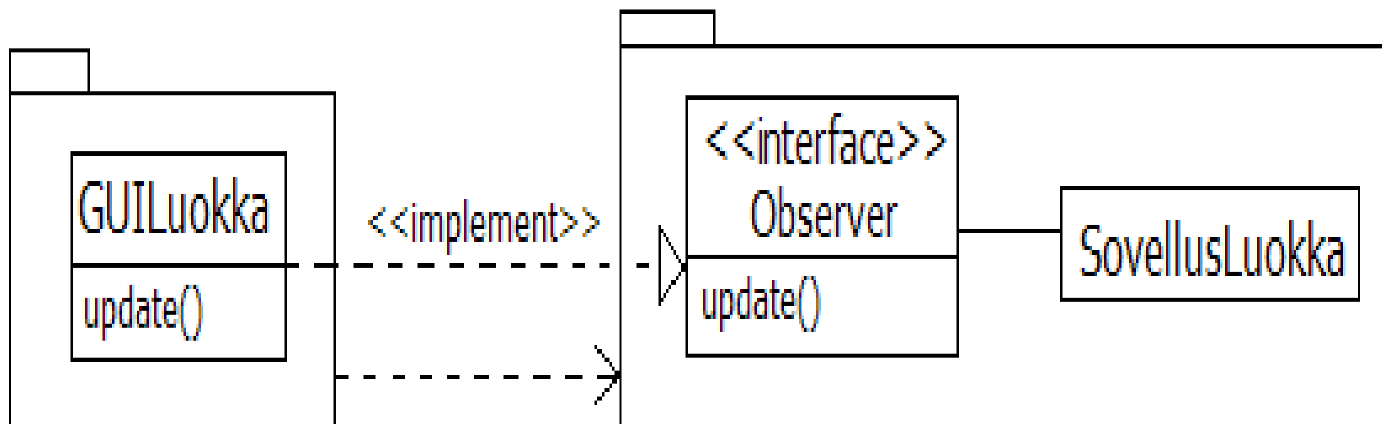
- Edellisen sivun ratkaisu toimii joskus, mutta usein vaaditaan ovelampi ratkaisu
- Kerrosarkkitehtuurissa törmätään usein tilanteeseen, jossa sovelluslogiikan on kerrottava käyttöliittymälle jonkin sovellusolion tilan muutoksesta, jotta käyttöliittymä näyttäisi koko ajan ajantasaista tietoa
- Tästä 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 hieman yllättävällä tavalla rajapintaluokkaa käyttäen
  - Ratkaisu seuraavalla sivulla

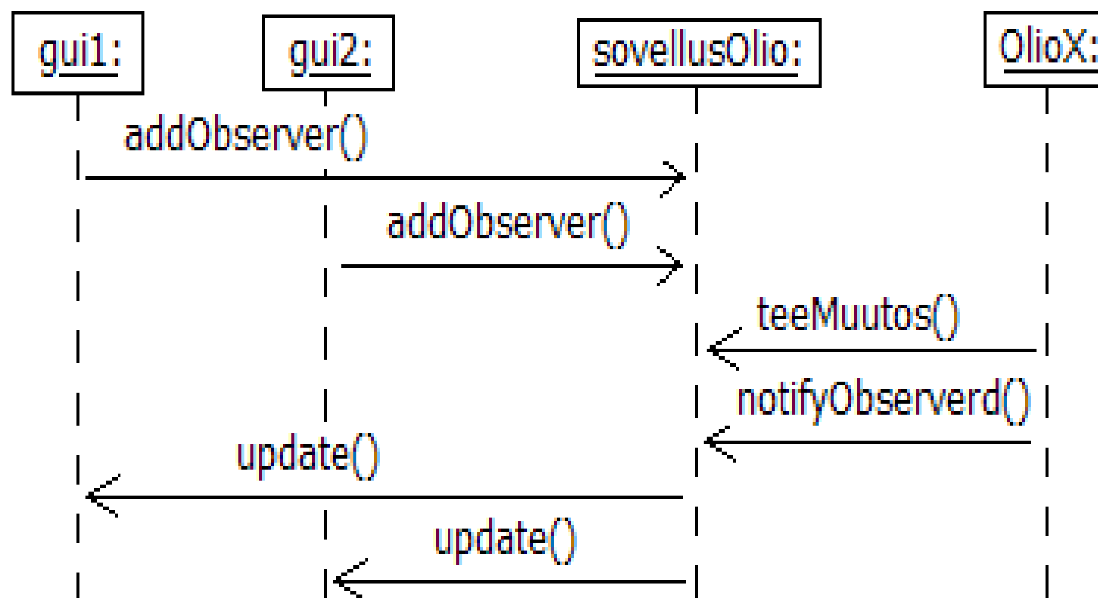
Riippuvuuksien eliminointi rajapinnan avulla

- Määritellään rajapinta, joka sisältää käyttöliittymäluokan päivitysmetodin update(), jota sovellusluokka kutsuu
  - Rajapinta määritellään *sovelluslogiikkapakkauksen sisälle*
  - 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

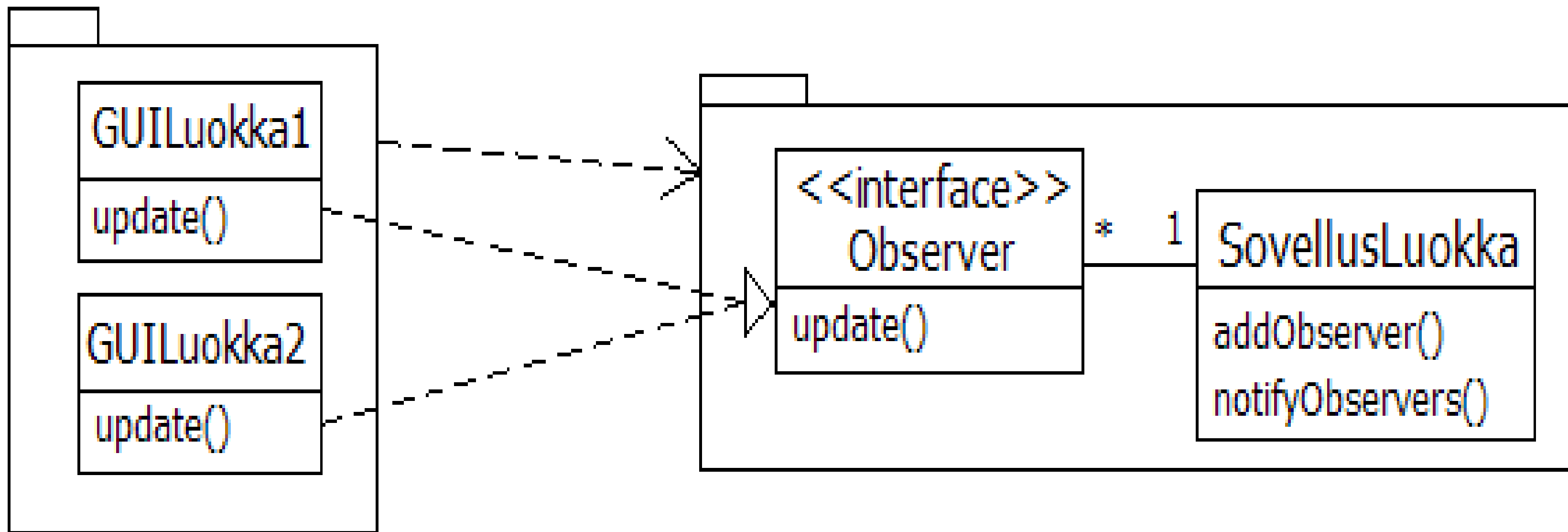


Tarkkailijaperiaate

- Edellisen sivun ratkaisu hieman laajennetussa muodossa on suunnittelumalli nimeltään *tarkkailijaperiaate* (engl. observer pattern)
- 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, jonka parametrina voidaan tarvittaessa välittää muutostieto



Tarkkailijaperiaatteen luokkakaavio ja koodihahmotelma




```
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 Observer{

    void update(){
            /* päivitetään näyttöä */

    }

    /* muu koodi*/

}
```

Uncle Bobin ajatuksia UML:stä

- Kurssin lopuksi muutamia Uncle Bobin eli Robert Martinin viisaita ajatuksia kirjasta *UML for Java programmers*
- Ajatuksia lukiessa täytyy muistaa, että Martin on ketterien menetelmien erityisesti ns. Test Driven Development -suunnittelumenetelmän kannattaja ja toisenlaisiakin mielipiteitä UML:n käytöstä löytyy
  - Itse allekirjoitan Martinin mielipiteet oikeastaan sellaisenaan
- Toinen huomionarvoinen seikka, on se, että Martinin kirja käsittelee ainoastaan *suunnittelu- ja toteutustasoista UML:n käyttöä*, joten hänen ohjeensa eivät sinällään koske UML:n käyttöä käsitteellisen mallinnuksen (esim. määrittelytason) välineenä

Mallinnuksesta:

- We investigate designs with models when models are much cheaper to build than the real thing we are building
- We make use of UML when we have something definitive we need to test, and when using UML to test it is cheaper than using code to test it
  - *Something to test viittaa tässä johonkin suunnitteluratkaisuun*

- UML is enormously convenient for communicating design concepts among software developers. A lot can be done with a small group of developers at a whiteboard.
- UML is very good for communicating focused design ideas
- UML can be used for creating road maps of large software structures. Such road maps give developers quick way to find out which classes depend upon which others and provide a reference to the structure of the whole system
- UML diagrams need to be carefully considered. We don't want a thousand pages of sequence diagrams! Rather we want a few very salient diagrams that describe the major issues in system.
- Get into the habit of throwing UML diagrams away.
- Do not use a case tool or drawing program as a rule. There is a time and place for such tools, but most of your UML should be short lived.
- Some diagrams, however are useful to save: These are the diagrams that express a common design solution in your system. These are the diagrams that record designer intent in a way better that code can express it.
- There is no point hunting for these diagrams. The real useful diagrams will keep showing up over and over again.

- Alla esimerkki ”Bob Martinin hengessä tapahtuneesta” UML:n käytöstä:
  - Luukkaisen ja Matti ”Lohjan Kent Beck” Paksulan mallinnussessio lokakuisena perjantai-iltana. Kyseessä esiaste monisteen lisäosan esimerkistä

- How do we create UML diagrams?
    - Do we draw them in one brilliant flash of insight?
    - Do we draw first class diagrams and then sequence diagrams?
- The answer to these questions is a resounding NO! Anything humans do well they do by taking tiny little steps.
- You start with a little bit of dynamics. Then you explore what those dynamics imply to static relationships. You alter the static relationships according to the principles of good design. Then you go back and improve your dynamic diagrams. Each of these steps is tiny.
    - *Eli tässä Martin sanoo, että suunnittelua tehdään siten, että rakennetaan sekvenssikaavioita ja luokkakaavioita rinta rinnan*
    - *Principles of good design viittaa tuttuihin oliosuunnittelun periaatteisiin (Single responsibility, ym...)*
    - *Monisteen Kirjastoesimerkissä toimittiin tämän ohjeen mukaisesti!*
- Keep your diagrams simple and clean. UML diagrams are not source code and should not be treated as the place to declare every method, variable and relationship.
- Drawing UML diagrams can be very useful activity. It can also be a horrible waste of time.

- When to draw diagrams:
  - When several people need understand the structure of a particular part of the design because they are all going to be working on it simultaneously
  - When two or more people disagree on how a particular element should be designed and you want team consensus
  - When you just want to play with a design idea and the diagrams can help you think it through
  - When you need to explain the structure of some part of the code to someone or yourself
  - When it's close to the end of the project and your customer has requested them as a part of a documentation for others

- When not to draw diagrams:
  - your process tells you to
  - You feel guilty not drawing them or you think that's what good designers do. Good designers write code and draw diagrams only when it is necessary
  - To create comprehensive documentation of the design phase prior to coding. Such documents are almost never worth anything and consume immense amounts of time
  - For other people to code. True software architects participate in the coding of their designs, so they can be seen to lie in the bed they have made.

- Näihin sanoihin on hyvä lopettaa. Kiitoksia osallistumisesta ja onnea kokeeseen!