Ohjelmistojen mallintaminen

Luento 4, 12.11.

Kertausta: Olioperustainen ohjelmistokehitys

- Lähdemme siis oletuksesta, että kehitettävän järjestelmän voidaan ajatella koostuvan oliosta
- Tehdään koko ohjelmistokehitys *olioperustaisesti*, eli vaiheittain seuraavan kaavan mukaan:

1. Luodaan **määrittelyvaiheen oliomalli** sovelluksen käsitteistöstä
   - Mallin oliot ja luokat ovat rakennettavan sovelluksen kohdealueen käsitteiden vastineita
2. Suunnitteluvaiheessa tarkennetaan edellisen vaiheen oliomalli **suunnitteluvaiheen oliomalliksi**
   - Oliot muuttuvat yleiskäsitteistä teknisen tason olioiksi
   - Mukaan tulee olioita, joilla ei ole suoraa vastinetta reaalimaailman käsitteistössä
3. Toteutetaan suunnitteluvaiheen oliomalli jollakin **olio-ohjelmointikielellä**

Kertausta: Luokka- ja oliokaaviot

- Järjestelmän luokkarakennetta kuvaa **luokkakaavio** (engl. class diagram)
  - Mitä luokkia olemassa
  - Minkälaisia luokat ovat
    - attribuutit ja metodit
  - Luokkien suhteet toisiinsa
    - Erityisesti kuvataan **luokkien väliset yhteydet**
      - Henkilö *omistaa* auton
      - Opiskelija *osallistuu* kurssille

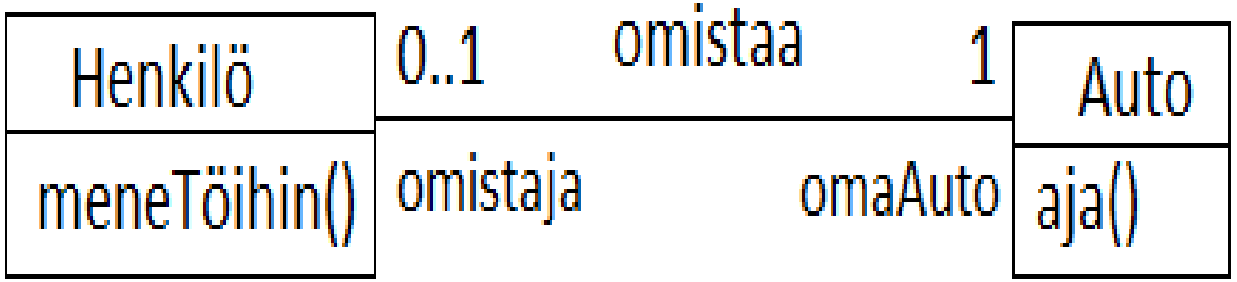
- Luokkakaavio kuvaa ikäänkuin kaikkia mahdollisia olioita, joita järjestelmässä on mahdollista olla olemassa
- **Oliokaavio** (engl. object diagram) taas kuvaa mitä olioita järjestelmässä on tietyllä hetkellä

Tuttu esimerkki

```
class Auto {
   void aja(){ System.out.println("liikkuu"); }
}

class Henkilo {
    Auto omaAuto;   // viite olioon, jonka tyyppinä Auto
    String nimi;
    Int ika;

    Henkilo(Auto a){  omaAuto = a;  }

    void meneTöihin(){
       omaAuto.aja();
   }
```

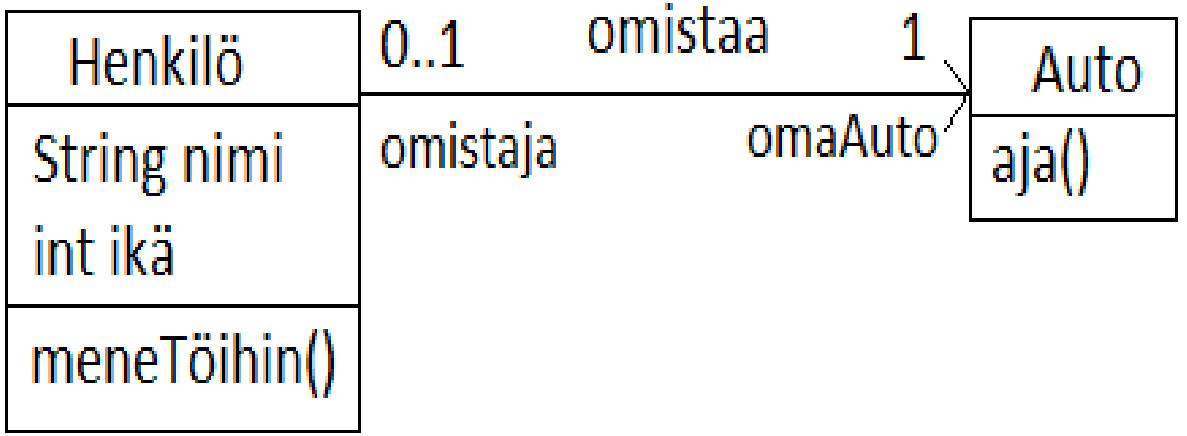
Nopea kertaus luokkakaaviosta

- Luokat laatikkoina
  - Luokan nimi, attribuutit ja metodit omissa osastoissa
  - Ainoastaan nimi pakko merkitä
  - Attribuuttien ja parametrien tyypit merkitään jos tarvetta
- Olioviitteitä ei merkitä luokkamäärittelyyn attribuutiksi jos kyseessä ”merkityksellinen” yhteys
  - omaAuto on olioviite jota ei merkitä luokkalaatikkoon
  - Nimi on String, eli oikeastaan olio. Merkitään attribuutiksi, sillä se että nimi on oikeasti olio ei nyt kiinnosta
- *Olioviitteet merkitään kaavioihin yhteyksinä*, eli viivoina laatikoiden välillä
  - Yhteyden nimi, roolit, kytkentärajoitteet...



Yhteyden navigointisuunta

- Auto-luokan koodista huomaamme, että auto-oliot eivät tunne omistajaansa
  - Henkilö-oliot taas tuntevat omistamansa autot Auto-tyyppisen attribuutin omaAuto ansiosta
- Yhteys siis on oikeastaan *yksisuuntainen,* henkilöstä autoon, mutta ei toisinpäin
- Asia voidaan ilmaista kaaviossa tekemällä yhteysviivasta nuoli
  - Käytetään nimitystä navigointisuunta
  - Nuolen kärki sinne suuntaan, johon on pääsy attribuutin avulla



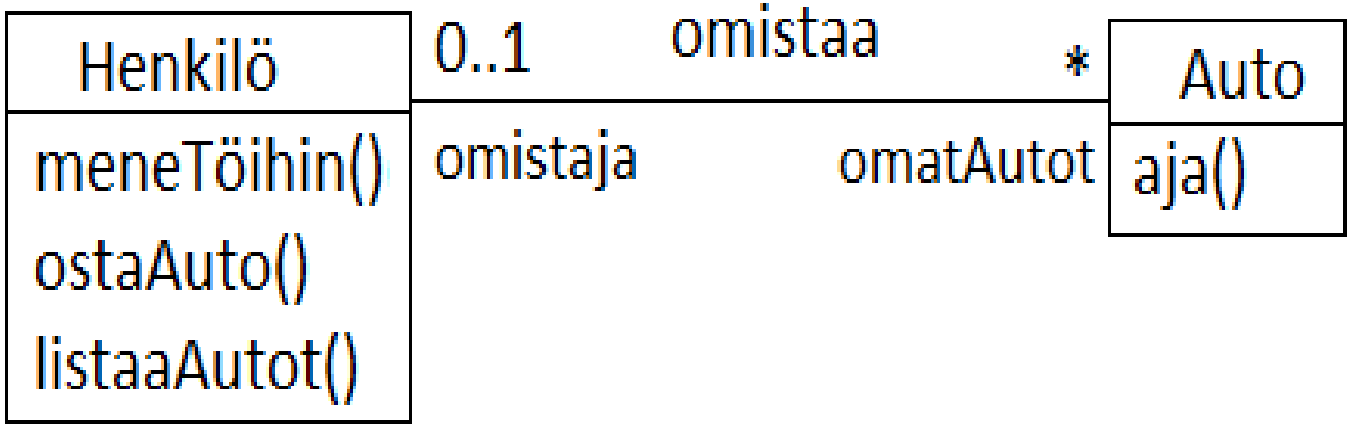
Yhteyden navigointisuunta

- Yhteyden navigointisuunnalla merkitystä lähinnä suunnittelu- ja toteutustason kaavioissa
  - Merkitään vain jos suunta tärkeä tietää
  - Joskus kaksisuuntaisuus merkitään nuolella molempiin suuntiin
  - Joskus taas nuoleton tarkoittaa kaksisuuntaista
- Määrittelytason luokkakaavioissa yhteyden suuntia ei yleensä merkitä ollenkaan

- Yhteyden suunnalla on aika suuri merkitys sille, kuinka yhteys toteutetaan kooditasolla
  - Vapaaehtoinen tehtävä: laajenna edellistä esimerkkiä niin, että Auto-olio tuntee omistajansa, ja että kuolleen henkilön auto voi saada uuden omistajan

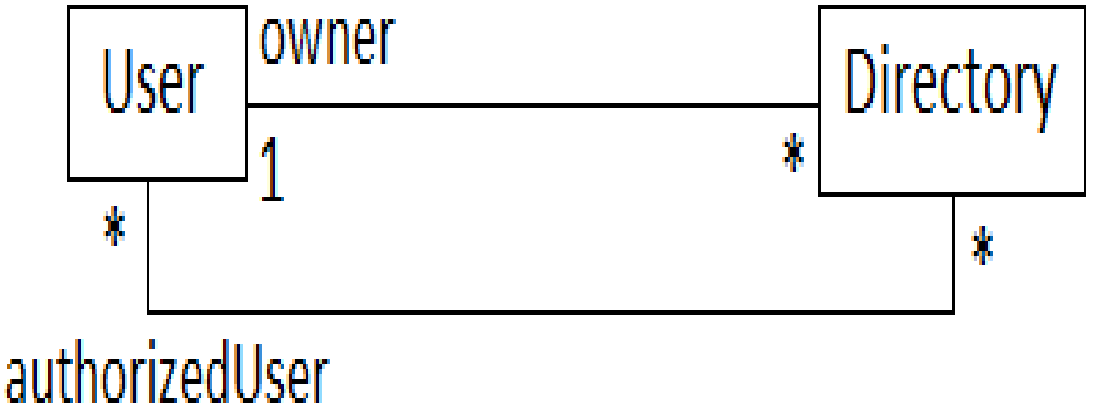
Kertaus monimutkaisemmasta yhteydestä

- Alla olevassa kaaviossa henkilö voi omistaa useita autoja
- Autolla on edelleen korkeintaan yksi omistaja
- Yhteen Henkilö-olioon voi siis liittyä monta Auto-olioa
  - kytkentärajoite * joka tarkoitaa 0...n
- Yhteen Auto-olioon liittyy 0 tai 1 Henkilö-olioa omistajan roolissa
  - kytkentärajoite 0..1
- Jos tilanne epäselvä: *piirrä oliokaavio*

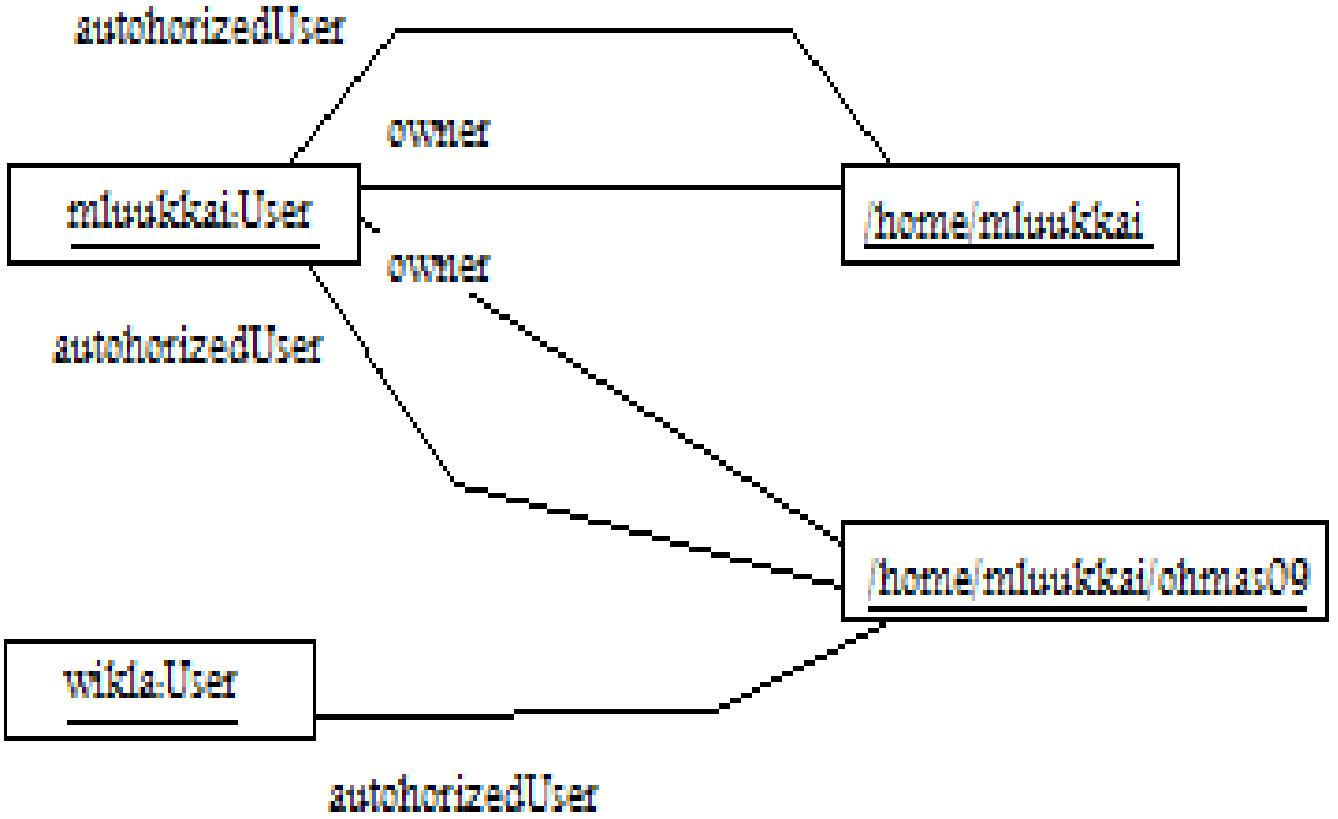


Useampia yhteyksiä olioiden välillä

- Esim. Linuxissa jokaisella hakemistolla on tasan yksi omistaja
  - Eli yhteen hakemisto-olioon liittyy roolissa owner tasan yksi käyttäjä-olio
- Jokaisella hakemistolla voi olla lisäksi useita käyttäjiä
  - Yhteen hakemistoon liittyy useita käyttäjiä roolissa authorizedUser
- Yksi käyttäjä voi omistaa useita hakemistoja
- Yhdellä käyttäjällä voi olla käyttöoikeus useisiin hakemistoihin
- Yhdellä käyttäjällä voi olla *samaan hakemistoon sekä omistus- että käyttöoikeus*

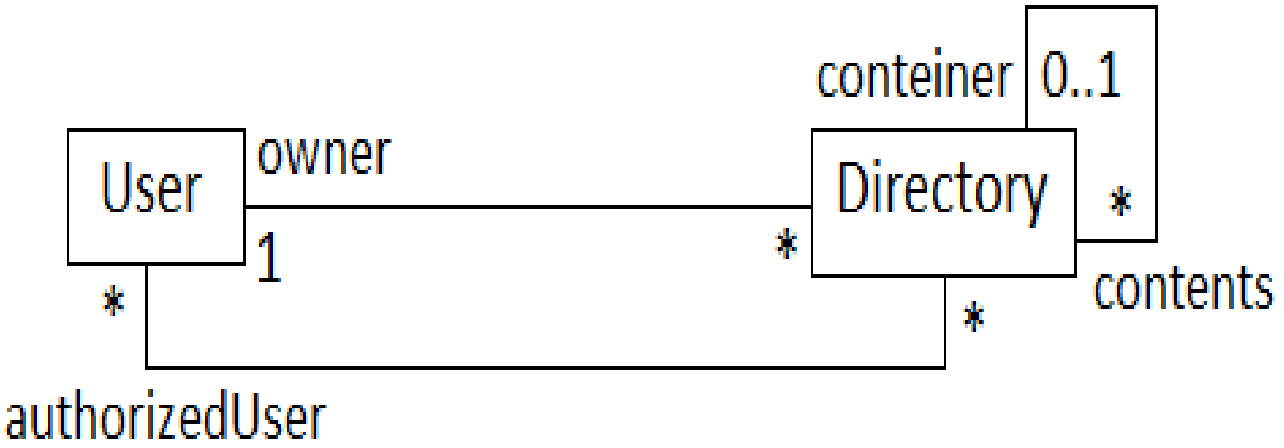


- Alla oliokaavio, joka kuvaa erään edellisen luokkakaavion mukaisen tilanteen
  - Käyttäjät mluukkai ja wikla
  - mluukkai omistaa kaksi hakemistoa
  - mluukkai:lla myös käyttöoikeus omistamiinsa hakemistoihin
    - Samojen olioiden välillä kaksi eri yhteyttä!
  - Wiklalla käyttöoikeus hakemistoon /home/mluukkai/ohmas09

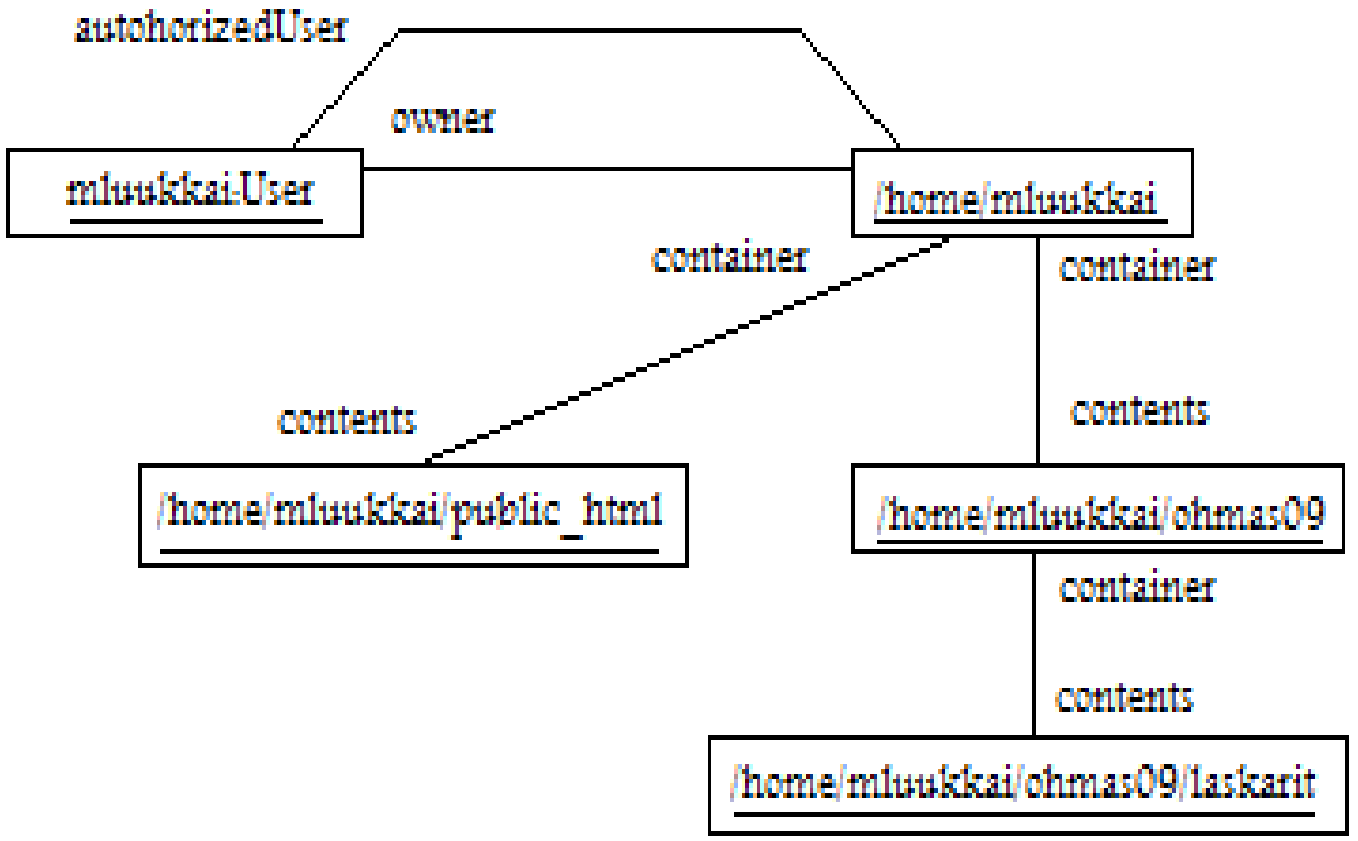


Yhteys kahden saman luokan olion välillä

- Miten mallinnetaan se, että hakemistolla on alihakemistoja?
  - Yhden olion suhteen yhteyksiä on oikeastaan kahdenlaisia
    - Hakemisto sisältää alihakemistoja
    - Hakemisto sisältyy johonkin toiseen hakemistoon
- Yhteen hakemisto-olioon voi liittyä 0 tai 1 hakemisto-olioa roolissa *container* (=sisältäjä), eli hakemisto voi olla jonkun toisen hakemiston alla
- Yhteen hakemisto-olioon voi liittyä mielivaltainen määrä (*) hakemisto-olioita roolissa *contents* (=sisältö), eli hakemisto voi sisältää muita hakemistoja

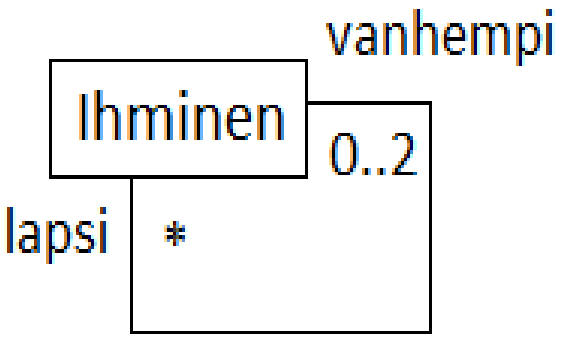


- Tilanne vaikuttaa sekavalta, selvennetään oliokaavion avulla
- /home/mluukkai sisältää kaksi hakemistoa
  - Alihakemistojen rooli yhteydessä on contents eli sisältö
  - Päähakemiston rooli yhteydessä on container eli sisältäjä
- /home/mluukkai/ohmas09 on edellisen alihakemisto, mutta sisältää itse alihakemiston



Yhteys kahden saman luokan olion välillä, toinen esimerkki

- Ihmisten välillä on suhteita, esim. vanhempi ja lapsi
- Ihmisellä voi olla 0-2 vanhempaa
  - Tietyn ihmisolionsuhteen nämä ovat roolissa vanhempi
- Ihmisellä voi olla lapsia
  - Tietyn ihmisolion suhteen nämä roolissa lapsi
- HUOM: malli mahdollistaa sen, että ihminen on itsensä lapsi!
  - Tarvittaisiin esim. kommentti joka kieltää tilanteen
- Luennolla piirretään asiaa selvittävä oliokaavio

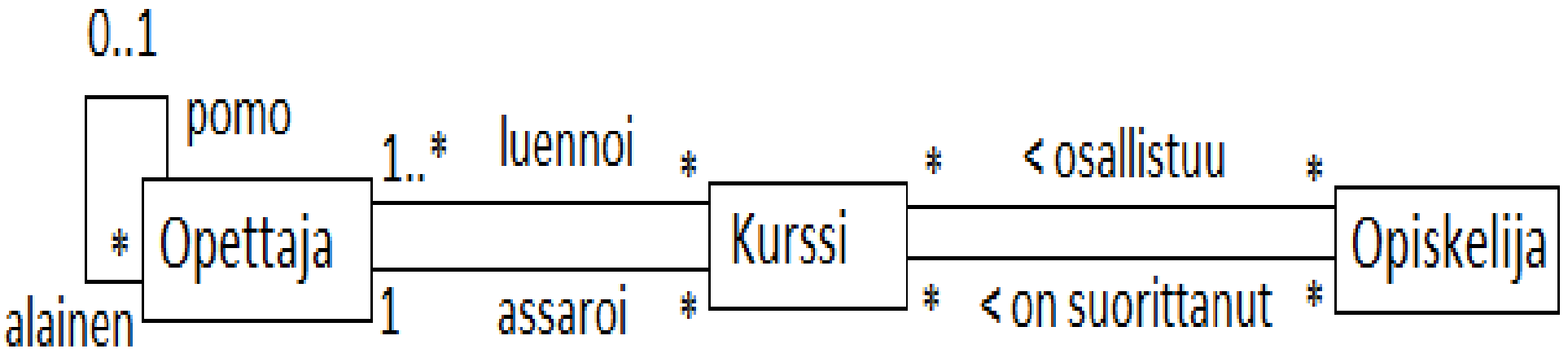


Monimutkaisempi esimerkki

- Mallinnetaan seuraava tilanne
  - Kurssilla on luennoijana 1 opettaja ja assarina useita opettajia
  - Opettaja voi olla useiden kurssien assarina ja luennoijana
  - Opettajalla voi olla pomo ja useita alaisia
    - Opettajat johtavat toisiaan
  - Opiskelija voi osallistua useille kursseille
  - Opiskelijalla voi olla suorituksia useista kursseista

- Seuraavalla sivulla luokkakaavio
  - Luennolla piirretään ehkä oliokaavio
- Assosiaation nimiin *osallistuu* ja *on suorittanut* on merkitty lukusuunta sillä ne luetaan oikealta vasemmalle:
  - Opiskelija *osallistuu* kurssille

- Monimutkaisia ongelmia kannattaa lähestyä paloissa
  - Mietitään ensin esim. kurssin ja opettajan suhdetta
  - Sitten opiskelijan ja kurssin suhdetta
  - Lopulta opettajien keskinäisiä pomotussuhteita
  - Kaikista eri vaiheista voidaan piirtää oma kaavionsa joka lopuksi yhdistetään
  - Tai samaa kaaviota voidaan laajentaa pikkuhiljaa



Luennolla tehtävä esimerkki

- Mallinnetaan yliopisto luokkakaaviona:
  - Yliopistossa on useita tiedekuntia
  - Tiedekunnissa on useita laitoksia
  - Tiedekunta kuuluu vain yhteen yliopistoon ja laitos vain yhteen tiedekuntaan
  - Jokainen henkilökunnan jäsen on töissä tietyllä laitoksella
  - Jokaisella laitoksella on yksi henkilökunnan jäsen esimiehenä
  - Yliopisto omistaa useita rakennuksia
  - Rakennuksessa voi sijaita yksi tai useampi laitos, kaikissa rakennuksissa tosin ei ole mitään laitosta
  - Laitos sijaitsee yhdessä tai joskus myös useammassa rakennuksessa

Yhteys vai ei?

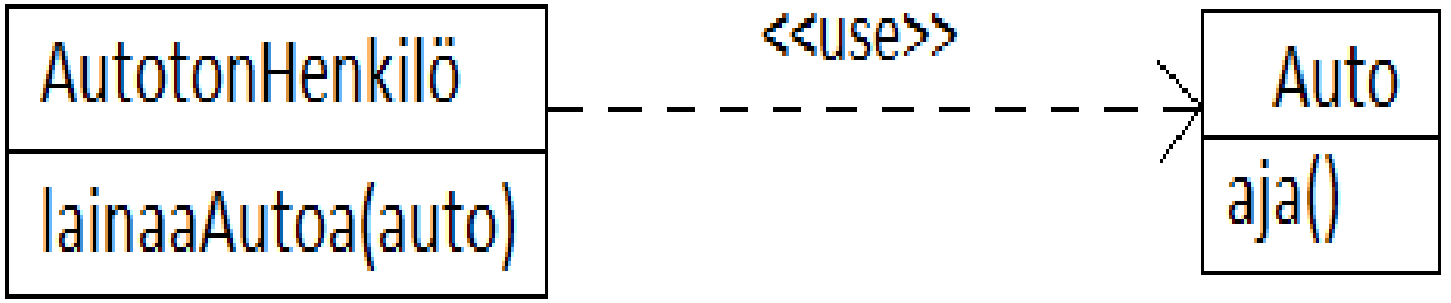
- Kuvitellaan, että on olemassa henkilöitä, jotka eivät omista autoa
- Autottomat henkilöt kuitenkin välillä lainaavat jonkun muun autoa
- Koodissa asia voitaisiin ilmaista seuraavasti:

```
class AutotonHenkilo {
    void lainaaJaAja( Auto lainaAuto ) {
        lainaAuto.aja();
    }
}
```

- Eli autottoman henkilön metodi lainaaJaAja saa parametrikseen auton (lainaAuto), jolla henkilö ajaa
- Auto on lainassa ainoastaan metodin suoritusajan
  - AutotonHenkilo ei siis omaa pysyvää suhdetta autoon

Ei yhteyttä vaan riippuvuus

- Kannattaako luokkien Auto ja AutotonHenkilo välille piirtää yhteys?
- Koska kyseessä ei ole pysyvämpiluontoinen yhteys, on parempi käyttää luokkakaaviossa *riippuvuussuhdetta* (engl. dependency)
- Riippuvuus merkitään katkoviivanuolena, joka osoittaa siihen luokkaan josta ollaan riippuvaisia
- Riippuvuus on tavallaan myös yhteys, mutta ”normaalia” yhteyttä ”heikompi” (= ei kestä yhtä kauaa)
- Alla on ilmaistu vielä riippuvuuden laatu
  - Tarkennin (eli stereotyyppi) <<use>> kertoo että kyseessä on käyttöriippuvuus, eli AutotonHenkilö kutsuu Auto:n metodia



Lisää riippuvuudesta

- Joskus riippuvuus määritellään siten, että luokka A on riippuvainen luokasta B jos muutos B:hen saa aikaan mahdollisesti muutostarpeen A:ssa
  - Näin on edellisessä esimerkissä: jos Auto-luokka muuttuu (esim. metodi aja muuttuu siten että se tarvitsee parametrin), joudutaan AutotonHenkilö-luokkaa muuttamaan

- Riippuvuus on siis jotain ”heikompaa” kun tavallinen luokkien välinen yhteys
  - Jos luokkien välillä on yhteys, on niiden välillä myös riippuvuus, sitä ei vaan ole tapana merkitä
  - Henkilö joka omistaa Auton on riippuvainen autosta...

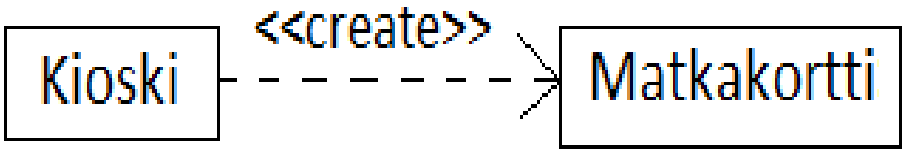
- OhPe:n viikon 1 matkakorttitehtävässä Kioski:lla on riippuvuus Matkakortti-luokkaan
  - Tämä on hiukan yllättävää, sillä Kioskihan luo matkakortin
  - Ylläolevan määritelmän mukaan kyseessä on riippuvuus, sillä jos Matkakortti-luokkaa muutetaan, luodaan matkakortin eri tavalla ja Kioskia on muutettava

Kioskilla on riippuivuus luokkaan Matkakortti

- Koodia katsomalla tämä on melko ilmeistä:

```
class Kioski {
    Matkakortti ostaMatkakortti( String omistaja, double arvo){
        Matkakortti uusi = new Matkakortti( omistaja );
        uusi.kasvataArvoa(arvo);
        return uusi;
    }
}
```

- Matkakortin koodissa ei Kioskia mainita mitenkään, joten riippuvuutta toiseen suuntaan ei ole
- Kioskin riippuvuus voidaan varustaa tarkenteella <<create>>



- Entä muiden matkakorttitehtävän luokkien suhde? Onko kyse yhteyksistä vai riippuvuuksista?

Esimerkki, **Auto sisältää neljä rengasta**

```
class Rengas{
    void pyöri(){  System.out.println("pyörii");  }
}

class Auto{  // neljä rengasta vasenEtu, oikeaEtu, ...
   Private  Rengas vEtu;  Rengas oEtu; Rengas vTaka; Rengas oTaka;


 Auto(){     // auto saa renkaansa syntyessään
    vEtu = new Rengas();     oEtu = new Rengas();
    vTaka = new Rengas();   oTaka = new Rengas();
 }

 void aja(){  // ajaessa kaikki renkaat pyörivät
    vEtu.pyöri();    oEtu.pyöri():
    vTaka.pyöri();  oTaka.pyöri():
 }
```

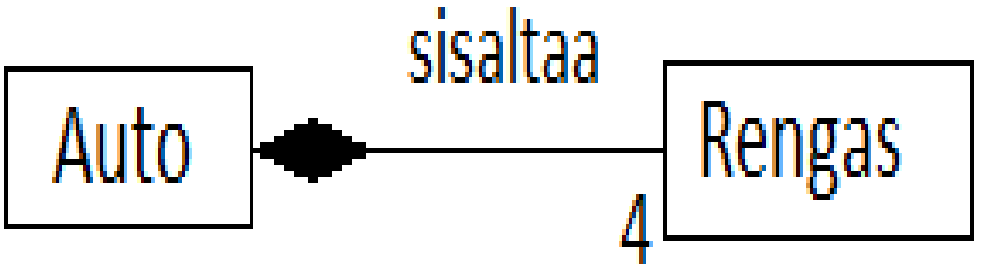
Kompositio

- Tilannehan voitaisiin mallintaa tekemällä autosta yhteys renkaisiin ja laittamalla kytkentärajoitteeksi 4
- Renkaat ovat kuitenkin siinä mielessä erityisessä asemassa, että voidaan ajatella, että ne ovat auton komponentteja
  - Renkaat sisältyvät autoon
- Kun auto luodaan, luodaan renkaat samalla
  - Koodissa auto luo renkaat
- Renkaat ovat private, eli niihin ei pääse ulkopuolelta käsiksi
- Kun roskienkerääjä tuhoaa auton, tuhoutuvat myös renkaat
- Eli ohjelman renkaat sisältyvät autoon ja niiden elinikä on sidottu auton elinikään (oikeat renkaat eivät tietenkään käyttäydy näin vaan ovat vaihdettavissa)

- Tämänkaltaista tilannetta, jota nimitetään **kompositioksi** (engl. composition), varten on oma symbolinsa, ks. seuraava sivu

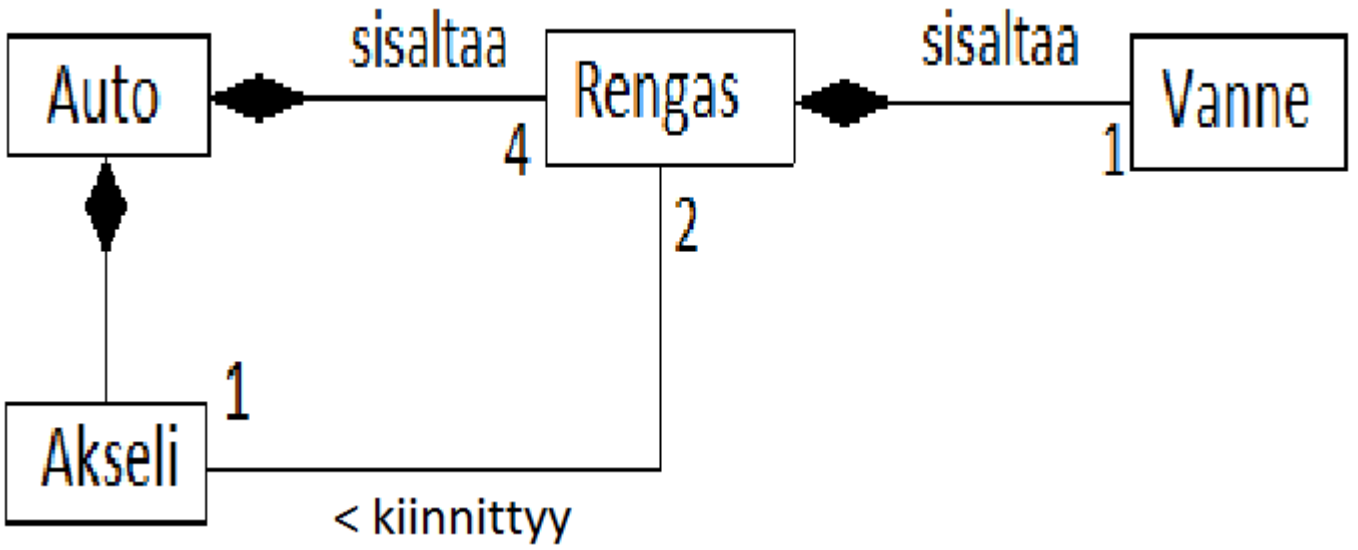
Kompositio

- Komposition symboli on ”musta salmiakkikuvio”, joka liitetään yhteyden siihen päähän, johon osat sisältyvät
- Kompositiota käytetään kun seuraavat ehdot toteutuvat:
  - *Osat ovat olemassaoloriippuvaisia kokonaisuudesta*
    - Auton tuhoutuessa renkaat tuhoutuvat
  - *Osa voi kuulua vaan yhteen samantyyppiseen kompositioon*
    - Rengasta ei voi siirtää toiseen autoon
  - *Osa on koko elinaikansa kytketty samaan kompositioon*
- Koska Rengas-olio voi liittyä nyt vain yhteen Auto-olioon, ei salmiakin puoleiseen päähän tarvita osallistumisrajoitetta koska se on joka tapauksessa 1



Monimutkaisempi esimerkki

- Tarkennettu Auto sisältää 4 rengasta ja 2 akselia
- Komposition osa voi myös sisältää oliota
  - Rengas sisältää vanteen
- Komposition osilla voi olla ”normaaleja” yhteyksiä
  - Akseli kiinnittyy kahteen renkaaseen
  - Rengas on kiinnittynyt yhteen akseliin



Kompositio, huomioita

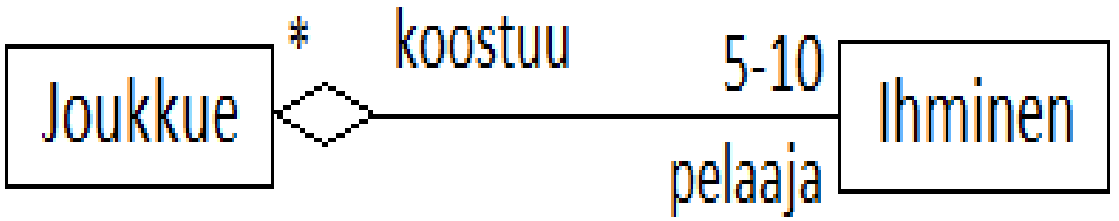
- Onko kompositiomerkkiä pakko käyttää?
  - Ei, mutta usein sen käyttö selkiyttää tilannetta

- Yksi edellisellä sivulla olleista komposition ehdoista on *Osa voi kuulua vaan yhteen samantyyppiseen kompositioon*
  - Eli rengas voi kuulua vain yhteen autoon
  - Mutta rengas voisi UML:n mielessä kuulua samaan aikaan johonkin muuhun kompositioon!
  - Renkaan kohdalla tilanne ei ole mielekäs
  - Esimerkki tilanteesta ensi tiistain luennolla

- Kompositio on siis erittäin rajoittava suhde olioiden välillä, toisin kuin ”normaali” yhteys
  - Välimuotona seuraavalla sivulla esiteltävä *kooste*

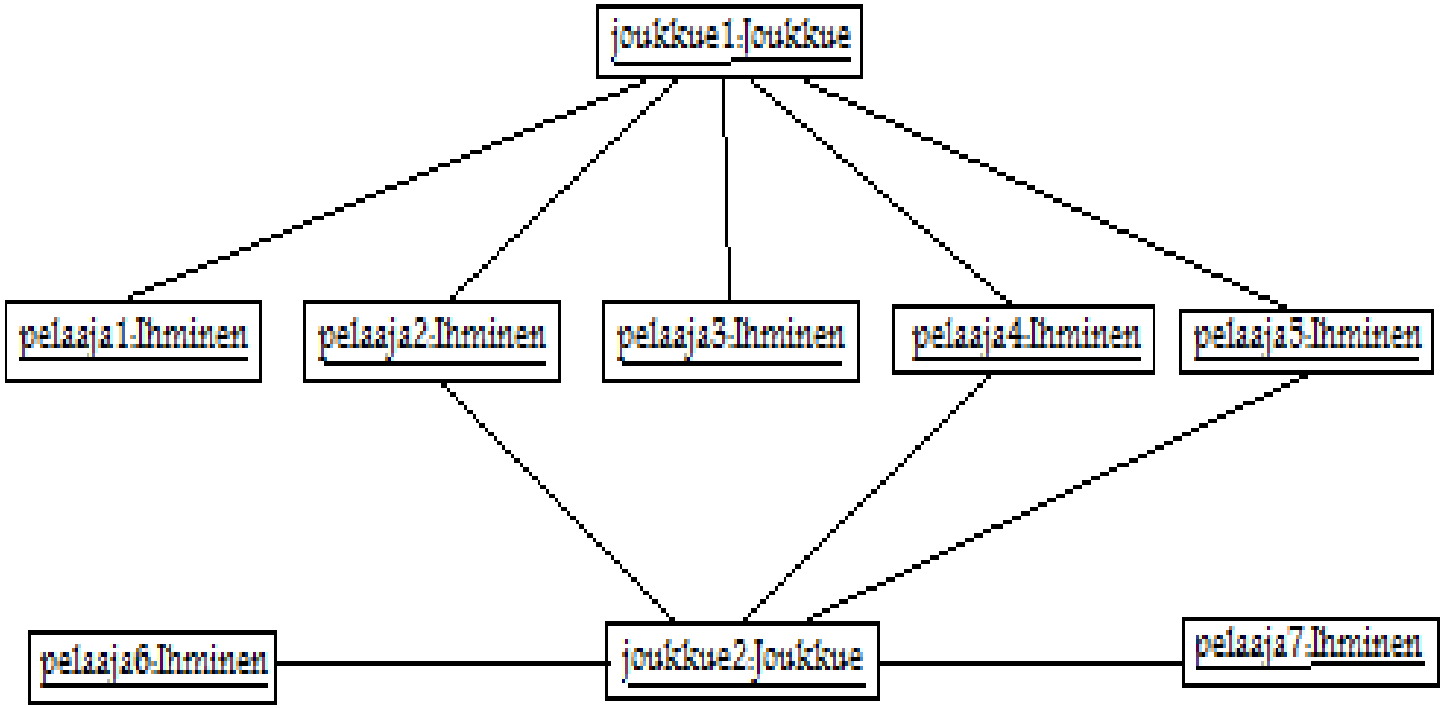
Kooste

- Koosteella (engl. agregation) tarkoitetaan koostumussuhdetta, joka ei ole yhtä komposition tapaan ”ikuinen”
  - koostesuhteisiin osallistumista ei ole rajoitettu yhteen
    - HUOM: suomenkieliset termit kooste ja kompositio ovat huonot ja jopa harhaanjohtavat
- Koostetta merkitään ”valkoisella salmiakilla” joka tulee siihen päähän yhteyttä, johon osat kuuluvat
- Esimerkki: Joukkue koostuu pelaajista (jotka ovat ihmisiä)
  - Ihminen ei kuitenkaan kuulu joukkueeseen ikuisesti
  - Joukkue ei syynnytä eikä tapa pelaajaa
  - Ihminen voi kuulua yhtäaikaa useampaan joukkueeseen
- Alla on rajattu, että joukkueella on 5-10 pelaajaa



Koosteen oliokaavio

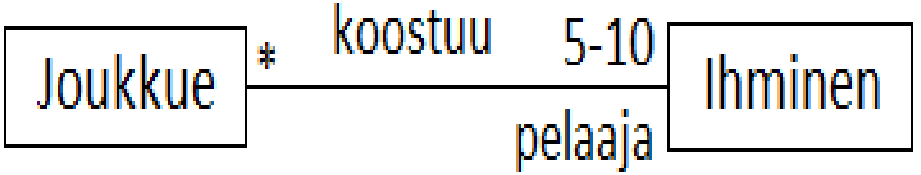
- Joukkueen 1 muodostavat pelaajat 1, 2, 3, 4 ja 5
- Pelaajat 2, 4, ja 5 kuuluvat myös joukkueeseen 2, jonka muut jäsenet ovat pelaajat 6 ja 7



Huomio koostesymboolin käytöstä

- Komposition (eli mustan salmiakin) merkitys on selkeä, kyseessä on olemassaoloriippuvuus
- On sensijaan epäselvää millon koostetta (eli valkoista salmiakkia) tulisi käyttää normaalin yhteyden sijaan
- Monet asiantuntevat oliomallintajat ovat sitä mieltä että koostetta ei edes tulisi käyttää
- Koostesuhde on poistunut UML:n standardista versiosta 2.0 lähtien
- Koostesuhde on kuitenkin edelleen erittäin paljon käytetty joten on hyvä tuntea symbooli passiivisesti

- Tällä kurssilla koostetta ei käytetä eikä sitä tarvitse osata
- Joukkueen ja pelaajien välinen suhde voidaankin ilmaista normaalina yhteytenä

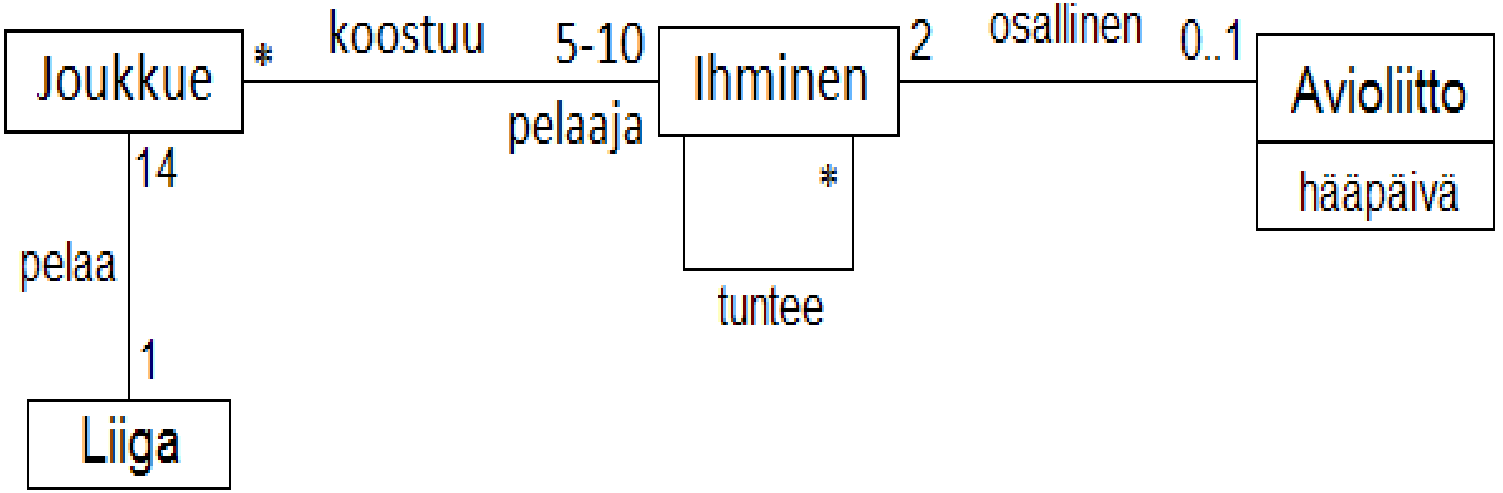


Monimutkaisempi esimerkki

- Joukkue pelaa liigassa jossa on 14 joukkuetta
- Ihminen voi kuulua mielivaltaisen moneen joukkueeseen
- Joukkueeseen kuuluu 5-10 ihmistä
- Ihminen tuntee useita ihmisiä
- Ihminen voi olla avioliitossa, mutta vain yhdessä avioliitossa kerrallaan
- Avioliitto koostuu kahdesta ihmisestä

- Vastaus seuraavalla sivulla
- Huomaa miten yhteys tuntee on mallinnettu
  - Toisessa päässä osallistumisrajoite 1 ja toisessa *
  - Voisi olla myös * ja *
  - Tuntee on symmetrinen yhteys toisin kuin jossain aiemmassa esimerkissä ollut johtaa, joten toisen pään osallistumisrajoituksella ei ole merkitystä

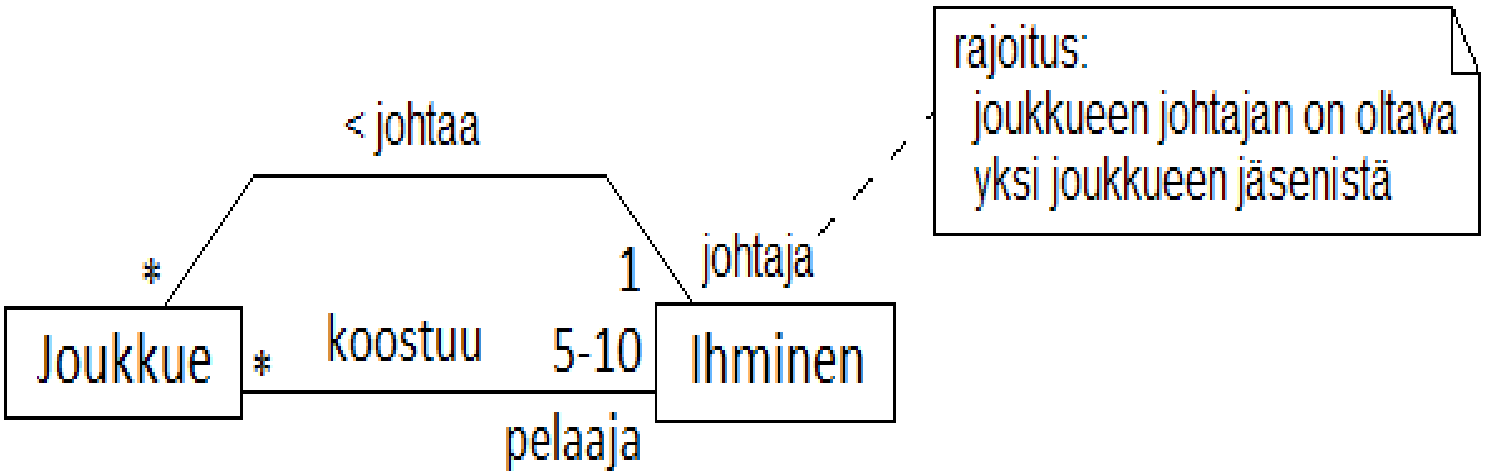
- Edellisen sivun tilannetta vastaava UML-kaavio:



- Entä jos tätä vielä laajennettaisiin seuraavasti:
  - Liigalla on sarjaohjelma
  - Sarjaohjelma sisältää 52 pelikierrosta
  - Kullakin pelikierroksella pelataan 7 ottelua
  - Ottelussa pelaa vastakkain 2 joukkuetta, joista toinen on kotijoukkue ja toinen vierasjoukkue

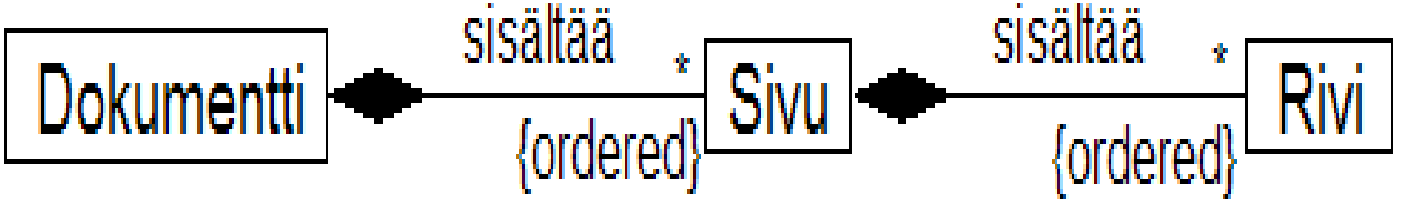
Rajoitukset

- Jos haluttaisiin mallintaa tilanne, että joku joukkueen jäsenistä on joukkueen johtaja, pelkkä luokkakaavio (siinä määrin kun tällä kurssilla UML:ää opitaan) ei riitä
- Tilanne voitaisiin mallintaa seuraavalla sivulla esitetyllä tavalla
  - Eli lisätään normaali yhteys *johtaa* joukkueen ja ihmisen välille
  - Määritellään kytkentärajoite: joukkueella on tasan 1 johtaja
  - Ilmaistaan UML-kommenttina, että joukkueen johtajan on oltava joku joukkueen jäsenistä



Yhteydessä olevien luokkien järjestys

- Esim. edellisessä esimerkissä joukkue koostuu pelaajista
  - Pelaajilla ei ole kuitenkaan mitään erityistä järjestystä yhteyden kannalta
- Joskus osien järjestys voi olla tärkeä
- Esim. dokumentti sisältää sivuja ja kukin sivu sisältää tekstirivejä
  - Sivujen ja rivien on oltava tietyssä järjestyksessä, muuten dokumentissa ei ole järkeä
- Se seikka, että osat ovat järjestyksessä voidaan ilmaista lisämääreenä {ordered}, joka laitetaan niiden olioiden päähän yhteyttä joilla on järjestys
  - Jos ei ole selvää minkä suhteen oliot on järjestetty, voidaan asia ilmaista kommentilla



Millä kaaviot kannattaa piirtää?

- Ensinnäkään, kaikkea ei tarvitse eikä kannatakaan tunkea samaan kuvaan
  - Esim. LAL-järjestelmän käyttötapauskaavion voi aivan hyvin esittää monena erillisenä kaaviona
  - Nyrkkisääntönä voisi olla, että yhdessä kaaviossa kannattaa olla max 10 asiaa (= luokkaa, käyttötapausta, ...)
- Eli mitä työkaluja kannattaa käyttää?
- Kynä ja paperia tai valkotaulu
  - Paperi talteen tai roskiin
  - Tarvittaessa skannaus tai digikuva
- Tarjolla paljon ilmaisia työkaluja
  - Paint (suuri osa näidenkin kalvojen kuvista tehty paintilla)
  - Dia (win+linux)
  - Umbrello (ei osaa ns. sekvenssikaavioita)
  - ArgoUML (osaa vaan standardin vanhaa versiota, käy tälle kurssille)
  - Openoffice

- Paljon maksullisia vaihtoehtoja, mm.
  - Visual Paradigm
  - Magic Draw
  - Rational Rose
  - Microsoft Visio
  - Omnigraffle (Mac)
- Magic Draw:n Community edition saatavissa ilmaiseksi, ks.
  - ks. https://www.cs.helsinki.fi/intranet/group/cinco/teaching/md/
  - Saatetaan käyttää kevään kurssilla Tietokantojen perusteet
- Visio löytyy laitoksen ohjelmistojakelusta:
  - ks. www.cs.helsinki.fi/compfac/ohjeet/msdnaa

- Monet maksulliset tuotteet osaavat generoida luokkamäärittelyistä koodirunkoja ja myös toisinpäin, koodista UML:ää

- Ei ole olemassa selkeää vastausta mitä työkalua kannattaa käyttää. Tämän kurssin tarpeisiin kynä ja paperia riittää hyvin