Ohjelmistojen mallintaminen

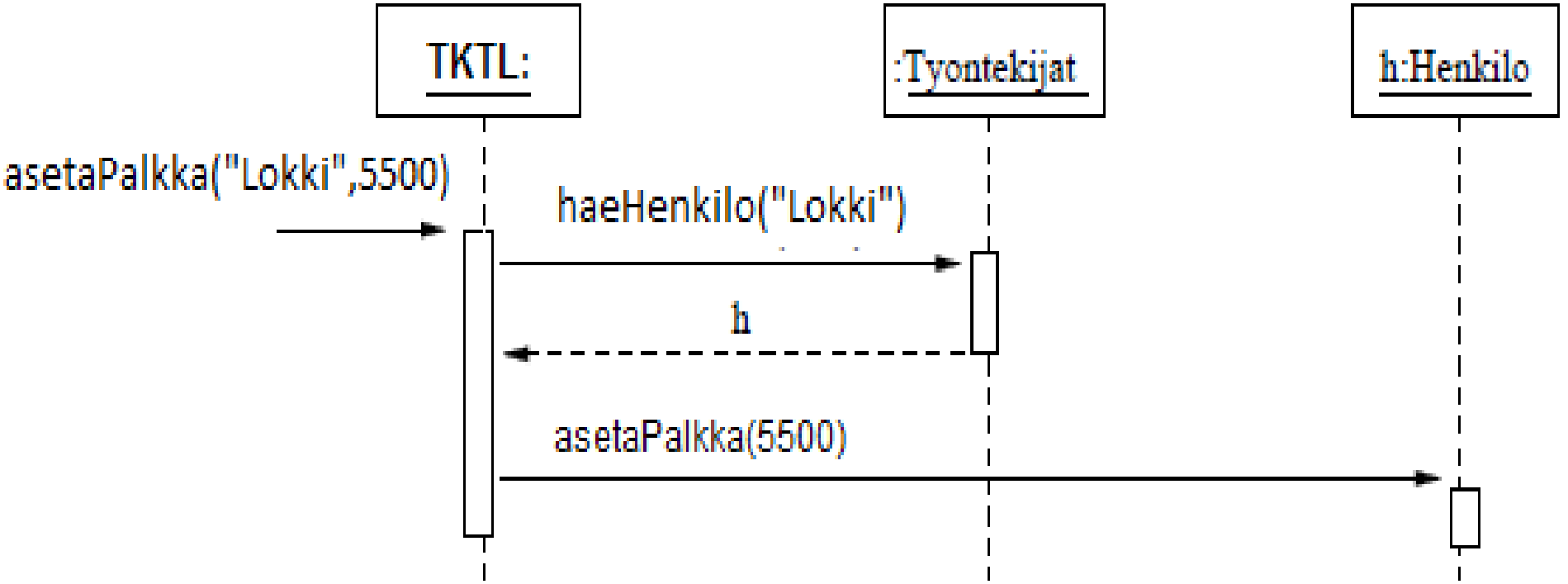
Luento 4, 20.11.

Kertaus: olioiden yhteistyön kuvaaminen

- Luokkakaavion avulla voidaan kuvata ohjelman rakenne
  - Minkälaisista luokista ohjelma koostuu ja miten luokat liittyvät toisiinsa
- Ohjelman toiminnallisuudesta vastaavat luokkien instanssit eli oliot. Oleellista olioiden yhteistyö
  - Miten oliot kutsuvat toistensa metodeja suorituksen aikana
  - Eli minkälaista yhteistyötä oliot tekevät
- Olioiden yhteistyön kuvaamiseen omat kaaviotyyppinsä:
  - Suositumpi **sekvenssikaavio**
  - ja vähemmän käytetty *kommunikaatiokaavio*
- Oliosuunnittelussa näillä kaavioilla erityisen tärkeä asema

- Seuraavilla kalvoilla esimerkki viimeviikolta

- Esimerkkisovellus:
  - Työntekijät-olio pitää kirjaa työntekijöistä, jotka Henkilö-oliota
  - Sovelluslogiikka-olio hoitaa korkeamman tason komentojen käsittelyn
- Tarkastellaan operaatiota lisaaPalkka(nimi, palkka)
  - Lisätään parametrina annetulle henkilölle uusi palkka
- Suunnitellaan, että operaatio toimii seuraavasti:
  - Ensin sovelluslogiikka hakee Tyontekijat-oliolta viitteen Henkilo-olion
  - Sitten sovelluslogiikka kutsuu Henkilo-olion palkanasetusmetodia
- Seuraavalla sivulla operaation suoritusta vastaava sekvenssikaavio
  - Havainnollistuksena myös osa luokan Sovelluslogiikka koodista




```
public class Sovelluslogiikka{
    Tyontekijat tyontek;     // attribuutti, jonka kautta sovelluslogiikka tuntee työntekijät
    public void lisaaPalkka(String nimi, int palkka ){
        Henkilo h = tyontek.haeHenkilo( nimi );
        h.asetaPalkka( palkka );
    }
}
```

Muutamia huomioita (ks. myös viikon 3 tehtävien mallivastaukset)

- Olioiden nimeäminen
  - nimi:Luokka, kumpikin osista voi jäädä pois
  - Olion nimiosaan voidaan viitata muualla kaaviossa, esim. paluarvona tai metodin parametrina
- Mistä mihin metodikutsua kuvaava nuoli piirretään?
  - Kutsuvasta oliosta siihen olioon kenen metodia kutsutaan
  - Esim sovelluslogiikan koodissa metodikutsu

    Henkilo h = tyontek.haeHenkilo(nimi);

    kuvataan nuolena sovelluslogiikkaoliosta Tyontekijat-olioon
- Metodkutsujen parametrit merkataan tarvittaessa sulkuihin
- Paluuarvo merkataan tarvittaessa katkoviivalla tai metodikutsun yhteyteen
- Viittaus olioiden nimiin:
  - Esimerkissä h viittaa Henkilo-olioon, huomaa, miten h:ta käytetään metodin paluuarvona
- Vain alusta asti olemassaolevat oliot merkataan ylälaitaan, muut syntyhetken kohdalle

Yhteenveto olioiden yhteistoiminnan kuvaamisesta

- Sekvenssikaaviot ja luokkakaaviot eivät ole toistensa kilpailijoita, molemmat kuvaavat järjestelmää omasta näkökulmastaan

- Sekvenssikaaviot ja edellisellä luennolla nopeasti käsitellyt kommunikaatiokaaviot ovat erittäin tärkeä asema oliosuunnittelussa
  - Sekvenssikaaviot ovat huomattavasti yleisemmässä käytössä
- Kaaviot kannattaa pitää melko pieninä ja niitä ei kannata tehdä kuin järjestelmän tärkeimpien toiminnallisuuksien osalta
  - Kommunikaatiokaaviot ovat yleensä hieman pienempiä, mutta toisaalta metodikutsujen ajallinen järjestys ei käy niistä yhtä hyvin ilmi kuin sekvenssikaavioista
- On epäselvää missä määrin luennolla 3 mainittuja sekvenssikaavioiden valinnaisuutta ja toistoa kannattaa käyttää

- Sekvenssikaaviot on alunperin kehitetty tietoliikenneprotokollien kuvaamista varten

Määrittelyvaiheen luokkakaavion laatiminen

Kertaus ja esimerkki

Kertaus:
käsiteanalyysi eli menetelmä luokkamallin muodostamiseen

1. Etsi luokkaehdokkaat tekstikuvauksista (substantiivit)
2. Karsi luokkaehdokkaita (mm. poista tekemistä tarkoittavat sanat)
3. Tunnista olioiden väliset yhteyksiä (verbit ja genetiivit vihjeenä)
4. Lisää luokille attribuutteja
5. Tarkenna yhteyksiä (kytkentärajoitteet, kompositiot)
6. Etsi ”rivien välissä” olevia luokkia
7. Etsi yläkäsitteitä
8. Toista vaiheita 1-7 riittävän kauan

- Aloitetaan vaiheella 1, sen jälkeen edetään muihin vaiheisiin peräkkäin, rinnakkain tai/ja sekalaisessa järjestyksessä toistaen
- Lopputuloksena alustava toteutusriippumaton sovelluksen kohdealueen luokkamalli
  - Malli tulee tarkentumaan ja täsmentymään suunnitteluvaiheessa
  - Siksi ei edes kannata tähdätä ”täydelliseen” malliin

Esimerkki: elokuvateatteri

- Ratkaisu luennon 3 kalvoilla
- Tarkasteltavana ilmiönä on elokuvalipun varaaminen. Lippu oikeuttaa paikkaan tietyssä näytöksessä. Näytöksellä tarkoitetaan elokuvan esittämistä tietyssä teatterissa tiettyyn aikaan. Samaa elokuvaa voidaan esittää useissa teattereissa useina aikoina. Asiakas voi samassa varauksessa varata useita lippuja yhteen näytökseen. Asiakas tekee lippuvarauksen elokuvateatterin internetpalvelun kautta. Elokuvateatterissa on useita lippukassoja. Asiakas lunastaa varauksensa lippukassalta viimeistään tuntia ennen esitystä.

Huomioita määrittelyvaiheen luokkakaaviosta

- Substantiiveja saa yleensä siivota aika rankalla kädellä, läheskään kaikki eivät ole ”hyviä” luokkia
- Määrittelyvaiheen luokkakaaviota tehdessä ei kannata ajatella liikaa toiminnallisuutta eli sitä kuka tekee ja mitä
  - Toiminnallisuutta ajatellaan tarkemmin vasta suunnitteluvaiheessa
- Olioiden välisissä suhteissa huomio kannattaa kiinnittää pysyvämpiluonteisiin yhteyksiin
  - Esim. edellisessä tekstissä ilmenee että *asiakas tekee varauksen*, luokkakaavion kannalta tässä ei ole oleellista kuka varauksen tekee vaan se että **varauksiin liittyy asiakas**
  - Tekeminen siis välillä implikoi että olioilla on pysyvämpiluonteisiin yhteys
  - (Asiakas-oliot eivät todennäköisesti ohjelmakoodissa tule olemaan niitä joka tekevät varauksen, asiakas on lähinnä asiakkaan tietoja tallettava entiteetti. Toiminnasta, esim. varauksen tekemisestä vastaa todennäköisesti joku muu olio)
- Olioiden välillinen hetkellinen yhteys merkitään joskus riippuvuutena
  - Riippuvuuksia merkitään kuitenkin aika harvoin määrittelyvaiheen luokkakaavioihin

Huomioita määrittelyvaiheen luokkakaaviosta

- Tekstuaalisista kuvauksista eivät aina kaikki tärkeät luokat tule esille

- Tosimaailmassa ei tietenkään aina ole mitään tekstuaalista kuvausta
  - Käsiteanalyysiä voi tehdä esim. käyttötapausten tekstuaalisille kuvauksille tai asiakkaan ”puheelle”
- Määrittelyvaiheen luokkakaavion tekemiseen ei kannata tuhlata hirveästi aikaa, on nimittäin (lähes) varmaa, että suunnittelu- ja toteutusvaiheessa luokkamallia muutetaan
  - Lopputulosta tärkeämpi joskus itse prosessi
- Suunnittelu- ja toteutusvaiheessa järjestelmään lisätään muutenkin luokkia hoitamaan esim. seuraavia tehtäviä
  - Käyttöliittymän toteutus
  - Tietokantayhteyksien hallinta
  - Sovelluksen ohjausoliot
- Määritteluvaiheen luokkakaavio toimii pohjana tietokantasuunnittelulle
  - Osa olioistahan (esim. olutkaupassa Tuote-oliot) on tallennettava

Mallinnuksen eteneminen

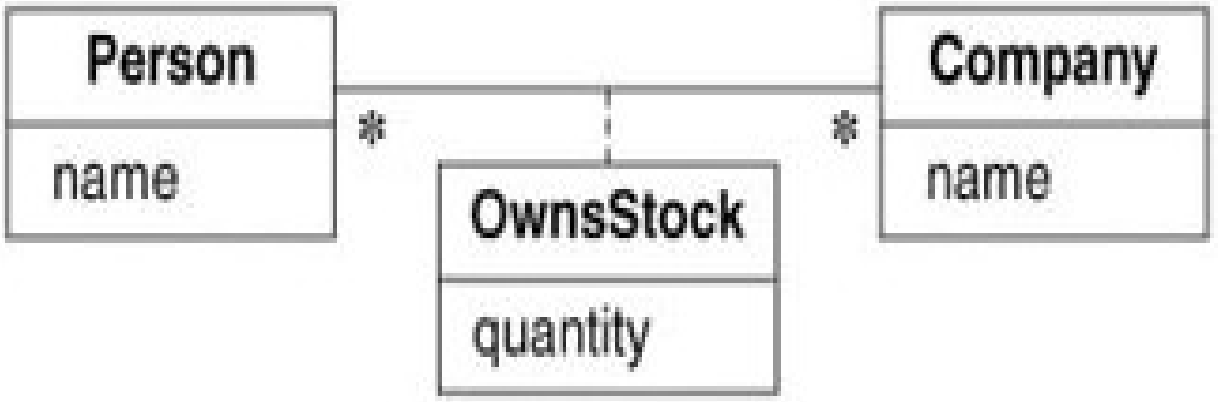
- Isoa ongelmaa kannattaa lähestyä pienin askelin, esim:
  - Yhteydet ensin karkealla tasolla, tai
  - Tehdään malli pala palalta, lisäten siihen muutama luokka yhteyksineen kerrallaan

- Mallinnus iteratiivisesti etenevässä ohjelmistokehityksessä
  - Ketterissä menetelmissä suositaan *iteratiivista* lähestymistapaa ohjelmistojen kehittämiseen
    - kerralla on määrittelyn, suunnittelun ja toteutuksen alla ainoastaan osa koko järjestelmän toiminnallisuudesta
  - Jos ohjelmiston kehittäminen tapahtuu ketterästi, kannattaa myös ohjelman luokkamallia rakentaa iteratiivisesti
  - Eli jos ensimmäisessä iteraatiossa toteutetaan ainoastaan muutaman käyttötapauksen kuvaama toiminnallisuus, esitetään iteraation luokkamallissa vain ne luokat, jotka ovat merkityksellisiä tarkastelun alla olevan toiminnallisuuden kannalta
  - Luokkamallia täydennetään myöhempien iteraatioiden aikana niiden mukana tuoman toiminnallisuuden osalta

Yhteyteen liittyvät tiedot

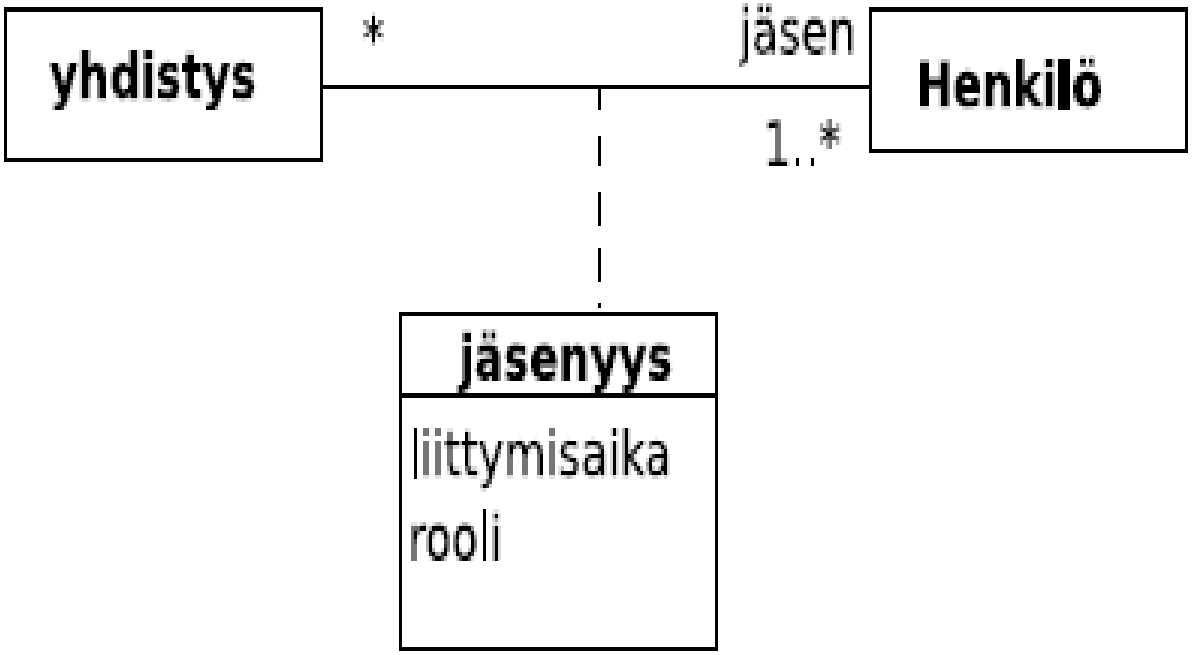
Yhteyden tietojen mallinnus

- Yhteyteen voi joskus liittyä myös tietoa
- Esim. tilanne missä henkilö voi olla (usean) yhtiön osakkeenomistaja
  - Osakkeenomistuksen kannalta tärkeä asia on omistettujen osakkeiden määrä
- Yksi tapa mallintaa tilanne on käyttää **yhteysluokkaa** (engl. Association class), eli yhteyteen liittyvää luokkaa, joka sisältää esim. yhteyteen liittyviä tietoja
  - Alla yhteysluokka sisältää omistettujen osakkeiden määrän



Toinen yhteysluokkaesimerkki

- Luentomonisteessa mallinnetaan tilanne, jossa henkilö voi olla jäsenenä useassa yhdistyksessä
  - yhdistyksessä on vähintään 1 jäsen
- Jäsenyys kuvataan yhteytenä, johon liittyy yhteysluokka
  - jäsenyyden alkaminen (liittymisaika) sekä jäsenyyden tyyppi (rooli, eli onko rivijäsen, puheenjohtaja tms...) kuvataan yhteysluokan avulla

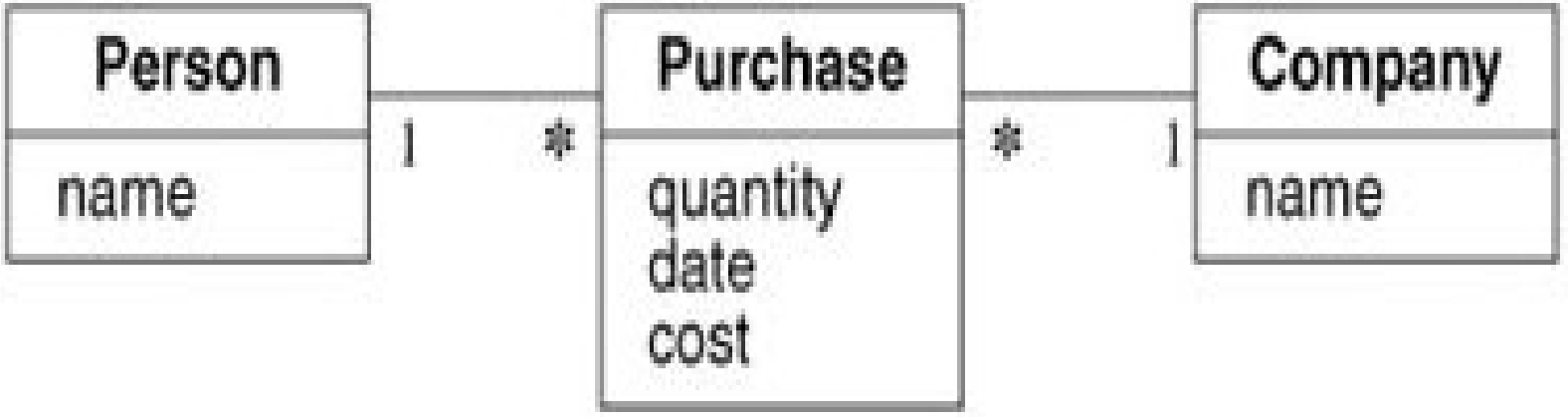


Kannattaako yhteysluokkia käyttää?

- Kannattaako yhteysluokkia käyttää?
  - Korkean tason abstrakteissa malleissa ehkä
  - Suunnittelutason malleissa todennäköisesti ei, sillä ei ole selvää, mitä yhteysluokka tarkoittaa toteutuksen tasolla
- Yhteysluokan voi aina muuttaa tavalliseksi luokaksi
- Yhteysluokka joudutaankin käytännössä aina ohjelmoidessa toteuttamaan omana luokkanaan, joka yhdistää alkuperäiset luokat joiden välillä yhteys on
  - Tämän takia yhteysluokkia ei välttämättä kannata käyttää alunperinkään

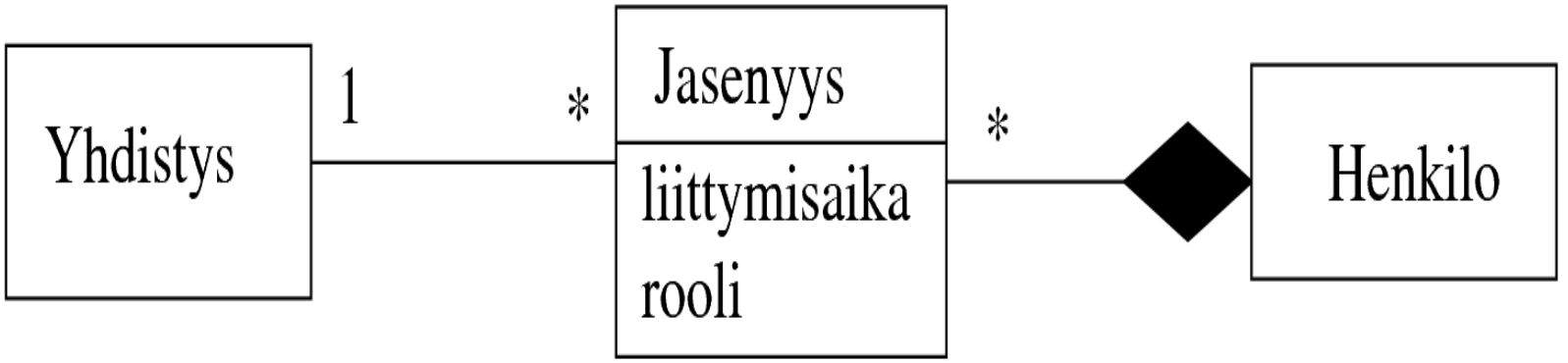
Yhteysluokasta normaaliksi luokaksi

- Alla muutaman sivun takainen osake-esimerkki ilman yhteysluokkaa
- Henkilöllä on useita ostoksia (purchase)
- Ostokseen liittyy määrä (kuinka monesta osakkeesta kyse), päiväys ja hinta
- Yksi ostos taas liittyy tasan yhteen yhtiöön ja tasan yhteen henkilöön
- Henkilö ei ole enää suorassa yhteydessä firmaan
  - Henkilö ”tuntee” kuitenkin omistamansa firmat ostos-olioiden kautta



Yhteysluokasta normaaliluokaksi

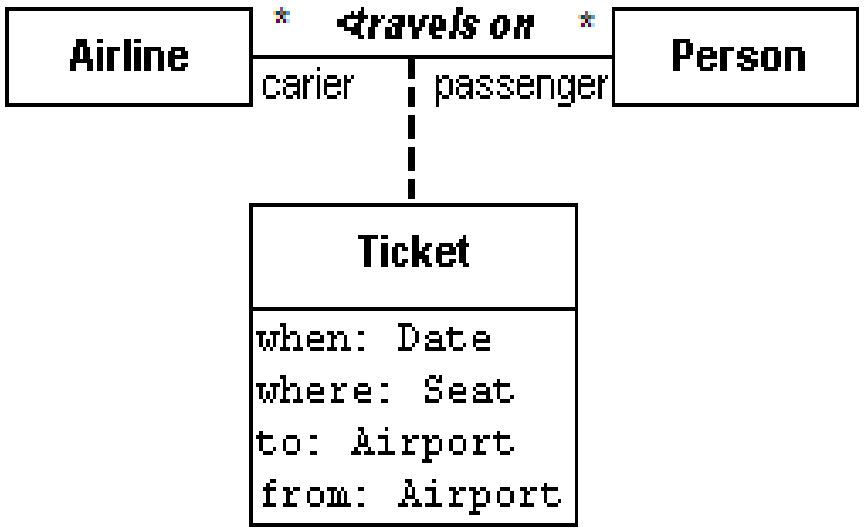
- Henkilön jäsenyys yhdistyksessä on myös luontevaa kuvata yhteysluokan sijaan omana luokkanaan



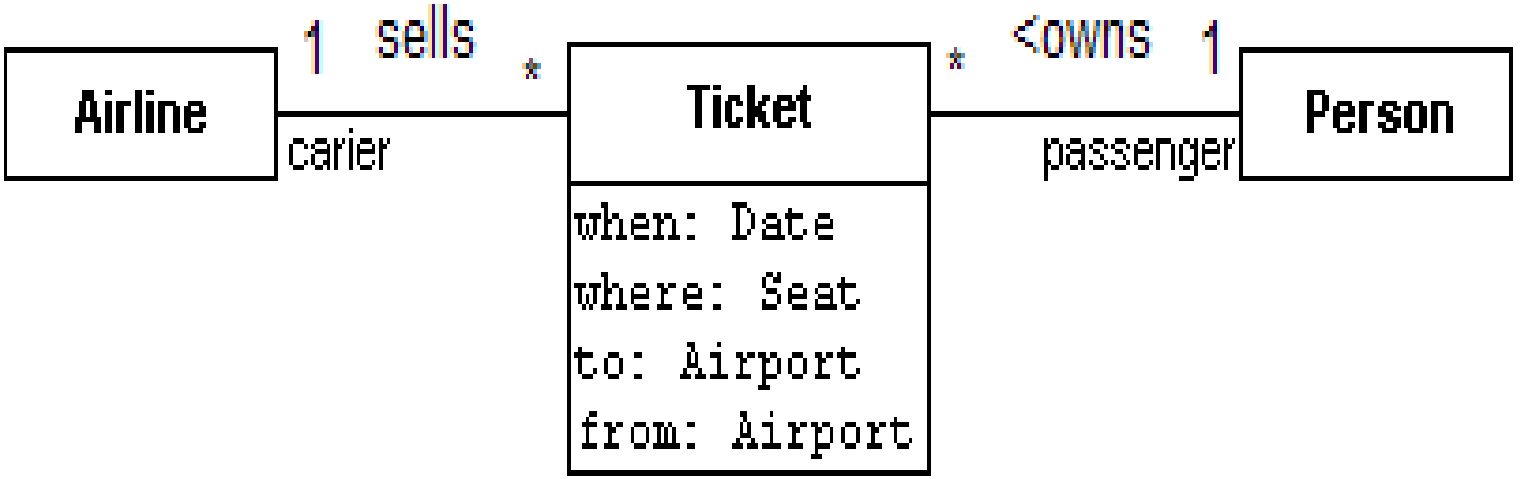
- Jäsenyyden olemassaoloriippuvuus on nyt merkitty henkilöön
- Miksi näin? Miksei yhdistykseen tai peräti molempiin?
- Periaatteessa loogisinta olisi tehdä jäsenyydestä olemassaoloriippuvainen sekä yhdistyksestä että henkilöstä, mutta UML ei salli tätä
- Olemassaoloriippuvuus saa siis olla vain yhteen olioon ja hetken mietinnän jälkeen on päätetty valita Henkilö jäsenyyden ”omistavaksi” osapuoleksi, periaatteessa Yhdistys olisi ollut yhtä hyvä valinta

Kaksi olioa ja yhteyksien lukumäärä

- Kurssin kotisivulle linkitetyssä Holubin UML quick referensessä on alla oleva esimerkki
- Eli henkilöllä voi olla *travels on* -yhteyksiä useiden lentoyhtiöiden kanssa
  - Yhteysluokkana Ticket on kerrottu matkan tiedot
- Tähän malliin sisältyy ongelma:
  - Henkilö voi olla *travels on* -yhteydessä moniin eri lentoyhtiöolioihin
  - Saman lentoyhtiöolion (esim. Finnair) kanssa ei kuitenkaan voi olla useampaa yhteyttä



- Siis: **jos luokkakaaviossa kahden luokan välillä on yhteys, voi kaksi luokkien olioa olla vain yhdessä yhteydessä kerrallaan**
  - Esim. olioiden arto:Person ja finnair:Airline välillä voi olla vain yksi yhteys, eli Arto voi lentää Finnairilla vain kerran
  - Tämä siitä huolimatta, että kytkentärajoitus on *
  - Artolla voi olla useita lippuja, mutta jokainen täytyy olla eri lentoyhtiöltä!
- Järkevin ratkaisu ongelmaan on kuvata yhteys omana luokkanaan
  - Henkilöllä voi olla useita lippuja
  - Lippu liittyy tiettyyn lentoyhtiöön ja tiettyyn henkilöön
  - Lentoyhtiön liittyy useita myytyjä lippuja



Yleistys-erikoistussuhde eli perintä

Yleistys-erikoistus ja periminen

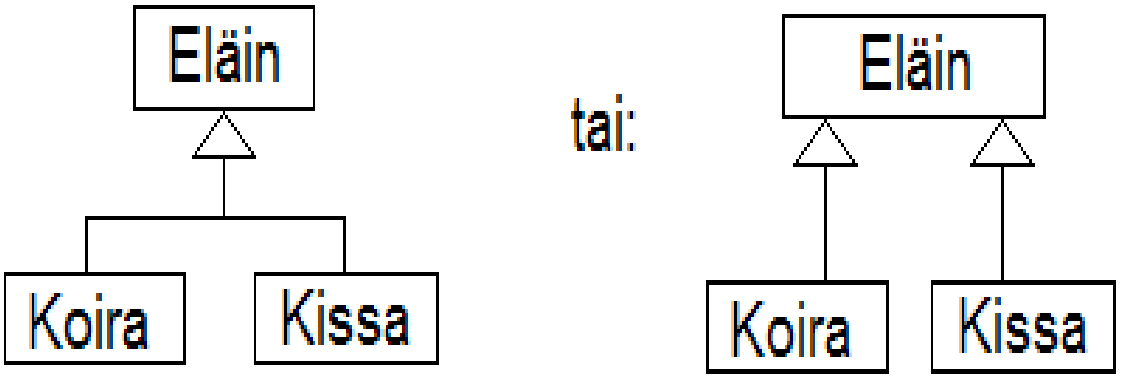
- Tähän mennessä tekemissämme luokkakaaviossa kaksi luokkaa ovat voineet liittyä toisiinsa muutamalla tapaa
- *Yhteys ja kompositio* liittyvät tilanteeseen, missä luokkien olioilla on rakenteellinen (= jollain lailla pysyvä) suhde, esim.:
  - Henkilö *omistaa* Auton (*yhteys*: normaali viiva)
  - Huoneet *sijaitsevat* Talossa (*kompositio*: musta salmiakki)
    - Musta salmiakki tarkoittaa olemassaoloriippuvuutta, eli salmiakin toisen pää olemassaolo riippuu salmiakkipäässä olevasta
    - Jos talo hajotetaan, myös huoneet häviävät, huoneita ei voi siirtää toiseen taloon
- Löyhempi suhde on taas *riippuvuus*, liittyy ohimenevämpiin suhteisiin, kuten tilapäiseen käyttösuhteeseen, esim:
  - AutotonHenkilö *käyttää* Autoa (katkoviivanuoli)
- Tänään tutustumme vielä yhteen hieman erilaiseen luokkien väliseen suhteeseen, eli *yleistys-erikostussuhteeseen,* jonka vastine ohjelmoinnissa on *periminen*

Yleistys-erikoistus ja periminen

- Ajatellaan luokkia Eläin, Kissa ja Koira
- Kaikki Koira-luokan oliot ovat selvästi myös Eläin-luokan oliota, samoin kaikki Kissa-luokan oliot ovat Eläin-luokan olioita
- Koira-oliot ja Kissa-oliot ovat taas täysin eriäviä, eli mikään koira ei ole kissa ja päinvastoin
- Voidaankin sanoa, että luokkien Eläin ja Koira sekä Eläin ja Kissa välillä vallitsee yleistys-erikoistussuhde:
  - Eläin on **yliluokka** (superclass)
  - Kissa ja Koira ovat eläimen **aliluokkia** (engl. Subclass)
- Yliluokka Eläin siis määrittelee mitä tarkoittaa olla eläin
  - Kaikkien mahdollisten eläinten yhteiset ominaisuudet ja toiminnallisuudet
- Aliluokassa, esim. Koira tarkennettaan mitä muita ominaisuuksia ja toiminnallisuutta luokan olioilla eli Koirilla on kuin yliluokassa Eläin on määritelty
- Aliluokat siis *perivät* (engl. inherit) yliluokan ominaisuudet ja toiminnallisuuden

Yleistys-erikoistus ja periminen

- Luokkakaaviossa yleistyssuhde merkitään siten, että **aliluokasta piirretään yliluokkaan kohdistuva nuoli, jonka päässä on iso ”valkoinen” kolmio**
- Jos aliluokkia on useita, voivat ne jakaa saman nuolenpään tai molemmat omata oman nuolensa, kuten alla



- Tarkkamuistisimmat huomaavat ehkä, että olemme jo törmänneet kurssilla yleistys-erikoisstussuhteeseen *käyttötapausten* yhteydessä
  - Luennolta 1: Yleistetty käyttötapaus *opetustarjonnan ylläpito* erikoistui *kurssin perustamiseen*, *laskariryhmän perustamiseen* ym...
  - Sama valkoinen kolmiosymbooli oli käytössä myös käyttötapausten yleistyksen yhteydessä

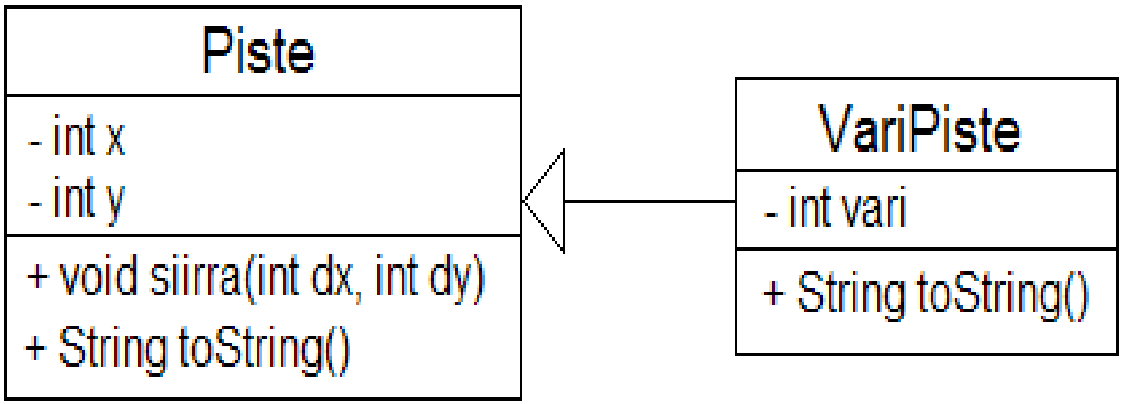
Periytyminen

- Luokkien välinen yleistys-erikoistussuhde eli yli- ja aliluokat toteutetaan ohjelmointikielissä siten, että aliluokka perii yliluokan
- Tuttu esimerkki Ohjelmoinnin jatkokurssilta, konstrutoreja ei merkitty:
  - Luokkakaavio seuraavalla sivulla

```
public class Piste{
    private int x, y;
    public void siirra(int dx, int dy) {
        x+=dx; y+=dy;
    }
    public String toSting(){ return ”(”+x+”)”; }
}
public class VariPiste extends Piste {
    private String vari;
    public String toSting(){ return super.toString()+” väri: ”+vari; }
}
```

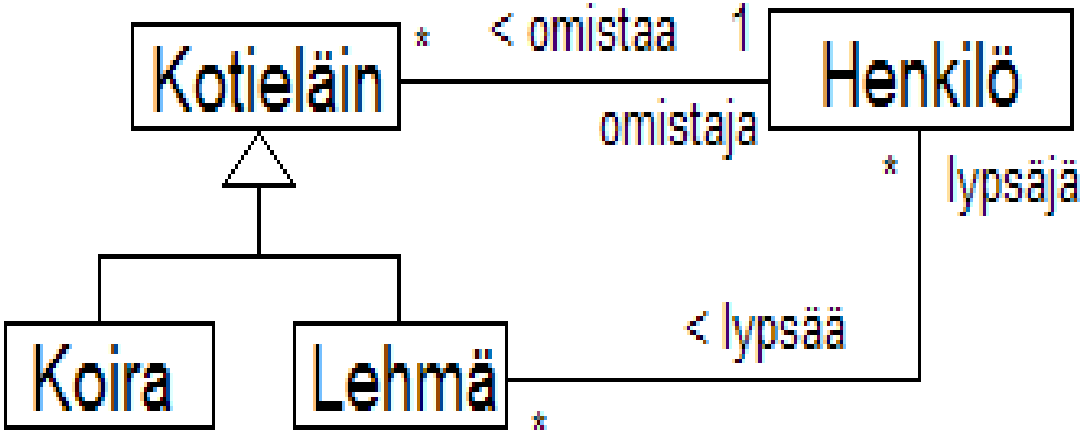
Periytyminen ja luokkakaavio

- Yliluokan Piste attribuutit x ja y sekä metodi siirra() siis periytyvät aliluokkaan VariPiste
  - Perityviä attribuutteja metodeja ei merkitä aliluokan kohdalle
- Jos ollaan tarkkoja, Piste-luokan metodi toString periytyy myös VariPiste-luokalle, joka *syrjäyttää* (engl. override) perimänsä omalla toteutuksella
  - Korvaava toString()-metodi merkitään aliluokkaan VariPiste
- Eli kuviosta on pääteltävissä, että VariPisteellä on:
  - Atribuutit x ja y sekä metodi siirra perittynä
  - Attribuutti vari, jonka se määrittelee itse
  - Itse määritelty metodi toString joka syrjäyttää yliluokalta perityn
  - Koodista nähdään, että korvaava metodi käyttää yliluokassa määriteltyä metodia



Mitä kaikkea periytyy?

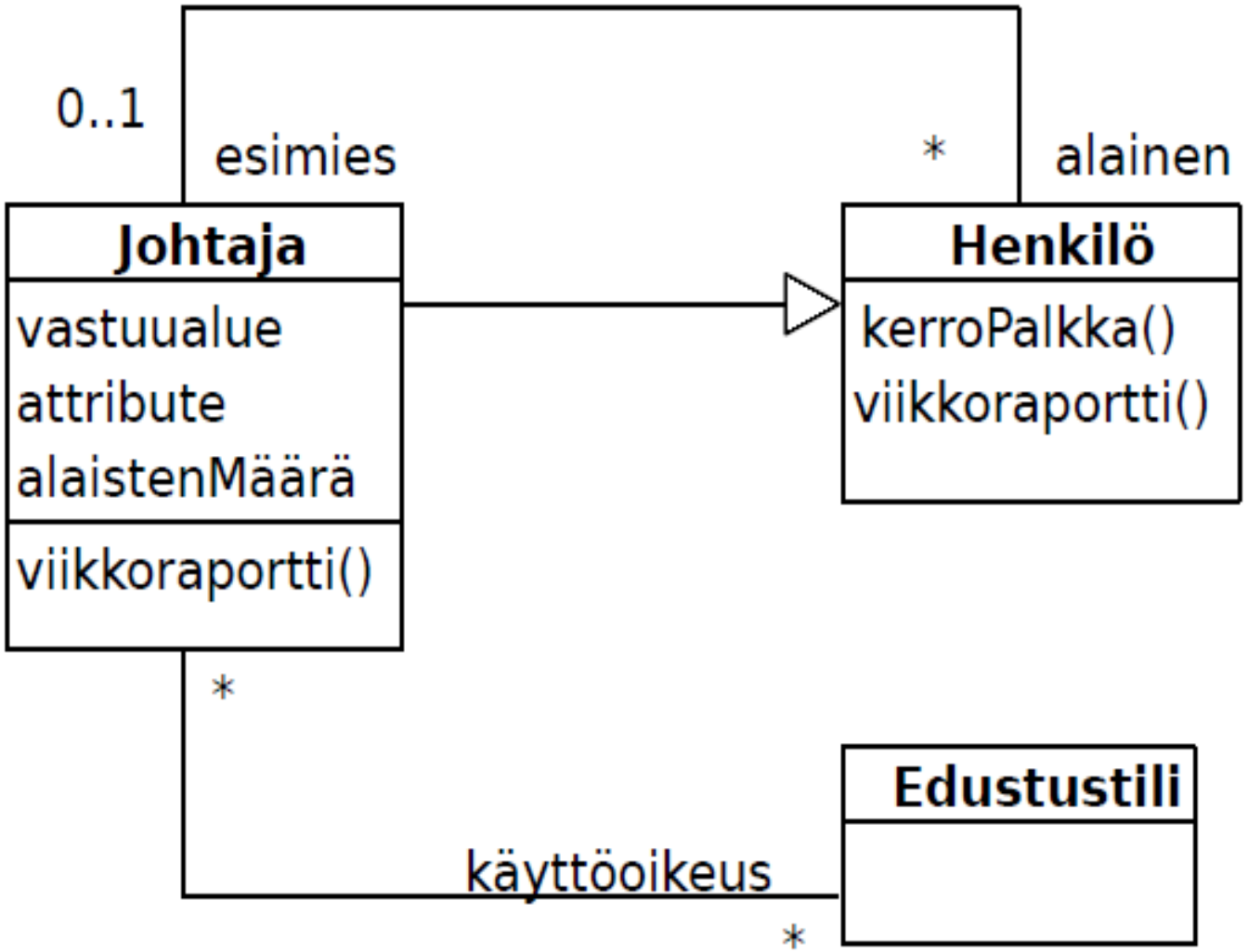
- Luokat Koira ja Lehmä ovat molemmat luokan Kotieläin aliluokkia
- Jokaisella kotieläimellä on omistajana joku Henkilö-olio
- Koska omistaja liittyy kaikkiin kotieläimiin, merkitään yhteys Kotieläin- ja Henkilö-luokkien välille
  - **Yhteydet periytyvät aina aliluokille**, eli Koira-olioilla ja Lehmä-olioilla on omistajana yksi Henkilö-olio
- Ainoastaan Lehmä-olioilla on lypsäjiä
  - Yhteys lypsää tuleekin Lehmän ja Henkilön välille



Aliluokan ja yliluokan välinen yhteys

- Yrityksen työntekijää kuvaa luokka Henkilö
  - Henkilöllä on metodit kerroPalkka() ja viikkoraportti()
- Johtaja on Henkilön aliluokka
  - Johtajalla on alaisena useita henkilöitä
  - Henkilöllä on korkeintaan yksi johtaja esimiehenä
  - Johtajalla voi olla käyttöoikeuksia Edustustileihin
  - Edustustilillä on useita käyttöoikeuden omaavia johtajia
  - Johtajan viikkoraportti on erilainen kuin normaalin työntekijän viikkoraportti
- Tilannetta kuvaava luokkakaavio seuraavalla sivulla

Osa yrityksen luokkakaaviota



Aliluokan ja yliluokan välinen yhteys

- Johtaja siis perii kaiken Henkilöltä
  - Henkilö on *alainen*-roolissa yhteydessä nollaan tai yhteen Johtajaan
  - Tästä seuraa, että *myös Johtaja-olioilla on sama yhteys, eli myös johtajilla voi olla johtaja!*
- Metodi viikkoraportti on erilainen johtajalla kuin muilla henkilöillä, siispä Johtaja-luokka korvaa Henkilö-luokan metodin omallaan
- Esim. Henkilö-luokan metodi viikkoraportti():

  Kerro ajankäyttö työtehtäviin
- Johtaja-luokan korvaama metodi viikkoraportti():

  Kerro ajankäyttö työtehtäviin

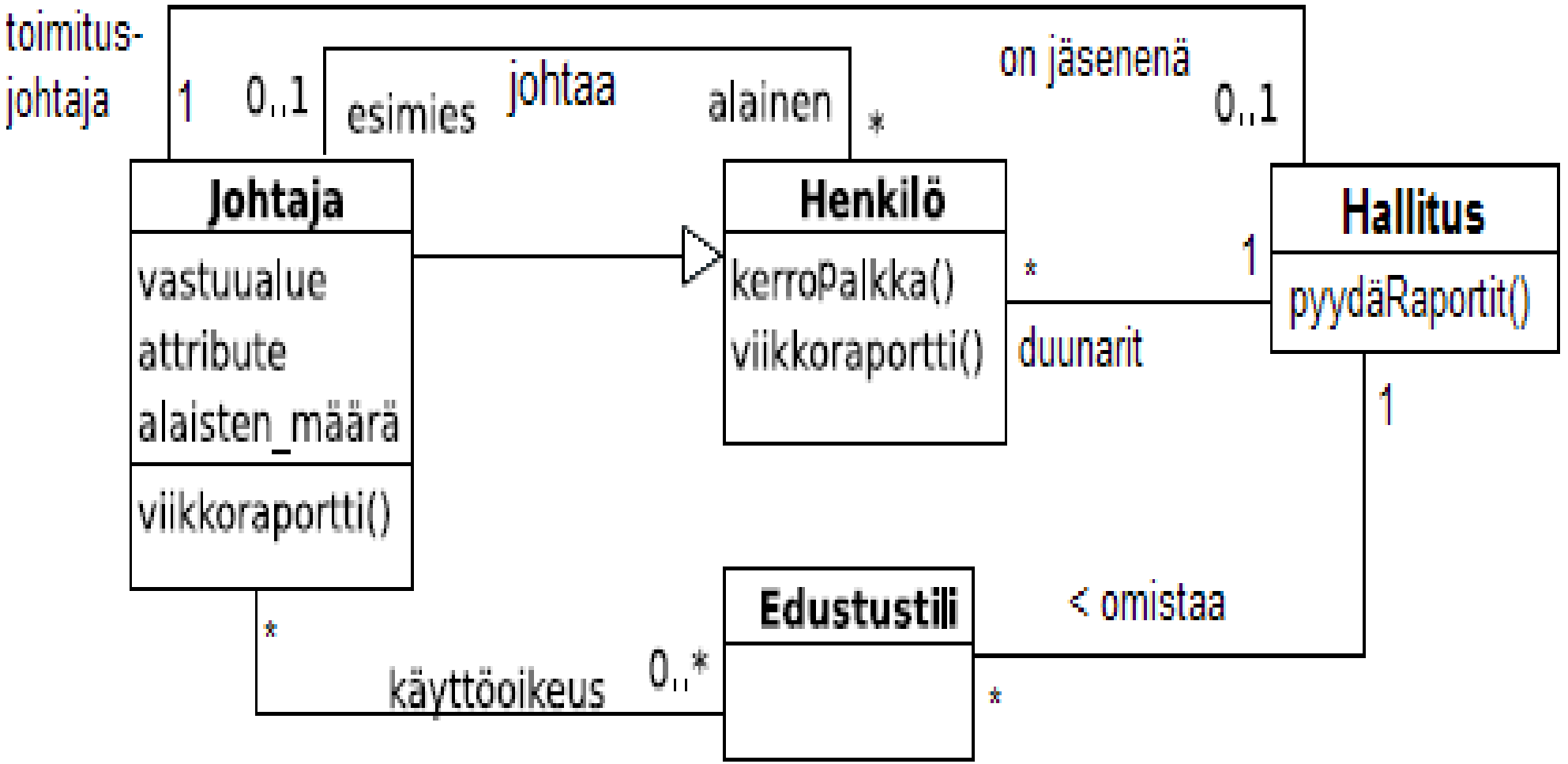
  Laadi yhteenveto alaisten viikkoraporteista

  Raportoi edustustilin käytöstä
- Yhteys käyttöoikeus Edustustileihin voi siis ainoastaan olla niillä henkilöillä jotka ovat johtajia

Laajennetaan mallia

- Yrityksen hallitus koostuu ulkopuolisista henkilöistä (joita ei sisällytetä malliin) ja yrityksen toimitusjohtajasta joka siis kuuluu henkilöstöön
  - Hallitus on edustustilien omistaja
  - Hallitus ”tuntee” toimitusjohtajaa lukuunottamatta kaikki työntekijät, myös normaalit johtajat ainoastaan Henkilö-olioina
- Hallitus pyytää työntekijöiltä viikkoraportteja
  - Viikkoraportin tekevät kaikki paitsi toimitusjohtaja

Hallitus mukana kuvassa



Olio tietää luokkansa

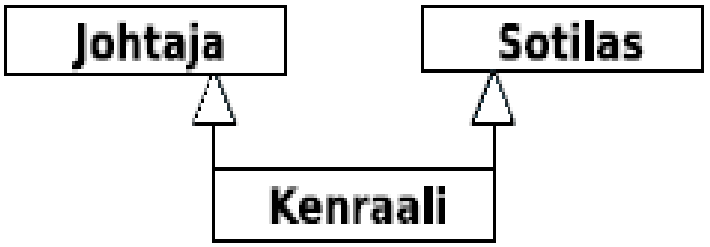
- Hallitus siis tuntee kaikki työntekijänsä, mutta ei erittele ovatko he normaaleja työntekijöitä vai johtajia
  - Hallituksen koodissa kaikkia työntekijöitä pidetään Henkilö-oliosta koostuvassa listalla *duunarit*. Johtajathan ovat myös henkilöitä!
- Hallituksen ei siis ole edes tarvetta tuntea kuka on johtaja ja kuka ei
- Pyytäessään viikkoraporttia, hallitus käsittelee kaikkia samoin:

```
public class Hallitus{

  private ArrayList<Henkilo> duunarit ;          // attribuutti, joka sisältää kaikki työntekijät

  private Johtaja toimitusjohtaja;               // attribuutti, joka tietää toimitusjohtajan

  public void pyydaRaportit(){

     for ( d : duunarit ) {   if ( d != toimitusjohtaja ) d.viikkoraportti()  }

  }
```

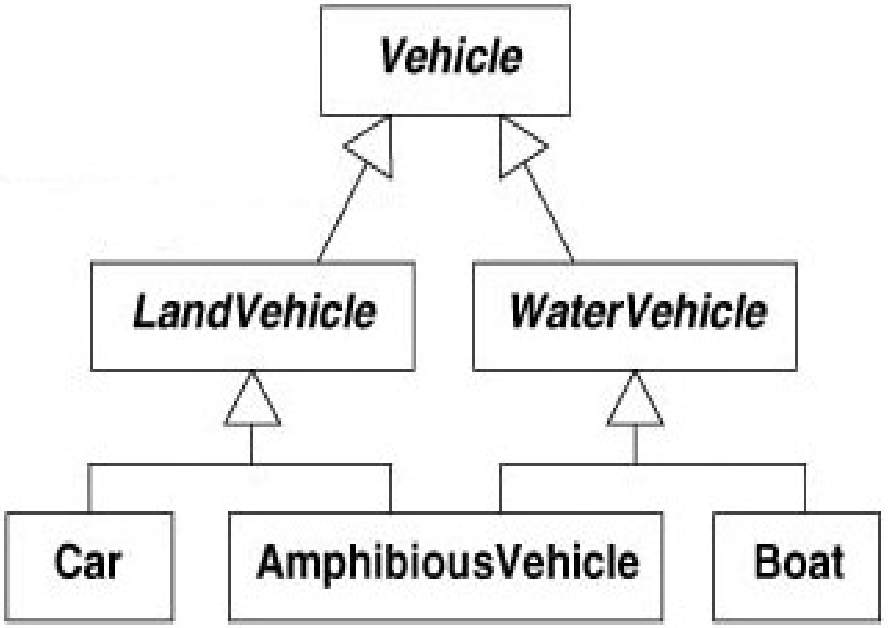
- Jokainen duunari tuntee ”oikean” luokkansa
- Kun hallitus kutsuu duunarille metodia viikkoraportti(), jos kyseessä on normaali henkilö, suoritetaan henkilön viikkoraportointi, jos taas kyseessä on johtaja, suoritetaan johtajan viikkoraportti (*polymorfismia!*)

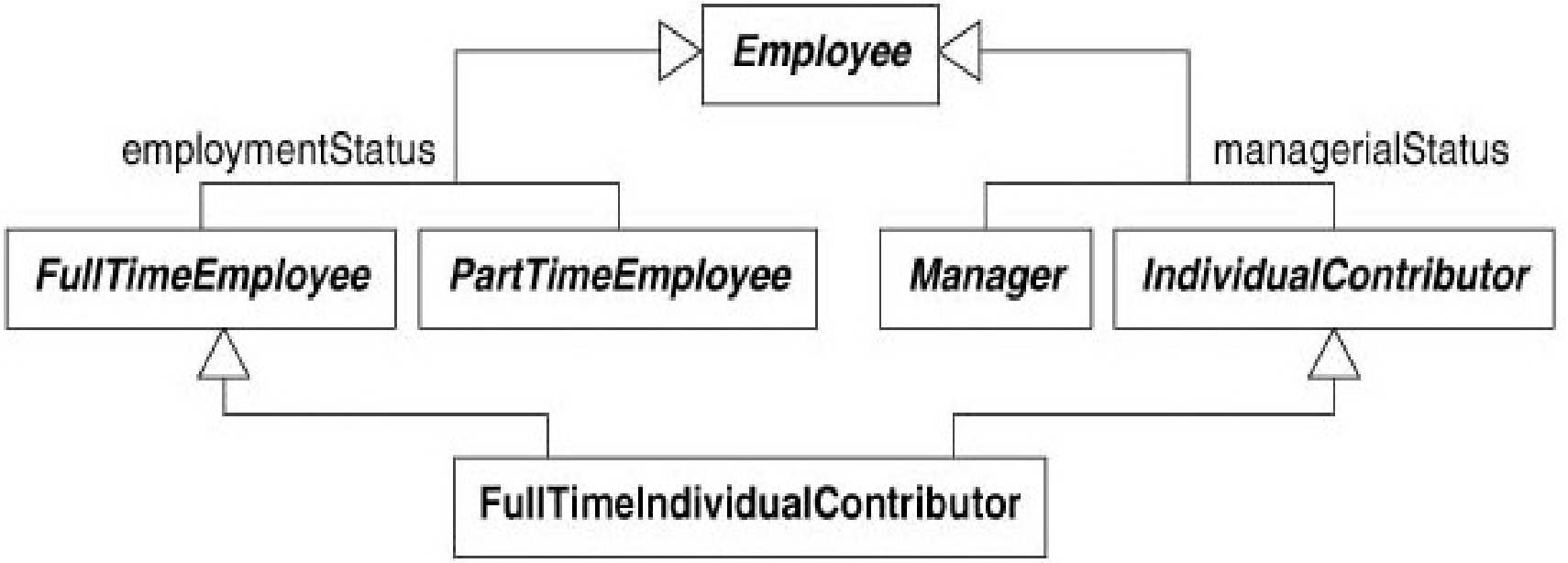
Moniperintä

- Joskus tulee esiin tilanteita, joissa yhdellä luokalla voisi kuvitella olevan useita yliluokkia
- Esim. kenraalilla on sekä sotilaan, että johtajan ominaisuudet
  - Voitaisiin tehdä luokat Johtaja ja Sotilas ja periä Kenraali näistä
- Kyseessä *moniperintä* (engl. multiple inheritance)
- Seuraavalla sivulla: Kulkuneuvo jakautuu maa- ja merikulkuneuvoksi
  - Auto on maakulkuneuvo, vene merikulkuneuvo, amfibio sekä maa- että merikulkuneuvo
- Työntekijät voi jaotella kahdella tavalla:
  - Pää- ja sivutoimiset
  - Johtajat ja normaalit
  - Yksittäinen työntekijä voi sitten olla esim. päätoiminen johtaja



Lisää moniperintää





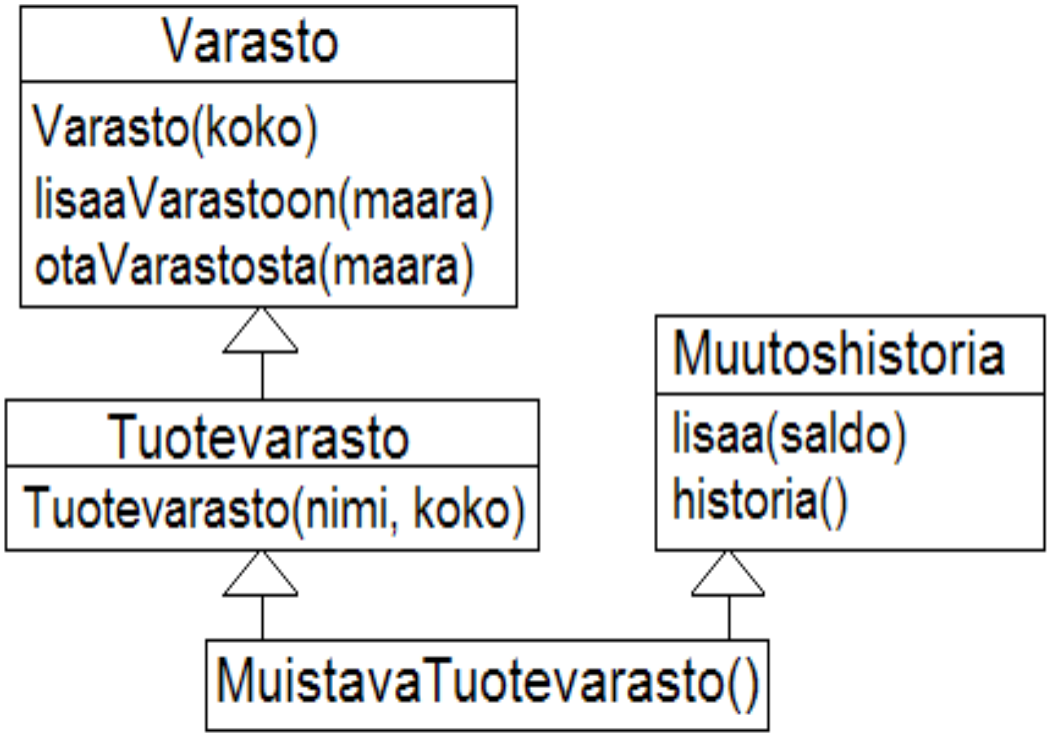
Moniperinnän ongelmat

- Moniperintä on monella tapaa ongelmallinen asia ja useat kielet, kuten Java eivät salli moniperintää
  - C++ sallii moniperinnän
  - ”moderneissa” kielissä kuten Python, Ruby ja Scala on olemassa ns. mixin-mekansismi, joka mahdollistaa ”hyvinkäyttäytyvän” moniperintää vastavan mekanismin
- Onkin viisasta olla käyttämättä moniperintää ja yrittää hoitaa asiat muin keinoin
- Näitä muita keinoja (jos unohdetaan mixin-mekanismi) ovat:
  - Moniperiytymisen korvaaminen yhteydellä, eli käytännössä ”liittämällä” olioon toinen olio, joka laajentaa alkuperäisen olion toiminnallisuutta
  - Javasta löytyvät rajapintaluokat (interface) toimivat joissain tapauksessa moniperinnän korvikkeena
- Ohjelmoinnin jatkokurssin viikon 4 laskareissa, ks http://www.cs.helsinki.fi/group/java/s12/ohja/materiaali.html#w4e30 toteutetaan MuistavaTuotevarasto, joka on luokka johon lisätään toiminnallisuutta perimisen sijaan liittämällä siihen toinen olio

Muistava tuotevarasto

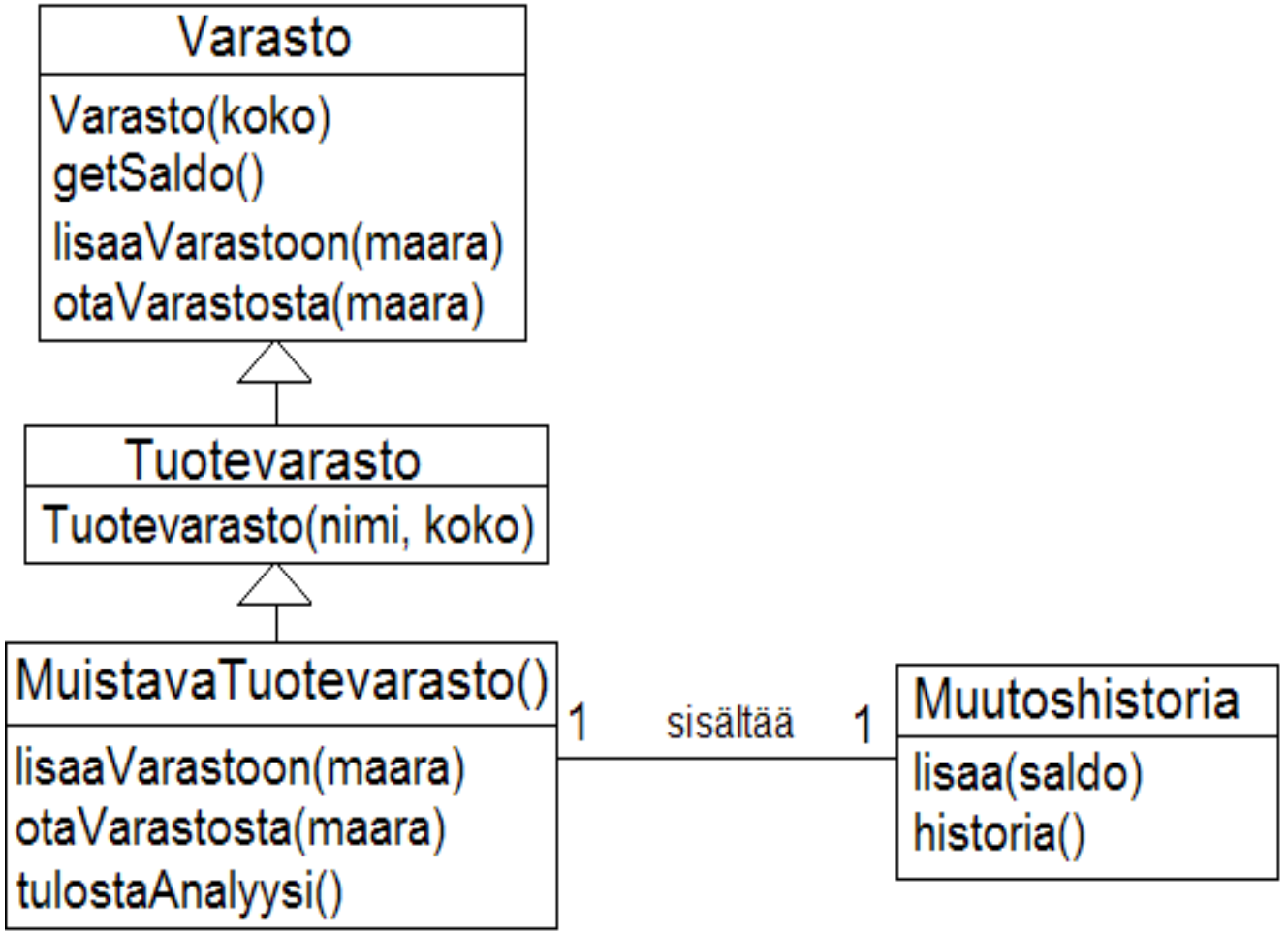
- Luokka Tuotevarasto toteutetaan luokka Varasto
  - Tuotteelle lisätään nimi
- Ensin toteutetaan luokka Muutoshistoria
  - Käytännössä kyseessä on lista double-lukuja, joiden on tarkotus kuvata peräkkäisiä varastosaldoja
- Sitten MuistavaTuotevarasto joka yhdistää edellisten toiminnallisuuden
- Joku C++-ohjelmoija soveltaisi tilanteessa kenties moniperintää:

EI NÄIN!



Muistava tuotevarasto

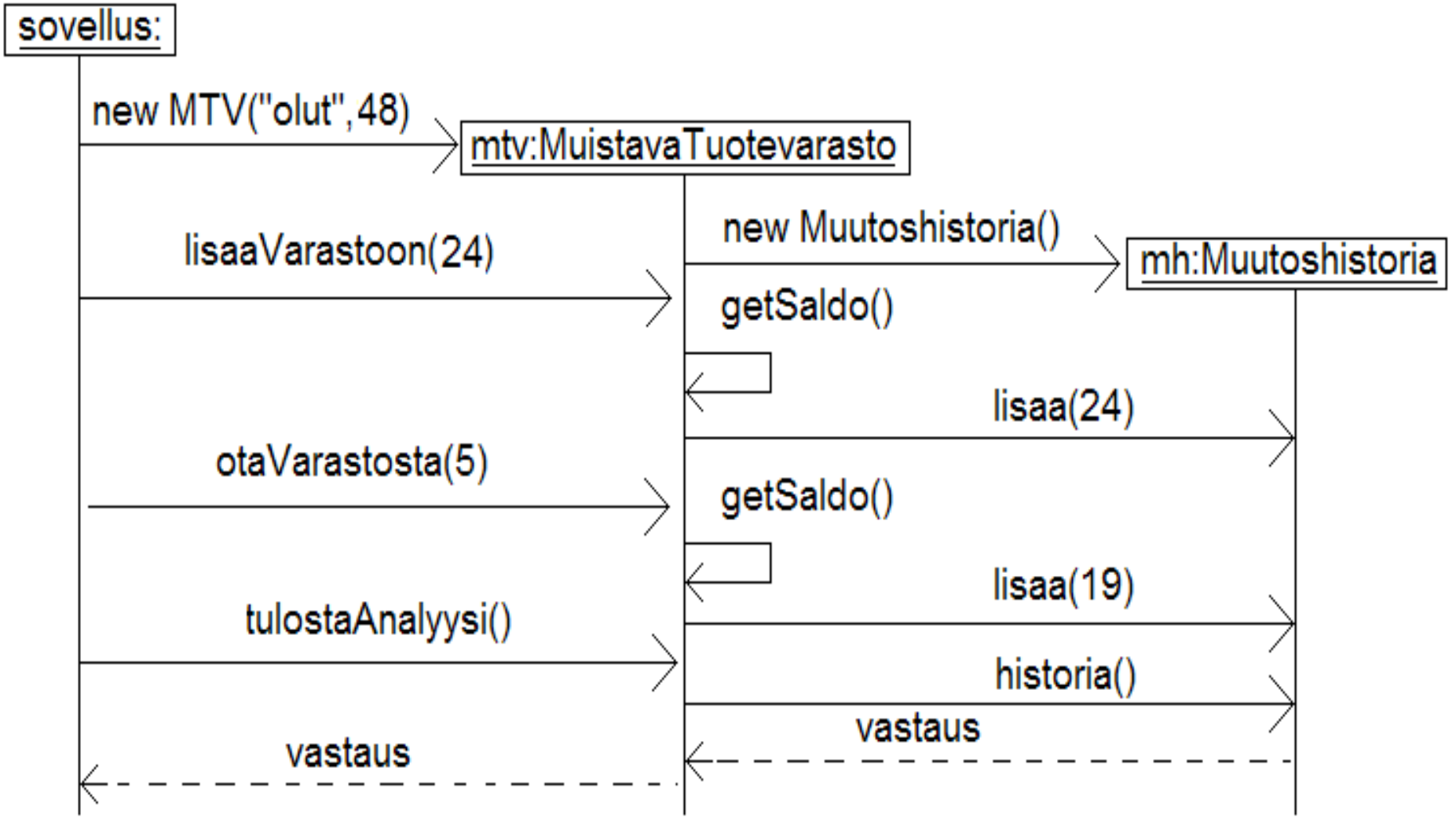
- Javassa ei moniperintää ole, ja vaikka olisikin, on parempi liittää ”muistamistoiminto” muistavaan tuotevarastoon erillisenä oliona:



- Aina kun muistavan tuotevaraston saldo päivittyy (metodien lisaaVarastoon ja otaVarastosta yhteydessä), samalla laitetaan uusi saldo muutoshistoriaan

Muistavan varaston toimintaa kuvaava sekvenssikaavio

- Sovellus luo muistavan tuotevaraston joka tallettaa olutta ja kapasiteetti on 48
- MuistavaTuotevarasto luo käyttöönsä Muutoshistoriaolion
- Aina kun varaston tilanne muuttuu, selvitetään saldo ja välitetään se muutoshistorialle
  - Kaavio ei sisällytö seuraavalla sivulla koodissa näkyviä super-kutsuja
- Analyysin tulostus delegoituu muutosvaraston hoidettavaksi



Muistavan tuotevaraston koodihahmotelma

```
public MuistavaTuotevarasto extends Tuotevarasto {
    private Muutoshistoria varastotilanteet;
    public MuistavaTuotevarasto(String tuote, double koko){
        super(tuote, koko);
        varastotilanteet = new Muutoshistoria();   // luodaan oma kirjanpito-olio
    }
    public void lisaaVarastoon(double maara){
        super.lisaaVarastoon(maara);              // yliluokan metodi päivittää varastotilanteen
        double sadlo = getSaldo();
        varastotilanteet.lisaa( saldo );           // kerrotaan saldo kirjanpito-oliolle
    }
    public String tulostaAnalyysi() {
        return varastotilanteet.historia();        // delegoidaan raportin luominen kirjanpito-oliolle
    }
}
```

Kaksi tärkeää oliosuunnittelun periaatetta

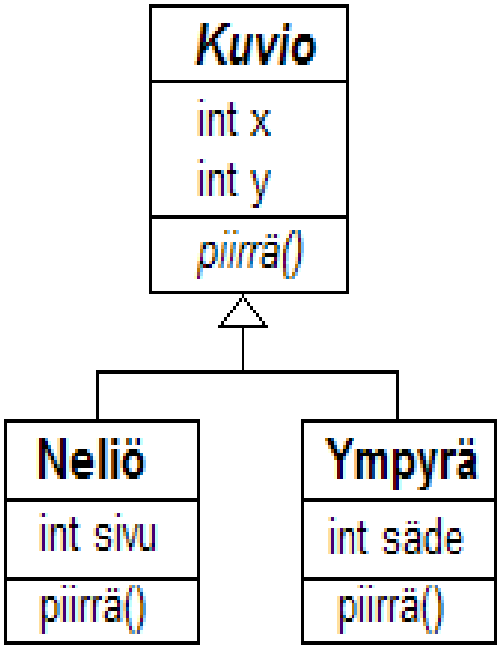
- Esimerkissä toteutuu kaksi tärkeää oliosuunnittelun periaatetta
- **Single responsibility**
  - Tarkoittaa karkeasti ottaen että **oliolla tulee olla vain yksi vastuu**
  - MuuttuvaTuotevarasto toteuttaa periaatetta koska sen vastuulla on vain varaston nykyisen tilanteen ylläpito
  - Se *delegoi* vastuun aikaisempien varastosaldojen muistamisesta Muutoshistoria-oliolle
- **Favour composition over inheritance**
  - eli **suosi yhteistominnassa toimivia oliota perinnän sijaan**
  - Perinnällä on paikkansa, mutta sitä tulee käyttää harkiten
  - Esimerkissä käytettiin perintää järkevästi
  - Jos olisi moniperitty Tuotevarasto ja Muutoshistoria, olisi muodostettu luokka joka rikkoo single responsibility -periaatteen
- Onko näissä periaatteissa järkeä: Kyllä, ne lisäävät ohjelmien ylläpidettävyyttä
- Pitääkö periaatteita noudattaa: useimmiten. Joskus voi olla jonkun muun periaatteen nojalla viisasta rikkoa jotain periaatetta...
- Muitakin periaatteita on, joitakin niistä esitellään myöhemmin kurssilla

Abstraktit luokat

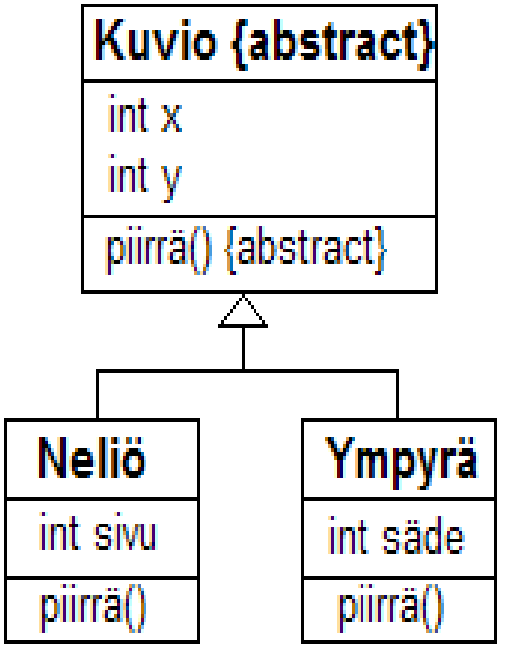
- Yliluokalla Kuvio on sijainti, joka ilmaistaan x- ja y-kordinaatteina sekä metodi piirrä()
- Kuvion aliluokkia ovat Neliö ja Ympyrä
  - Neliöllä on sivun pituus ja Ympyrällä säde
- Kuvio on nyt pelkkä abstrakti käsite, Neliö ja Ympyrä ovat konkreettisia kuvioita jotka voidaan piirtää ruudulle kutsumalla sopivia grafiikkakirjaston metodeja
- Kuvio onkin järkevä määritellä *abstraktiksi luokaksi*, eli luokaksi josta ei voi luoda instansseja, joka ainoastaan toimii sopivana yliluokkana konkreettisille kuvioille
- Kuviolla on attribuutit x ja y, mutta metodi piirrä() on *abstrakti metodi*, eli Kuvio ainoastaan määrittelee metodin nimen ja parametrien sekä paluuarvon tyypit, mutta *metodille ei anneta mitään toteutusta*
- Kuvion perivät luokat Neliö ja Ympyrä antavat toteutuksen abstraktille metodille
  - Neliö ja Ympyrä ovatkin normaaleja luokkia, eli niistä voidaan luoda olioita
- Luokkakaavio seuraavalla sivulla

Abstrakti luokka

- Luokkakaaviossa on kaksi tapaa merkitä abstraktius
  - Abstraktin luokan/metodin nimi kursiivilla, tai
  - liitetään abstraktin luokan/metodin nimeen tarkenne {abstract}



tai:




```
public abstract class Kuvio{
    private int x;
    private int y;
    public abstract void piirrä();
}
```

```
public class Neliö extends Kuvio {
        private int sivu;
        public void piirrä(){
        graphics.drawRect(x, y, x+sivu, y+sivu);
    }
```

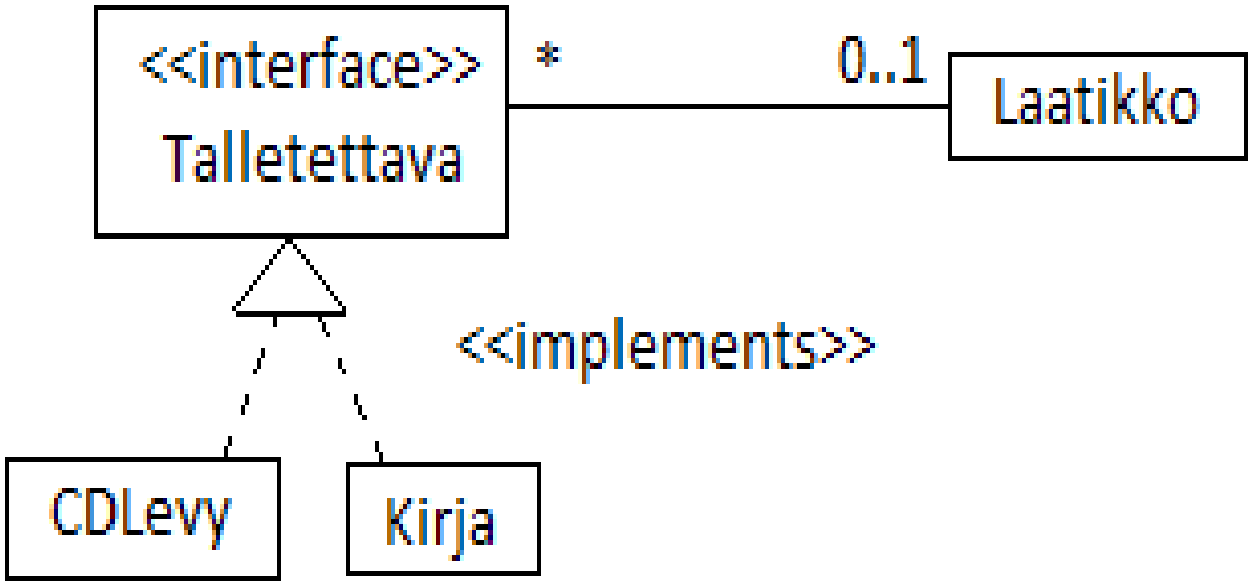
Rajapinta

- Abstraktiudella ei ole mitään tekemistä moniperiytymisongelman kanssa, eli periä saa vain yhden luokan oli se abstrakti tai ei
- Javan *rajapinta* (interface) on ikäänkuin abstrakti luokka joka ei sisällä attribuutteja ja jonka kaikki metodit ovat abstrakteja
- Rajapintaluokka siis listaa ainoastaan joukon metodien nimiä
- Yksi luokka voi *toteuttaa* useita rajapintoja
    - Ja sen lisäksi vielä periä yhden luokan

- Perimällä luokka saa yliluokasta attribuutteja ja metodeja
- Rajapinnan toteuttaminen on pikemminkin velvollisuus
    - Jos luokka toteuttaa rajapinnan, sen täytyy toteuttaa kaikki rajapinnan määrittelemät operaatiot
- Tai toisinpäin ajateltuna, **rajapinta on sopimus**
    - **toteuttaja lupaa toteuttaa ainakin rajapinnan määrittelemät operaatiot**

Tuttu esimerkki rajapintaluokan käytöstä

- Mallinnetaan Ohjelmoinnin jatkokurssin viikon 2 tehtävä Tavaroita ja Laatikoita
  - http://www.cs.helsinki.fi/group/java/s12/ohja/materiaali.html#w2e11
  - Kuva ei ota huomioon tehtävän viimeistä kohtaa!
- Rajapintaluokka kuvataan luokkana, johon liitetään tarkenne <<interface>>
- Rajapinnan toteuttaminen merkitään kuten periminen, mutta katkoviivana
  - Voidaan tarkentaa tarkenteella <<implements>>



Toinen esimerkki rajapintaluokan käytöstä

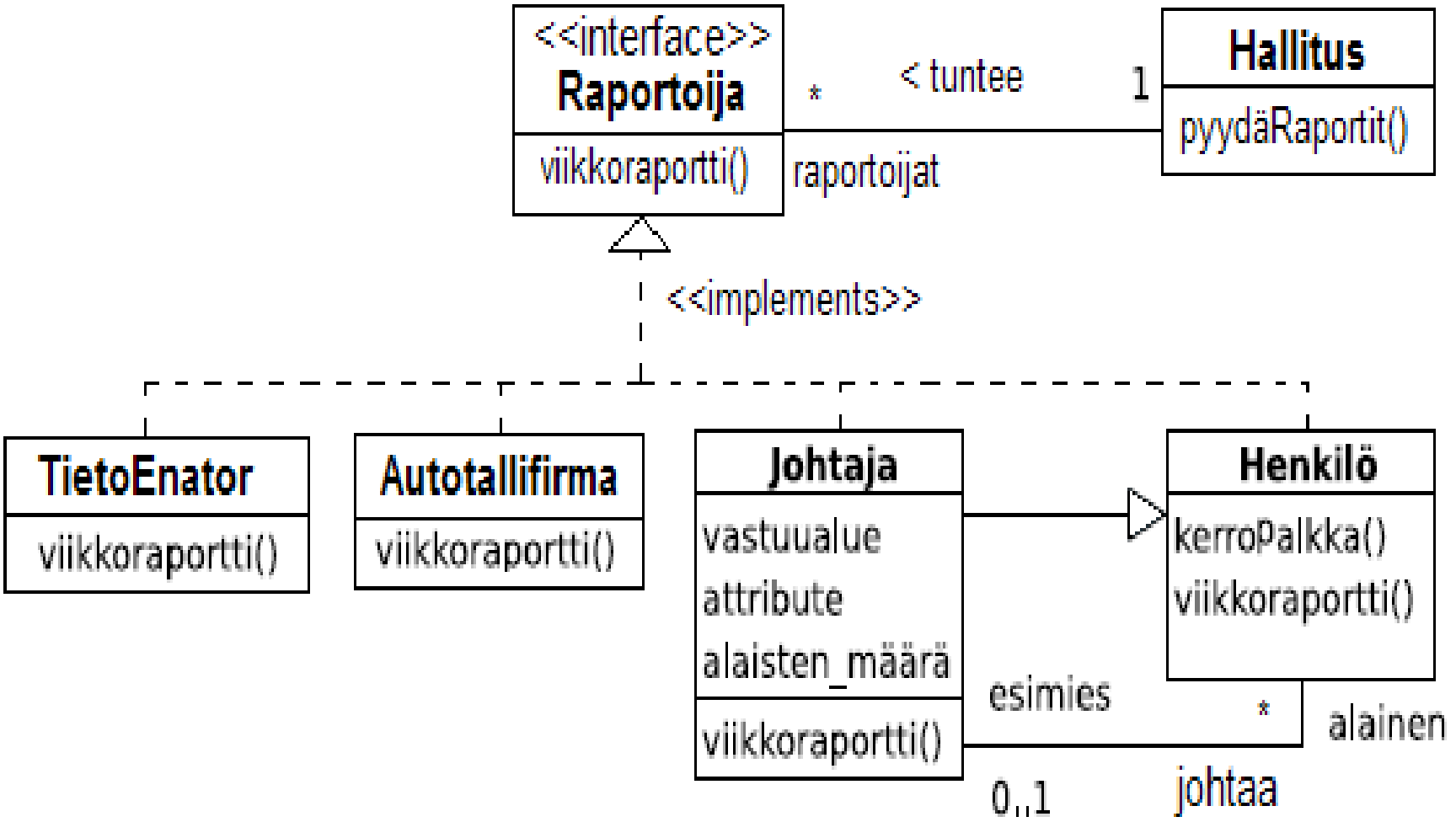
- Palataan muutaman sivun takaiseen yritysesimerkkiin
- Tilanne on nyt se, että yritys on ulkoistanut osan toiminnoistaan
- Hallitus on edelleen kiinnostunut viikkoraporteista
  - Hallitusta ei kuitenkaan kiinnosta se, tuleeko viikkoraportti omalta henkilöstöltä vai alihankkijalta
- Muuttuneessa tilanteessa hallitus tunteekin ainoastaan joukon raportointiin kykeneviä olioita
  - Jotka voivat olla Henkilöitä, Johtajia tai alihankkijoita
  - Kukin näistä toteuttaa metodin viikkoraportti() omalla tavallaan
- Tilanne kannattaa hoitaa määrittelemällä *rajapintaluokka* ja vaatia, että kaikki hallituksen tuntemat tahot toteuttavat rajapinnan

```
public interface Raportoija {

    void viikkoraportti()

}
```

- Hallitukselle riittääkin, että se tuntee joukon Raportoijia (eli rajapinnan toteuttajia)

- Hallituksen koodi voisi sisältää seuraavan:

```
public class Hallitus {
    private List<Raportoija> raportoijat;
    public void pyydaRaportit(){
       for ( r : raportoijat ) {   r.viikkoraportti()  }
   }
```



Perimisen kolmet kasvot

- Kai Koskimiehen 2000-luvun alussa ilmestyneen Oliokirjan mukaan periytymisellä on ”kahdet kasvot”:
- Uusiokäyttömekanismi
  - Periytymisellä voidaan toteuttaa aiemmin ohjelmoitujen välineiden uudelleenkäyttöä, välineitä täydennetään ja erityistetään uusien tarpeiden mukaisiksi (vrt Varasto – Tuotevarasto – MuistavaTuotevarasto)
- Käsitteiden luokittelu
  - Ongelmamaailmaa mallinnetaan peritymisen käsittein, yliluokka-aliluokka -suhde vastaa yleinen-erityinen -suhdetta. Eläin on kissan yleistys, lehmä on naudan erikoistapaus, jne.
- Kolmannet kasvot, *ne ylivoimaisesti tärkeimmät,* eivät käy kunnolla esille Koskimiehen esityksestä:
  - Näillä kasvoilla ei ole niin selkeää nimitystä kuin edellisillä kahdella
  - Seuraavalla kalvolla hiukan hahmotellaan mitä asialla tarkoitetaan
  - Asiaan liittyy *polymorfismi*
  - Juuri nämä kolmannet kasvot ovat se seikka, mikä on merkittävä muokattavuuden ja uusiokäytön mahdollistaja olio-ohjelmoinnissa

- Muutamassa esimerkissä näimme, miten on mahdollista käsitellä olioita tuntematta niiden varsinaista luokkaa
  - tunnetaan ainoastaan rajapinta tai yliluokka, eli asiat jotka oliot ainakin osaavat tehdä
- Esim. Hallituksen ei tarvitse tietää raportoijista juuri mitään
  - Koska raportoijat toteuttavat Raportoija-rajapinnan, on selvää, että niille voidaan kutsua metodia viikkoraportti()
  - Kukin olio tietää oman tyyppinsä ja osaa suorittaa omalla tavalla toteuttamansa viikkoraportti()-metodin (polymorfismin ansiosta)
  - Siispä kaikki raportoivat oikein vaikka Hallitus ei tiedä raportoijista mitään
  - Tämä mahdollistaa, että tulevaisuudessa voidaan ohjelmoida uusia luokkia, jotka saadaan liitettyä Hallitukseen kunhan ne vaan toteuttavat rajapinnan Raportoija
- Eli rajapintojen tai yliluokkien avulla saadaan **rajattua monimutkaisuutta** pois sieltä, missä siitä ei tarvitse välittää
  - Muutokset eivät haittaa niin kauan kun rajapinta toteutetaan asiallisesti
- Rajapinta/yliluokka tarjoaa **laajennuspaikan tulevaisuuden** varalle
  - Vanha ohjelma ei mene rikki uusista aliluokista tai rajapinnan toteuttajista jos ne toimivat oletusten mukaan

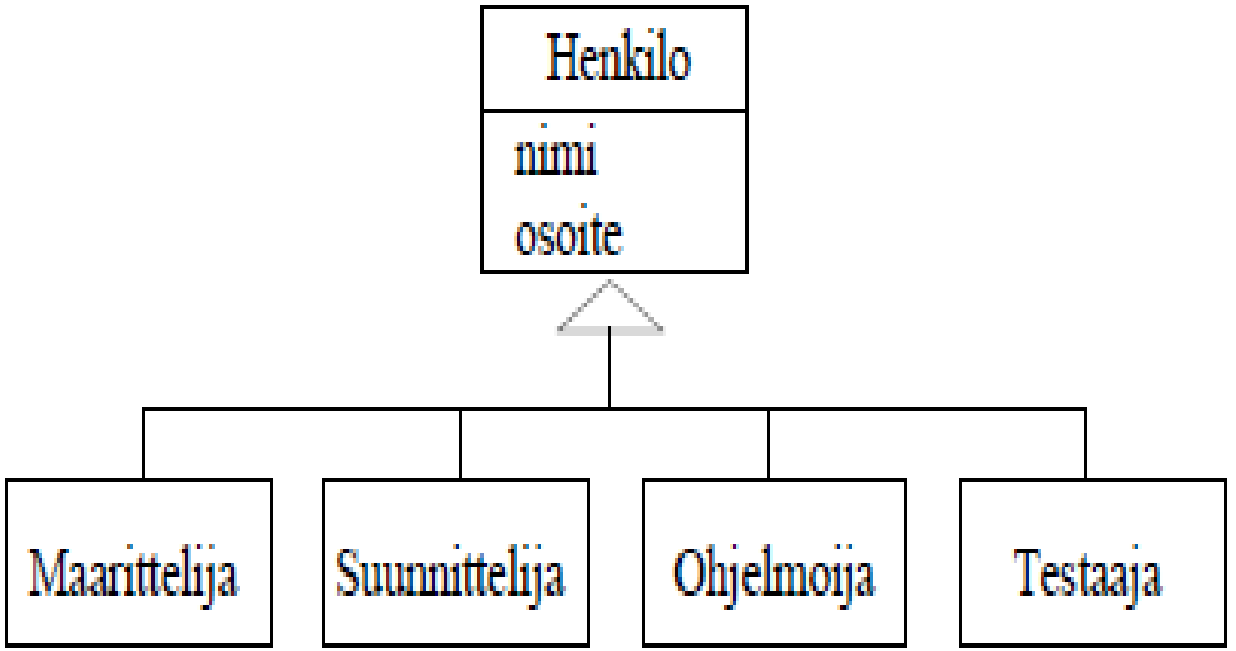
Kolmas oliosuunnittelun periaate

- Vähän aikaa sitten mainittiin kaksi tärkeää oliosuunnittelun periaatetta
  - Single responsibility
  - Favour composition over inheritance
- Kolmas ”kulmakivi on seuraava”:
  - **Program to an interface, not to an Implementation**, tai toisin ilmaistuna
  - **Depend on Abstractions, not on concrete implementations**
- Kyse on juuri edellisellä sivulla hahmotellusta asiasta:
  - Laajennettavuuden kannalta ei ole hyvä idea olla riippuvainen konkreettisista luokista sillä ne saattavat muuttua
  - Parempi on tuntea vain rajapintoja (tai abstrakteja luokkia) ja olla tietämätön siitä mitä rajapinnan takana on
- Tämä voidaan ajatella ”laajennettuna kapselointina”
  - Kapselointi piilottaa olioiden sisäisen toteutuksen esim. määrittelemällä instanssimuuttujat näkyvyydeltään privateiksi
  - Jos tunnetaan vaan rajapinta, ”kapseloituu” koko takana oleva olio ja tämä taas avaa uudenlaisen joustavuuden sillä rajapinnan toteuttava luokka on helppo muuttaa vaikuttamatta sen käyttäjiin

Perimisen virheellisiä ja oikeaoppisia käyttötapoja

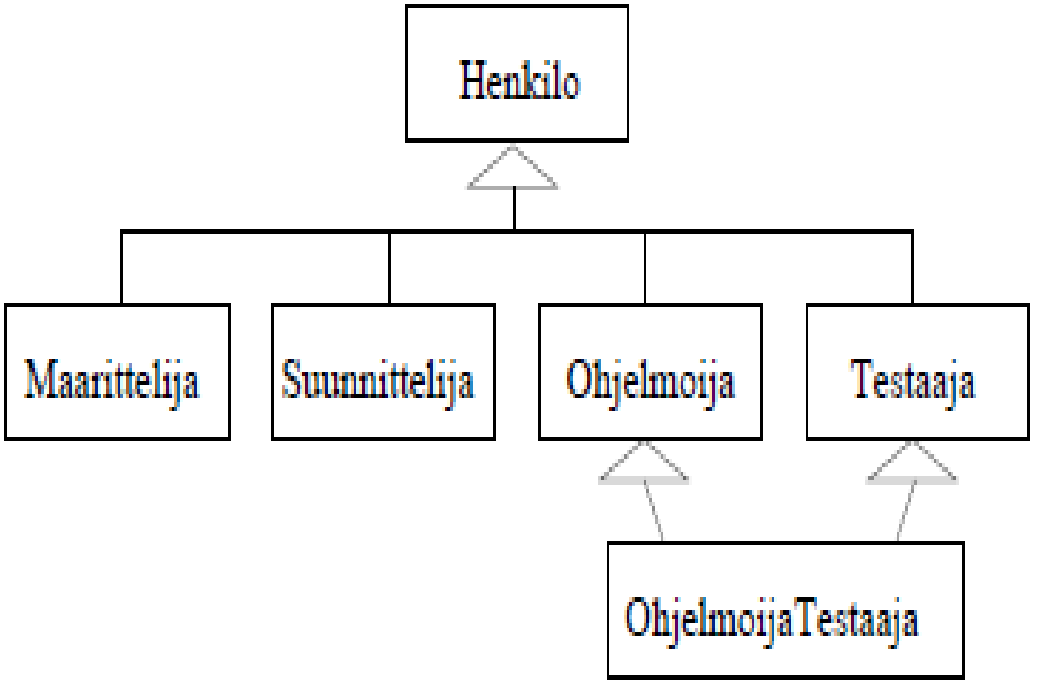
Esimerkki periytymisen virheellisestä käyttöyrityksestä

- Ohjelmistoyrityksessä työskentelee henkilöitä erilaisissa tehtävissä:
  - Määrittelijöinä
  - Suunnittelijoina
  - Ohjelmoijina
  - Testaajina
- Yritys toteuttaa omia tarpeitaan varten henkilöstöhallintajärjestelmän
- Ensimmäinen yritys mallintaa yrityksen työntekijöitä alla
  - Vaikuttaa loogiselta: esim. testaaja on Henkilö...



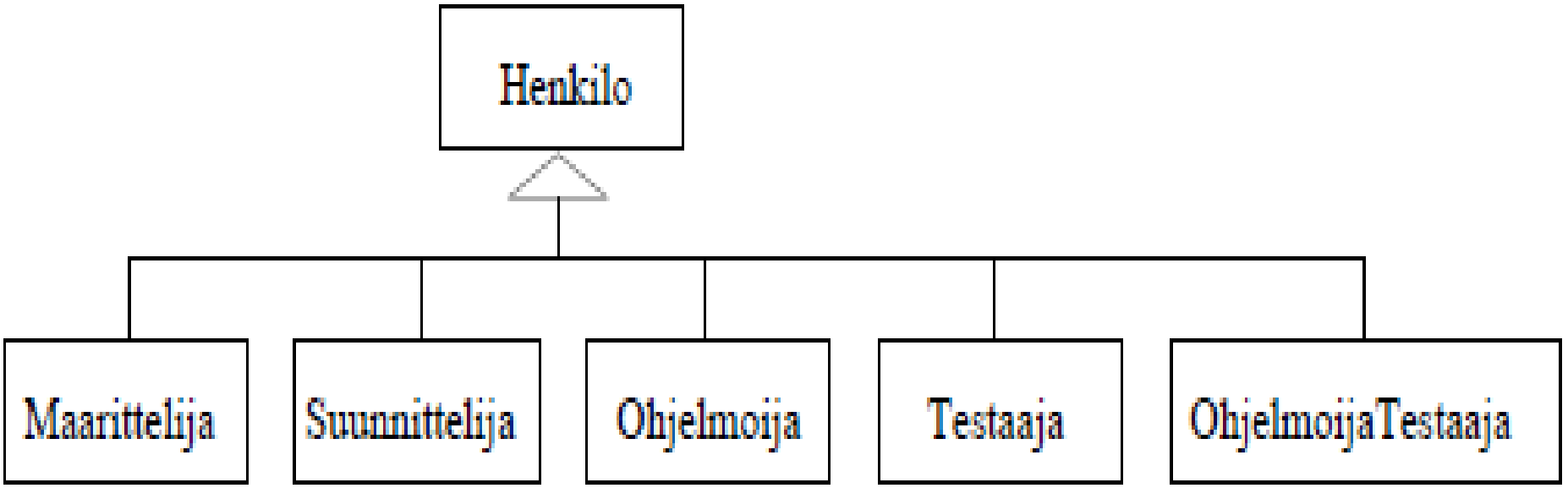
Ongelmia

- Entä jos työntekijällä on useita tehtäviä hoidettavanaan?
  - Esim. ohjelmoiva testaaja
- Yksi vaihtoehto olisi mallintaa tilanne käyttämällä *moniperintää* alla olevan kuvan mukaisesti
- Tämä on huono idea muutamastakin syystä
  - Jokaisesta työtehtäväkombinaatiosta pitää tehdä oma aliluokka
    - jos kaikki kombinaatiot otetaan huomioon, yhteensä luokkia tarvittaisiin 10 kappaletta
  - Kuten mainittua, esim. Javassa ei ole moniperintää



Huono ratkaisuyritys

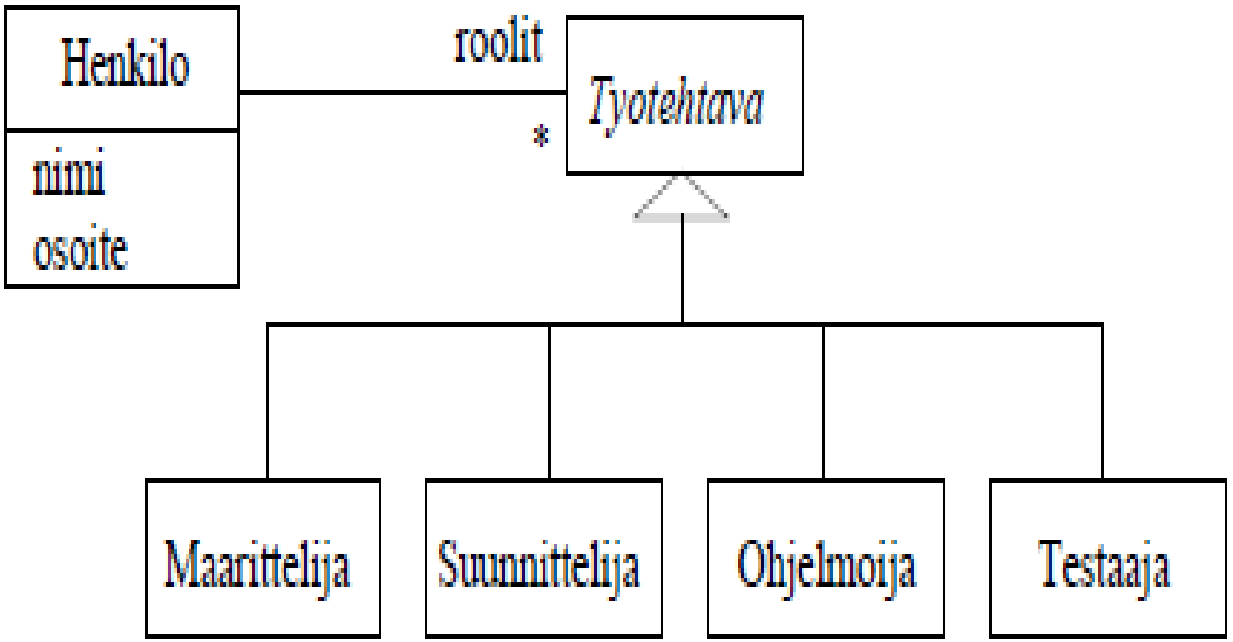
- Jos toteutuskieli ei tue moniperintää, yksi vaihtoehto on jokaisen työyhdistelmän kuvaaminen omana suoraan Henkilön alla olevana aliluokkana
  - Erittäin huono ratkaisu: nyt esim. OhjelmoijaTestaaja ei peri ollenkaan Ohjelmoija- eikä Testaaja-luokkaa
    - Seurauksena se, että samaa esim. Ohjelmoija-luokaan liittyvää koodia joudutaan toistamaan moneen paikkaan
- Yksi suuri ongelma tässä ja edellisessä ratkaisussa on miten hoidetaan tilanne, jossa *henkilö siirtyy esim. suunnittelijasta ohjelmoijaksi*
  - Esim. Javassa olio ei voi muuttaa luokkaansa suoritusaikana: Suunnittelijaksi luodut pysyvät suunnittelijoina!



Roolin kuvaaminen erillisenä luokkana

- Henkilön työtehtävää voidaan ajatella henkilön *rooliksi* yrityksessä
- Vaikuttaa siltä, että henkilön eri roolien mallintaminen ei kunnolla onnistu periytymistä käyttäen

- Parempi tapa mallintaa tilannetta on pitää luokka Henkilö kokonaan erillisenä ja *liittää* työtehtävät, eli henkilön *roolit*, siihen *erillisinä luokkina*
- Ratkaisu seuraavalla sivulla
  - Luokka Henkilö kuvaa siis ”henkilöä itseään” ja sisältää ainoastaan henkilön ”persoonaan” liittyvät tiedot kuten nimen ja osoitteen
  - Henkilöön liittyy yksi tai useampi Työtehtävä eli työntekijärooli
  - Työntekijäroolit on mallinnettu periytymishierarkian avulla, eli jokainen henkilöön liittyvä rooli on jokin konkreettinen työntekijärooli, esim. Ohjelmoija tai Testaaja
- Oikeastaan kaikki ongelmat ratkeavat tämän ratkaisun myötä
  - Henkilöön voi liittyä nyt kuinka monta roolia tahansa
  - Henkilön rooli voi muuttua: poistetaan vanha ja lisätään uusi rooli



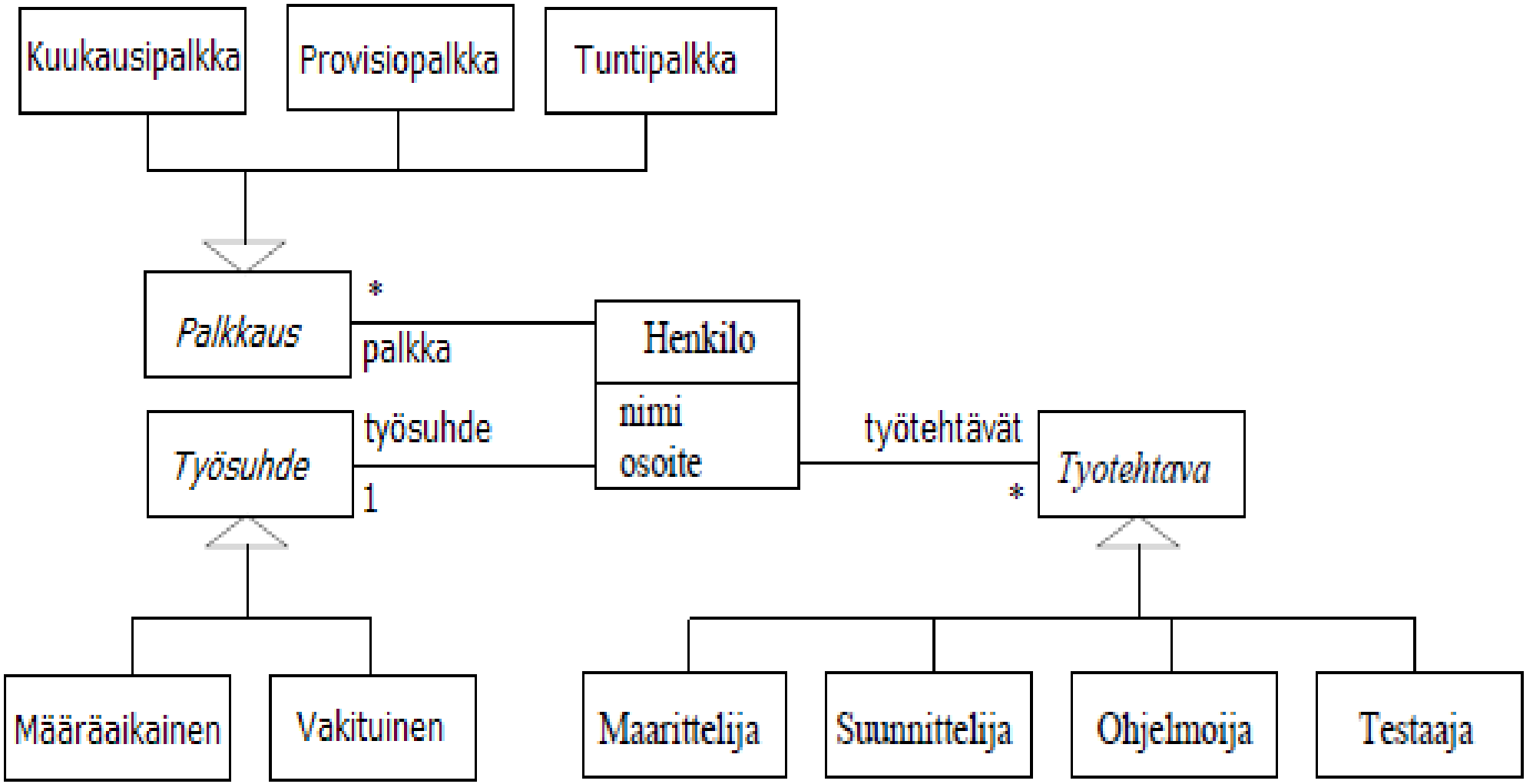
- Tyotehtava on nyt abstrakti luokka, sillä se on pelkkä käsite, jonka merkitys konkretisoituu vasta aliluokissa, esim. Ohjelmoija osaa koodata...
- Ratkaisun ”hinta”, on luokkien määrän kasvu
  - yhtä käsitettä, esim. ohjelmointia tekevää työntekijää kuvataan usealla oliolla: Henkilö-olio ja siihen liittyvä Ohjelmoija-olio
- Onko tämä ongelma?
  - Ei, päinvastoin! Single responsibility -periaate sanoo: luokalla tulee olla vain yksi selkeä vastuu
  - Kuljetaan siis oikeaan suuntaan: olioita on enemmän, mutta ne ovat yksinkertaisempia, enemmän yhteen asiaan keskittyviä
- Huom: noudatettiin oliosuunnittelun periaatetta *favor composition over inheritance* ja päädyttiin yksinkertaisemman vastuun (single responsibility) omaaviin luokkiin

Roolin kuvaaminen erillisenä luokkana

- Esimerkissä käytetään oliomallinnuksessa hyvin tunnettua periaatetta, jonka mukaan käsite (esim. henkilö) ja sen roolit (esim. työtehtävät) kannattaa mallintaa erillisinä luokkina
- Käsitettä vastaavasta luokasta on yhteys sen rooleja kuvaaviin luokkiin

- Jos tietty roolityyppi, esim. työtehtävä jakautuu useiksi toisistaan eriäviksi alikäsitteiksi, kannattaa nämä kuvata perinnän avulla
  - työtehtävä tarkentuu ohjelmoijaksi, suunnittelijaksi, jne...

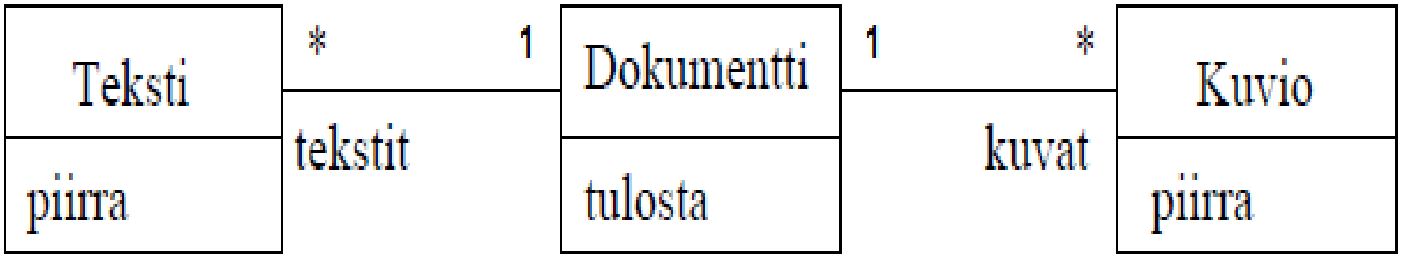
- Henkilöön voisi liittyä muitakin rooleja kun työntekijärooleja, esim.
  - työsuhteen laatua kuvaava rooli (vakinainen, määräaikainen)
  - palkkausta kuvaava rooli (tuntipalkka, kuukausipalkka, …)
  - ks. seuraava sivu



- Jos rooli on hyvin yksinkertainen, sen voi mallintaa normaalina attribuuttina
  - Työsuhteen laatu saattaisi olla parempi kuvata pelkän, esim. String-arvoisen attribuutin avulla
- Jos taas rooliin liittyy attribuutteja ja metodeja (esim. palkkaukseen liittyy palkan laskeminen), on se syytä kuvata omana luokkana

Monimutkainen esimerkki

- Dokumentti koostuu tekstielementistä ja kuvioista
- Kuvio voi olla piste, viiva, ympyrä tai joku näistä koostuva monimutkaisempi kuvio
- Yksinkertaistettu luokkakaavio, jossa ei ole vielä tarkennettu Kuvioa:



- Dokumentin operaatio tulosta() käy läpi kaikki tekstit ja kuvat ja pyytää niitä piirtämään itsensä

```
public class Dokumentti{
    private List<Teksti> tekstit;
    private List<Kuvio> kuvat;
    public void tulosta(){
      for ( Kuvio k : kuvat )  k.piirra();
      for ( Teksti t : tekstit )  t.piirra();
    }
}
```

Tarkennetaan kuvioa

- Kuvio voi siis olla

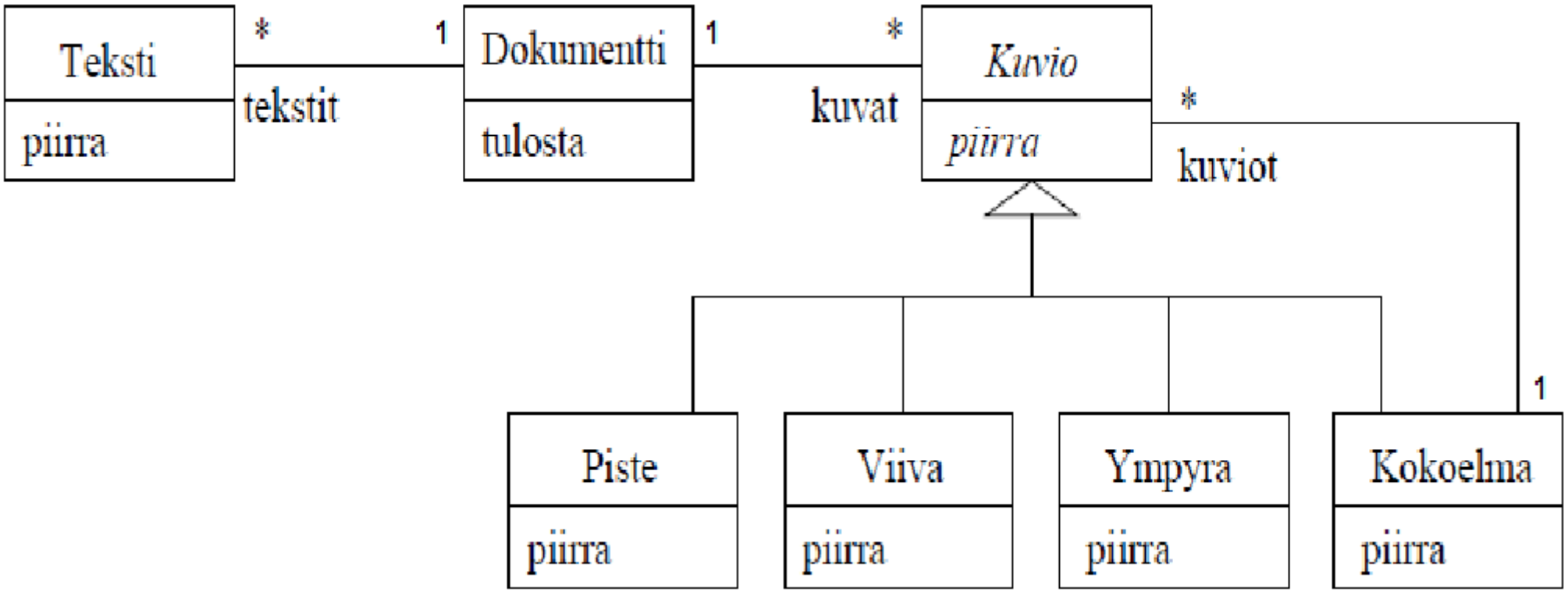
  piste, viiva tai ympyrä, tai
  - Edellisistä koostuva monimutkaisempi kuvio

- Kuvion määritelmä viittaa itseensä, eli määritelmä on rekursiivinen
  - Kuvio voi olla kooste yksinkertaisista kuvioista

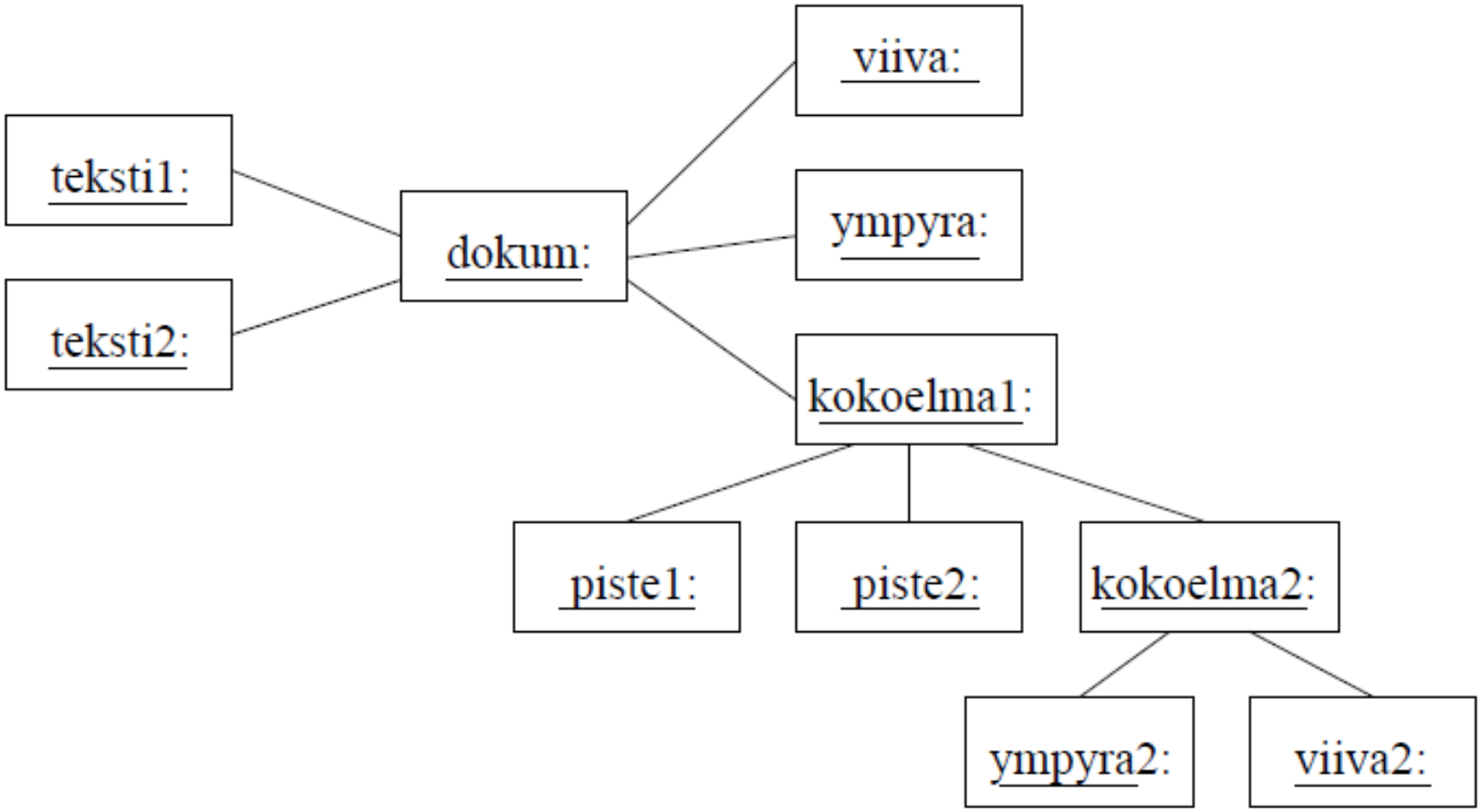
- Tarkennetaan määritelmää. Kuvio on, joko
  - piste,
  - viiva,
  - ympyrä tai
  - kokoelma kuvioita
- Luokkakaavio seuraavalla sivulla

Tarkentunut kuvio

- Koska Kuvio ei ole itsessään käyttökelpoinen luokka (siitä ei ole voi luoda olioita), on Kuviosta tehty abstrakti luokka, jolla on abstrakti metodi piirrä()
- Kuvion perivät konkreettiset luokat Piste, Viiva, Ympyrä ja Kokoelma, jotka toteuttavat piirrä()-metodin kukin omalla tavallaan
- Kokoelma sisältää joukon muita Kuvioita, eli kokoelma on koostesuhteessa sen sisältämiin Kuvio-olioihiin!
- Asia on hieman hämmentävä ja seuraavalla sivulla tilannetta selkeyttävä oliokaavio



- Oliokaaviossa kuvattu dokumentti, joka sisältää kaksi Teksti-olioa (*teksti1* ja *teksti2*) sekä kolme Kuvio-olioa
- Kuvio-olioista *viiva* ja *ympyra* ovat ”yksinkertaisia” kuvioita, kolmas dokumentin sisältävä kuvio on *kokoelma1*
- *kokoelma1* koostuu kolmesta kuviosta, joita ovat *piste1*, *piste2* ja *kokelma2*
- *kokoelma2* on siis koostekuvio, joka koostuu olioista *ympyrä2* ja *viiva2*



Polymorfismia...

- Kun dokumentti pyytää kuvioita piirtämään itsensä (koodi pari sivua aiemmin), polymorfismi pitää huolta, että kukin Kuvion aliluokka kutsuu toteuttamaansa piirrä()-metodia
  - Alla on luokkien Ympyrä ja Kokoelma piirrä()-metodin toteutus
- Ympyrä piirtää itsensä kutsumalla grafiikkakirjaston metodia drawCircle(...)
- Kokoelman piirrä()-metodin toteutus on mielenkiintoinen
  - Kokoelma koostuu joukosta Kuvio-olioita, joiden viitteet listassa kuviot
  - *Kokoelma piirtää itsensä käskemällä jokaisen sisältämänsä kuvion piirtämään itsensä*
  - Polymorfismin ansiosta jokainen kokoelman sisältämä Kuvio osaa kutsua todellisen luokkansa piirrä-metodia

```
public class Kokoelma extends Kuvio {
   private List<Kuvio> kuviot; // kokoelman kuviot
   public void piirra(){
      for ( Kuvio k : kuviot ) k.piirra();
   }
}
```

```
public class Ympyra extends Kuvio{
      public void piirra() {
         graphics.drawCircle( x, y, sade );
      }
}
```

Yleistetään mallia vielä hiukan

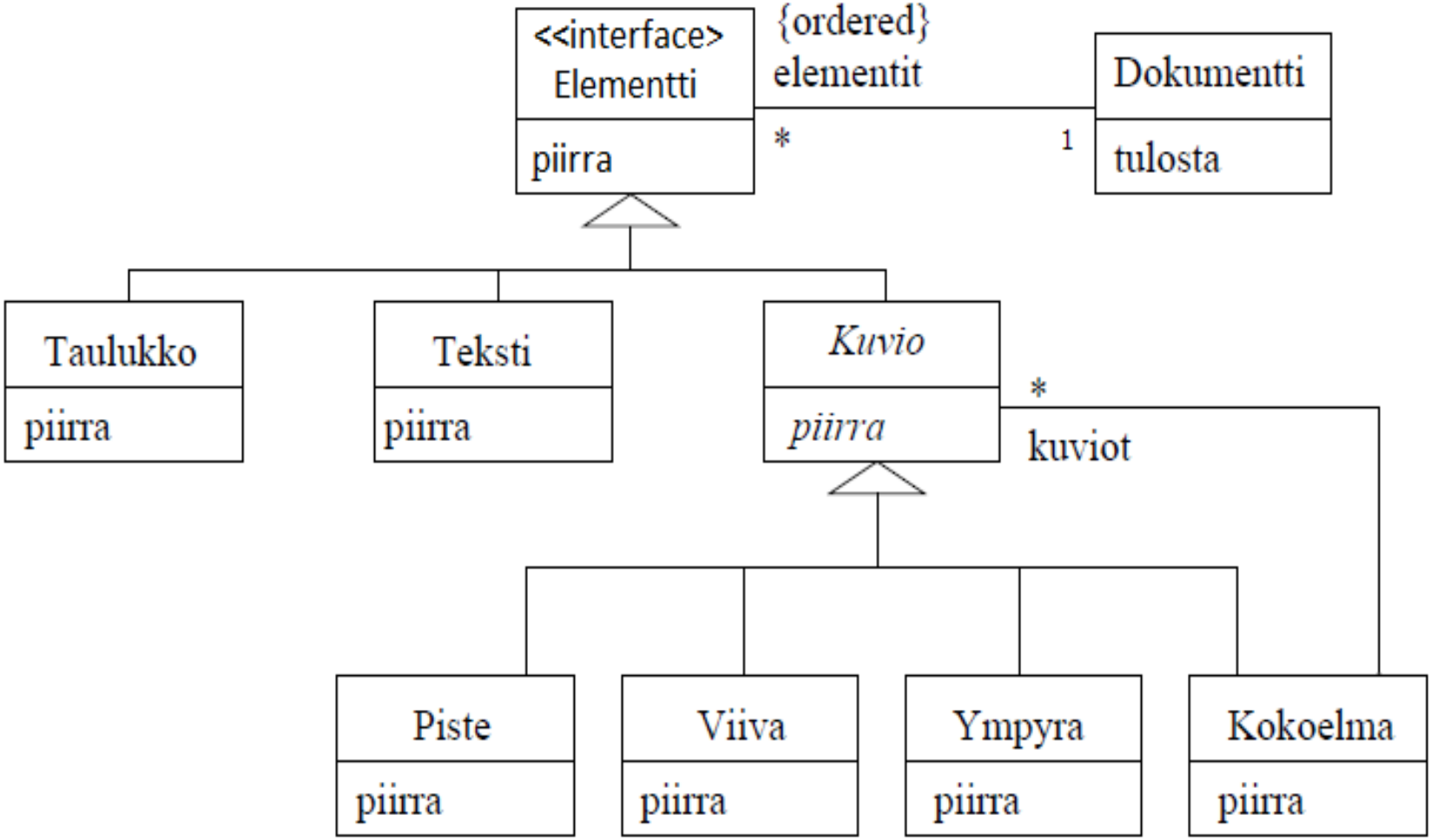
- Dokumentissa voisi olla muunkinlaisia rakenneosia kuin tekstiä ja kuvia, esim. taulukoita
- Mallia onkin järkevä yleistää, ja määritellä kaikille dokumentin rakenneosille yhteinen rajapinta Elementti, jolle on määritelty metodi piirrä jonka kaikki konreettiset elementit toteuttavat
- Dokumentti tuntee nyt joukon Elementti-rajapinnan toteuttavia oliota
- Dokumentin tulostaminen on helppoa, ote koodista:

```
public class Dokumentti {

    private List<Elementti> elementit;

    public void tulosta(){

        for ( Elementti elementti : elementit )

            elementti.piirra();

        }

    }
```

- Huom: noudetataan oliosuunnittelun periaatetta *program to interfaces not to concrete classes* eli ei olla riippuvaisia konkreettisista luokista
- Näin dokumenttiin on hyvin helppo lisätä myöhemmin uusia elementtityyppejä!

Dokumentin rakenteen kuvaava luokkamalli

- Jotta dokumentti olisi mielekäs, on eri elementit syytä liittää dokumenttiin tietyssä järjestyksessä
  - Järjestys voidaan ilmaista liittämällä yhteyteen määre ordered



- Tässä käytetty tapa mallintaa Kuvio perustuu yleisesti tunnettuun *composite pattern -periaatteeseen*