Ohjelmistotekniikan menetelmät

Luento 5, 24.11.

Olioiden pysyväistallennus

Olioiden talletus tietokantaan

- Käsittelimme viime viikolla olioiden pysyväistallennusta MongoDB-tietokantaan Morphia-kirjaston avulla
- Periaatteena se, että oliot talletetaan avain-arvo-pareista koostuvaksi *JSON-dokumenteiks*i, saman luokan oliot omaan *kokoelmaansa*

```
{
    nimi: ”Arto Vihavainen”,
    opnro: ”012345687”,
    puh: ”040-1234567”,
    osoite: {
                katu: ”Mannerheimintie 10 A 1”,
                postinumero: ”00100”
            },
    suoritukset: [ ”ohpe”, ”ohja”, ”tikape” ],

}
```

- Morphia edellyttää, että tallennettava luokka *annotoidaan* sopivasti

```
@Entity
public class Opiskelija {
   @Id
   private ObjectId id;

   // ...
```

Tietokantaoperaatiot

- Tietokannan käsittelyä varten tarvitaan *Datastore*-tyyppinen olio

```
Datastore store = new Morphia().createDatastore( … );
```

- Olion tallentaminen kantaan:

```
Opiskelija arto = new Opiskelija(”Arto”, ”012345678”, 2001, 401);
arto.lisaaSuorius(”ohpe”);
arto.lisaaSuorius(”ohja”);
store.save(arto);
```

- Tietokantakyselyt *Query*<*Luokka*>-olion avulla

```
Query<Opiskelija> query = store.createQuery(Opiskelija.class);
```

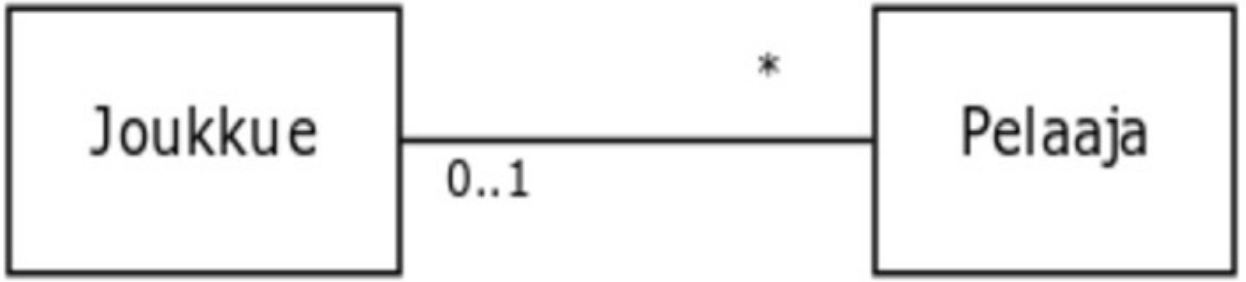
  - Jokaista erilaista kyselyä varten tarvitaan oma kyselyolio

- Hakuehdot liitetään kyselyyn ja pyydetään kyselyltä tuloksena olevat oliot tai olio

```
Opiskelija arto = query.field("nimi").equal("Arto Vihavainen").get();

List<Opiskelija> uudet = query.field("aloitusvuosi").greaterThan(2010).asList();

query.and(
      query.criteria("aloitusvuosi").lessThan(1999),
      query.criteria("opintopisteita").lessThan(200)
)
List<Opiskelija> laiskat = query.asList();
```

Dokumenttien väliset yhteydet

- Joukkueeseen kuuluu useita pelaajia ja pelaaja on vain yhdessä joukkueessa kerrallaan eli *yhden suhde moneen*



- Useita vaihtoehtoisia tapoja yhteyden toteuttamiseen
  - pelaajien *sisällytys* joukkueisiin
  - pelaajat omana kokoelmana, joukkueista *viite* (eli käytännössä Mongon generoima id) pelaajiin
- Kumpi ratkaisu parempi: riippuu käyttötarkoituksesta!
- Jos pelaajiin on vain viite joukkueista, kyselyä *etsi joukkue, jonka pelaajien maalimäärä suurin* ei pystytä tekemään tietokantatasolla
  - MongoDB ei tule liitoskyselyjä
  - Operaatio suoritettava kannasta ladatuilla olioilla

Dokumenttien väliset yhteydet: olioiden sisällytys

- Joukkueen ja pelaajan koodi

```
@Entity
public class Joukkue {
      @Id
      ObjectId id;
      private String nimi;
      @Embedded
       private ArrayList<Pelaaja> pelaajat;
}
```

```
public class Pelaaja {
     String nimi;
      int maaleja;
}
```

- Pelaajaoliot on tallennettu joukkueiden sisälle

```
{
       _id: ObjectId("5640ee842c2066422682baa6"),
      className: "com.mycompany.morphia.Joukkue",
      nimi: "HJK",
      pelaajat: [
            {  nimi: "Antti", maaleja: 7 },
            {  nimi: "Arto", maaleja: 3 }
      ]

}
```

- Pelaajat voidaan hakea kannasta ainoastaan hakemalla kaikki joukkueet

Dokumenttien väliset yhteydet: pelaajatunnisteet joukkueeseen

- Myös pelaaja merkitään kantaan tallennettavaksi

```
@Entity
public class Joukkue {
      @Id
      ObjectId id;
      private String nimi;
      @Reference
      private ArrayList<Pelaaja> pelaajat;
}
```

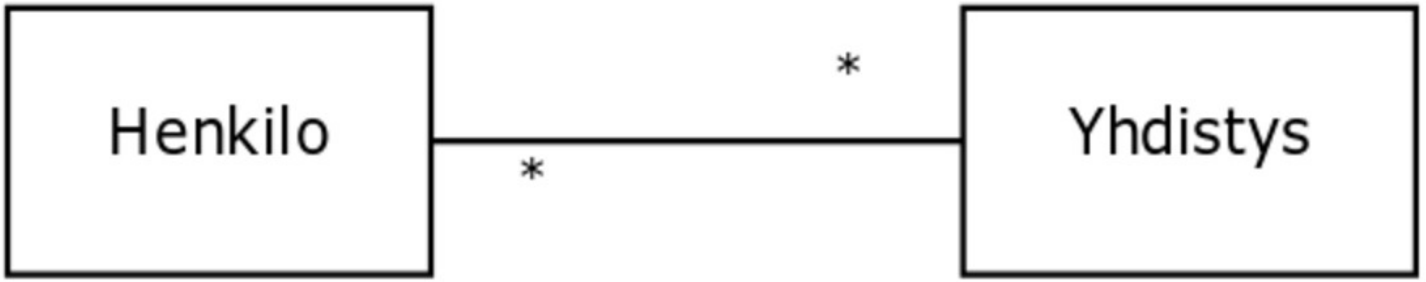
```
@Entity
public class Pelaaja {
      @Id
      ObjectId id;
      private String nimi;
      private int maaleja;
}
```

- Pelaajat tallentuvat omaan kokoelmaan ja joukkueet ainoastaan viittaavat pelaajiin

```
{
   _id: ObjectId("5641d80580905f5706e0220d"),
   className: "com.mycompany.morphia.Joukkue",
   nimi: "IFK",
   pelaajat: [
        {   $ref: "Pelaaja",  $id: ObjectId("5641d80580905f5706e0220e") },
        {   $ref: "Pelaaja",  $id: ObjectId("5641d80580905f5706e0220f")  }
   ]
}
```

Monen suhde moneen -yhteydet

- Toisin kuin relaatiotietokannant, dokumenttitietokannat pystyvät vaivattomasti esittämään myös *monesta moneen* -suhteita
- Oletetaan että meillä olisi henkilöitä ja yhdistyksiä, joiden välillä olisi monesta moneen -yhteys, eli henkilö voi kuulua moneen yhdistykseen ja yhdistyksessä voi olla monta henkilöä jäsenenä



- Tilanne hoidetaan MongoDB:llä seuraavasti
  - henkilödokumentti sisältää listan niiden yhdistysten tunnuksista, joihin henkilö kuuluu
  - yhdistysdokumentti sisältää listan niistä henkilöistä, jotka ovat yhdistyksen jäsenenä

- Henkilö-kokoelma

```
{
        _id: ObjectId(123),
        nimi: "Arto",
        yhdistykset: [{ $ref: ”Yhdistys”, $id:ObjectId(201)}, { $ref: ”Yhdistys”,  $id:ObjectId(202) }]
},

{
        _id: ObjectId(124),
        nimi: "Pekka",
        yhdistykset: [{ $ref:”Yhdistys”,  $id:ObjectId(202)}]

}
```

- Yhdistys-kokoelma

```
{
        _id: ObjectId(201),
        nimi: "Kumpulan kyläyhdistys",
        jasenet: [{ $ref:”Henkilo”, $id:ObjectId(123)}]

},

{
        _id: ObjectId(202),
        nimi: "TKTL-alumni",
        jasenet: [{ $ref:”Henkilo”, $id:ObjectId(123)}, {$ref:”Henkilo”, $id:ObjectId(124})]
}
```

Monen suhde moneen

- Henkilön ja yhdistyksen koodi ei sisällä mitään yllättävää:

```
@Entity
public class Henkilo {
    @Id
    private ObjectId id;
    private String nimi;
    @Reference
    List<Yhdistys> yhdistykset;
}
```

```
@Entity
public class Yhdistys {
    @Id
    private ObjectId id;
    private String nimi;
    @Reference
    List<Yhdistys> jasenet;
}
```

- Henkilöiden ja yhdistysten luominen ja niihin kohdistettavat kyselyt toimivat myös täysin samalla tavalla kuin aiemmissa esimerkeissä
- Jos yhteyteen liittyisi tietoja, esim. jäsenyyden alkamisaika. Jäsenmaksun suuruus ym. kannattaisi yhteys mallintaa omana luokkanaan:

- Tämä taas on analoginen sille, miten relaatiotietokannat toteuttavat monesta moneen -yhteydet *liitostaulujen* avulla

MongoDB konsoli

- Olemme tehneet kaikki tietokantaoperaatiomme Morphian kautta. Miten MongoDB:tä käytetään ilman apukirjastoja?
- Kaikissa Mongo-komennoissa parametri on JSON-muotoinen dokumentti
- Uusien dokumenttien luominen tapahtuu seuraavasti:

```
db.student.insert({
      ”nimi”: ”arto”,
      ”opintopisteita”: 100,
      ”osoite”: { ”katu”: ”mannerheimintie”,
                  ”kaupunki”: ”helsinki” }
});
```

- Esim hae 2014 jälkeen alottaneet opiskelijat, joilla alle 10 opintopistettä

```
db.student.find({ "$and" : [
                              { "aloitusvuosi" : { "$gt" : 2014}} ,
                              { "opintopisteita" : { "$lt" : 100}}
                            ]
                });
```

- Kyselyn JSON-muodon saa selville kutsumalla Morphian kyselylle *toString*
- Mongon käyttäminen ”natiivisti” kyselyjen tekemiseen on hieman ikävää. Tulemme kuitenkin laskareissa kokeilemaan myös JSON-muotoisia kyselyjä

Yleistys-erikoistussuhde eli perintä

Yleistys-erikoistus ja periminen

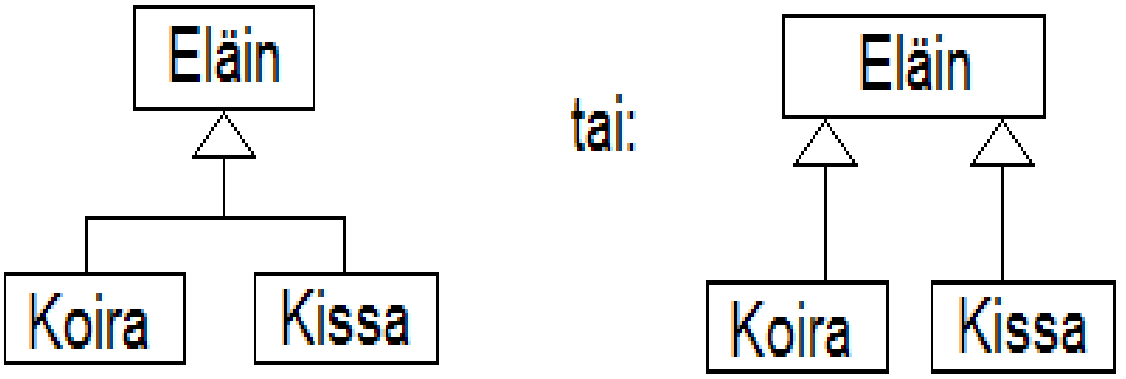
- Tähän mennessä tekemissämme luokkakaaviossa kaksi luokkaa ovat voineet liittyä toisiinsa muutamalla tapaa
- *Yhteys ja kompositio* liittyvät tilanteeseen, missä luokkien olioilla on rakenteellinen (= jollain lailla pysyvä) suhde, esim.:
  - Henkilö *omistaa* Auton (*yhteys*: normaali viiva)
  - Huoneet *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..
  - Luennolta 2: järjestelmällä eri oikeuksin varustettuja käyttäjiä User, Editor, Moderator. Editori perii Userin käyttötapaukset ja Moderator Editorin
  - Sama valkoinen kolmiosymbooli oli käytössä myös käyttötapausten yleistyksen yhteydessä

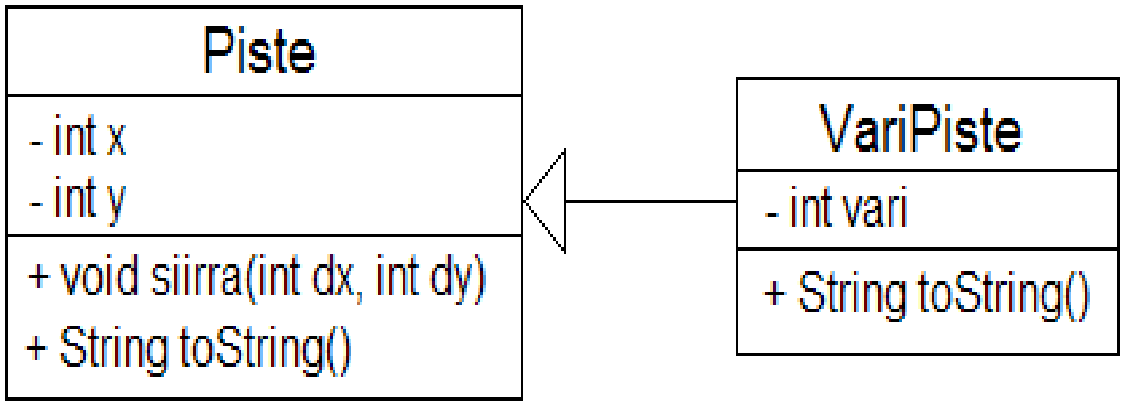
Periytyminen

- Luokkien välinen yleistys-erikoistussuhde eli yli- ja aliluokat toteutetaan ohjelmointikielissä siten, että aliluokka perii yliluokan
- Tuttu esimerkki Ohjelmoinnin jatkokurssilta, konstruktoreja 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; }
}
```

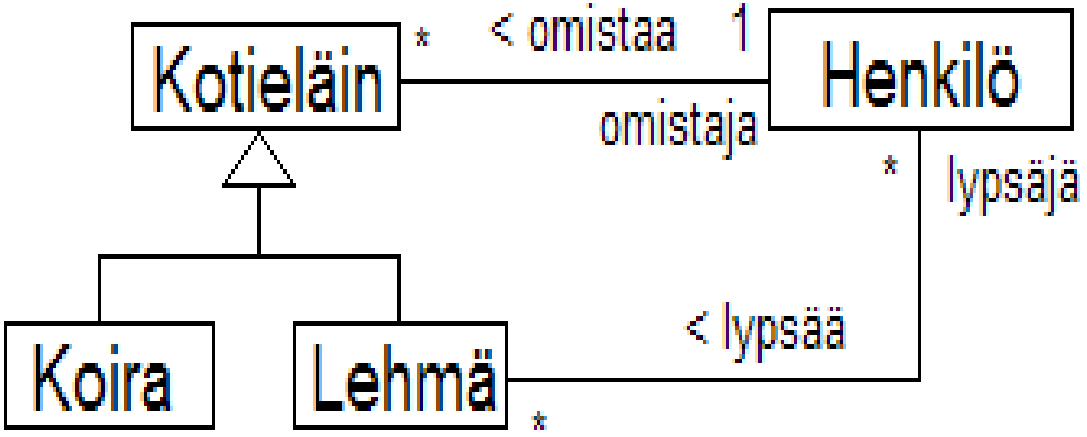
Periytyminen ja luokkakaavio

- Yliluokan Piste attribuutit x ja y sekä metodi siirra() siis periytyvät aliluokkaan VariPiste
  - Periytyviä 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 VariPisteella on:
  - Attribuutit 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?

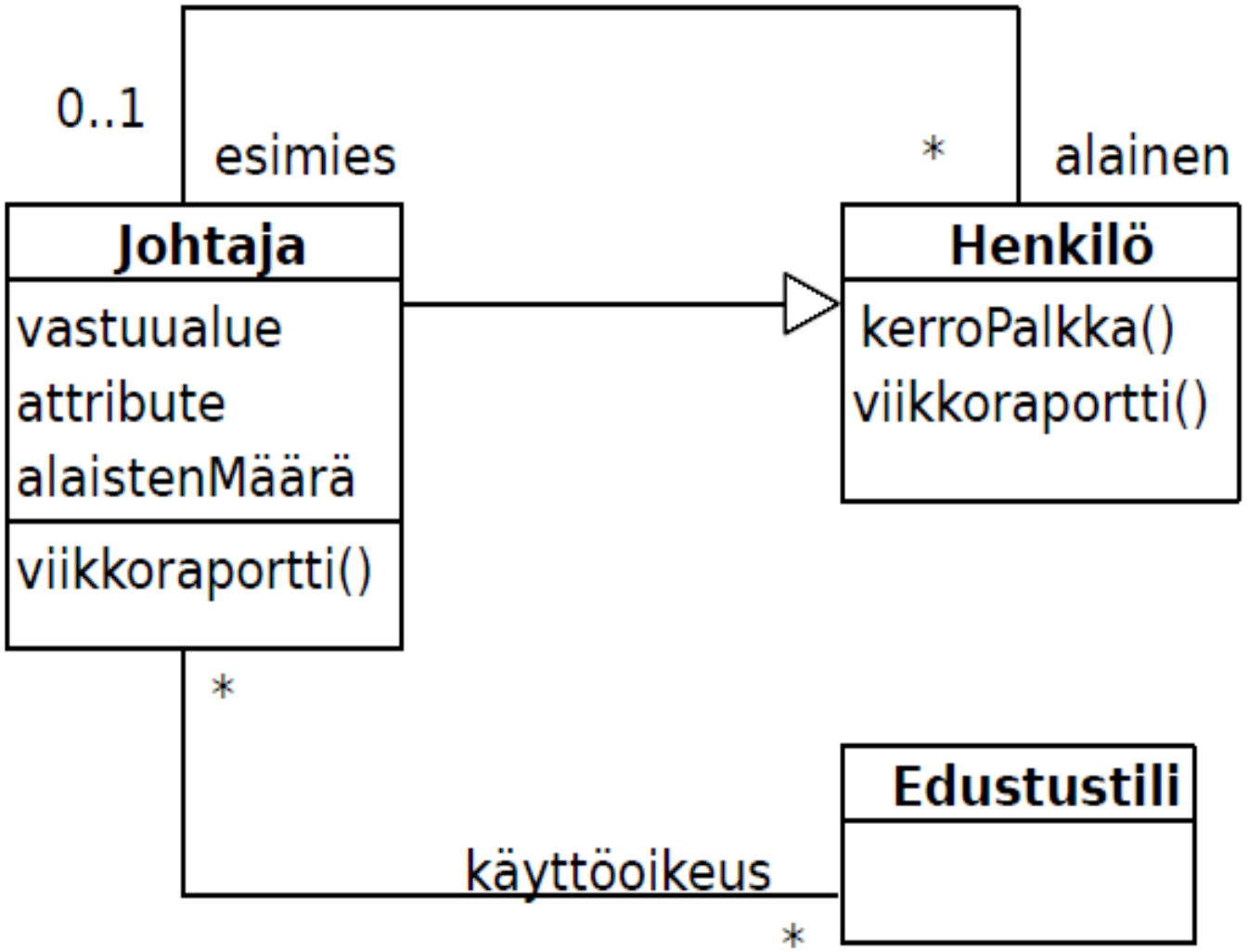
- Luokat Koira ja Lehmä ovat molemmat luokan Kotieläin aliluokkia
- Jokaisella kotieläimellä on omistajana joku Henkilö-olio
- Koska omistaja liittyy kaikkiin kotieläimiin, merkitään yhteys Kotieläin- ja Henkilö-luokkien välille
  - **Yhteydet periytyvät aina aliluokille**, eli Koira-olioilla ja Lehmä-olioilla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

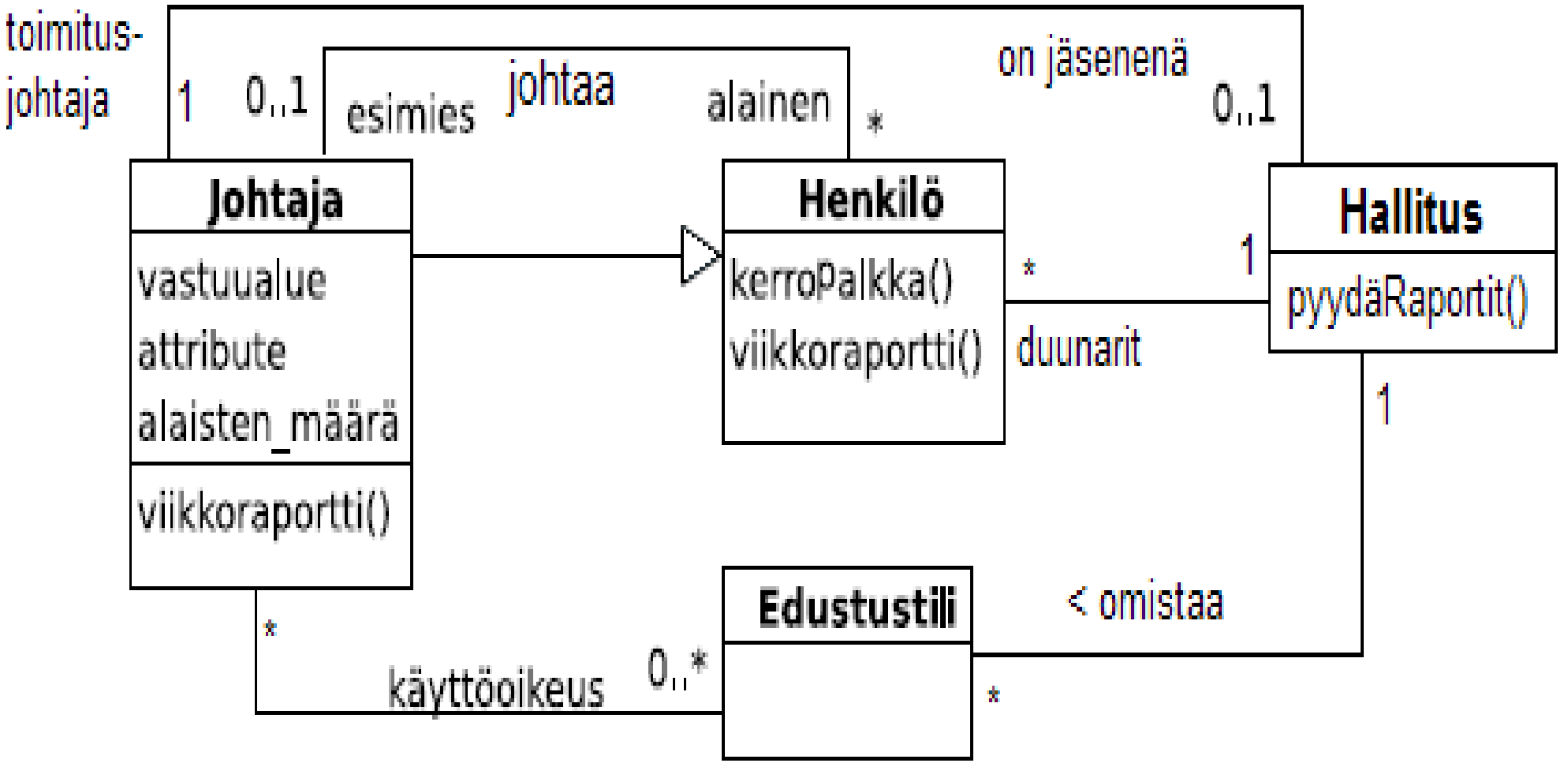
  Laadi yhteenveto alaisten viikkoraporteista

  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!*)

Lisää perinnästä

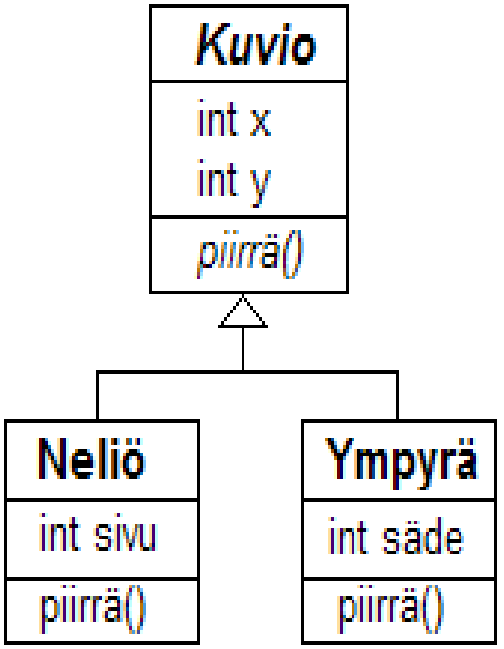
abstraktit luokat ja rajapinnat

Abstraktit luokat

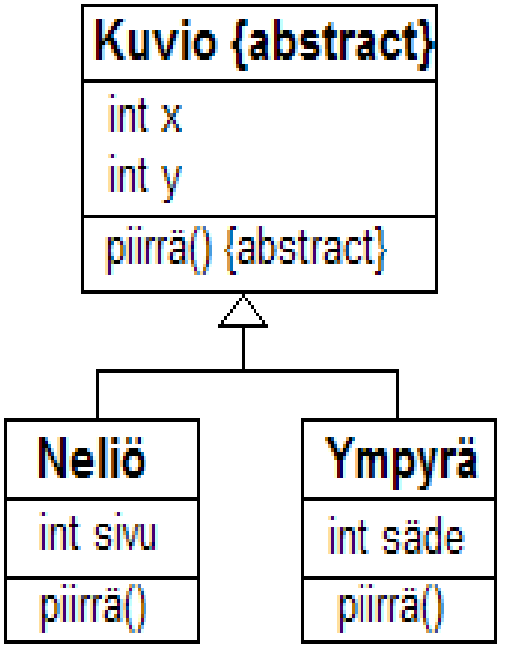
- Yliluokalla Kuvio on sijainti, joka ilmaistaan x- ja y-kordinaatteina sekä metodi piirrä()
- Kuvion aliluokkia ovat Neliö ja Ympyrä
  - Neliöllä on sivun pituus ja Ympyrällä säde
- Kuvio on nyt pelkkä abstrakti käsite, Neliö ja Ympyrä ovat konkreettisia kuvioita jotka voidaan piirtää ruudulle kutsumalla 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);
    }
```

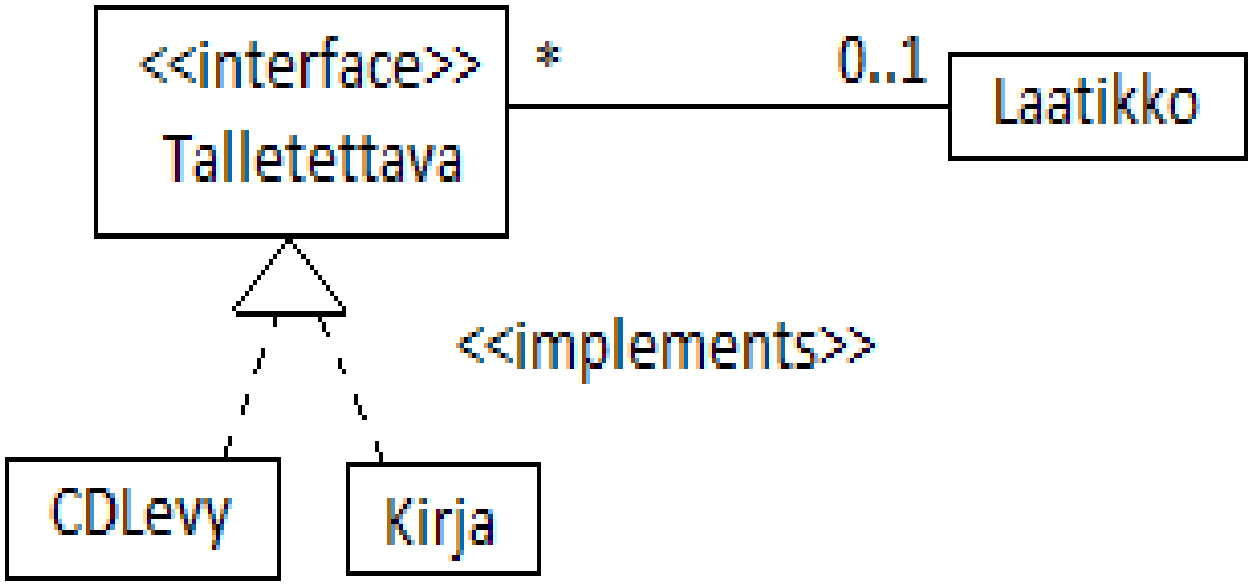
Rajapinta

- Javan *rajapinta* (interface) on ikäänkuin abstrakti luokka, joka ei sisällä attribuutteja ja jossa (useimmiten) kaikki metodit ovat abstrakteja
- Rajapintaluokka siis (yleensä) listaa ainoastaan joukon metodien nimiä
  - Java 8 on tuonut tähän sen poikkeuksen, että rajapintojen metodeila voi olla *oletustoteutuksia*
- 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 metodit (paitsi ne joilla on oletustoteutus)
- Tai toisinpäin ajateltuna, **rajapinta on sopimus**
    - **toteuttaja lupaa toteuttaa ainakin rajapinnan määrittelemät metodit**

Tuttu esimerkki rajapintaluokan käytöstä

- Mallinnetaan Ohjelmoinnin jatkokurssin toisen viikon tehtävä Tavaroita ja Laatikoita
  - http://www.cs.helsinki.fi/group/java/s15-materiaali/viikko9/#136tavaroita_ja_laatikoita
  - 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

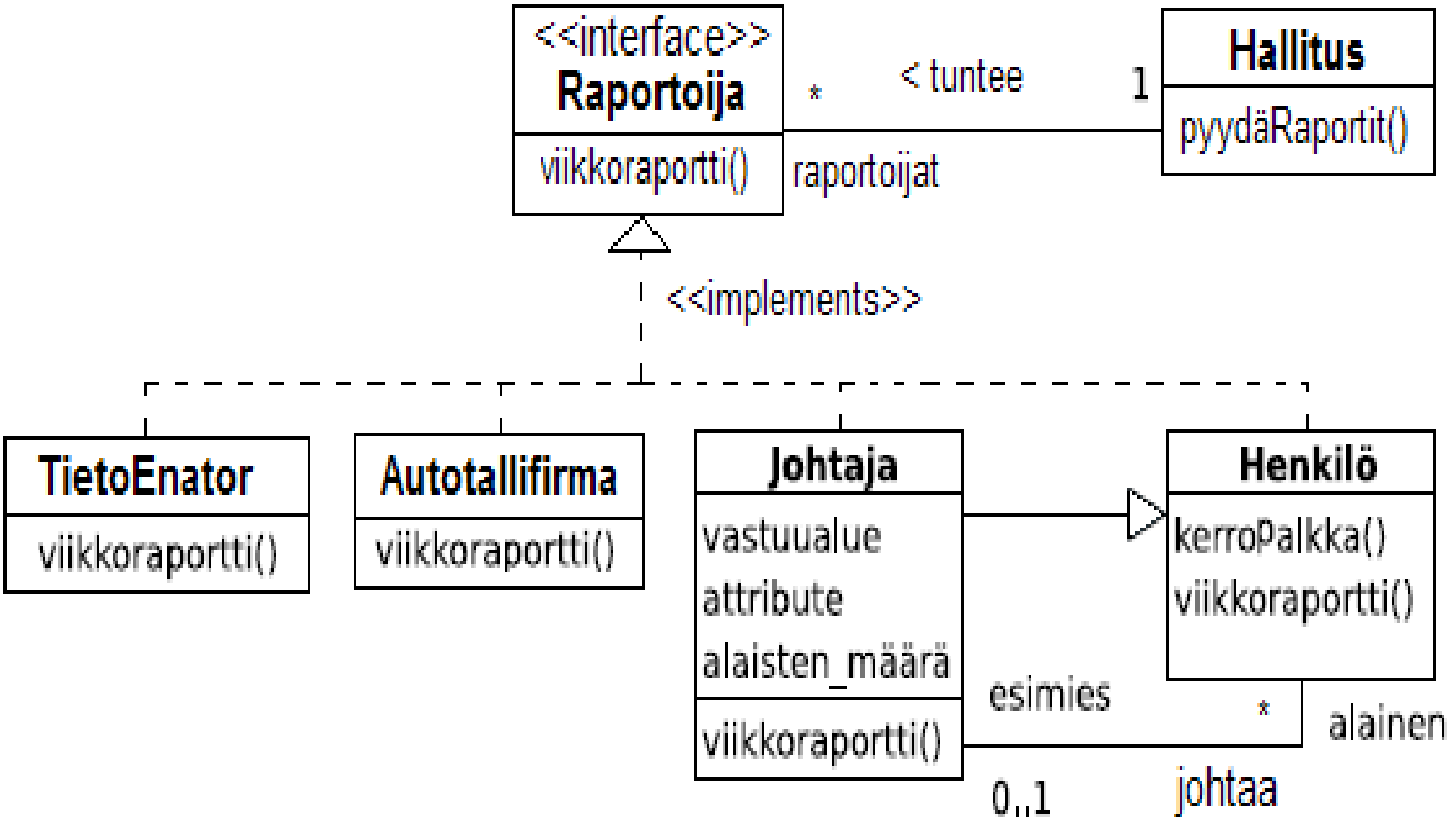
- Palataan muutaman sivun takaiseen yritysesimerkkiin
- Tilanne on nyt se, että yritys on ulkoistanut osan toiminnoistaan
- Hallitus on edelleen kiinnostunut viikkoraporteista
  - Hallitusta ei kuitenkaan kiinnosta se, tuleeko viikkoraportti omalta henkilöstöltä vai alihankkijalta
- Muuttuneessa tilanteessa 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 *rajapinta* 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()  }
   }
```

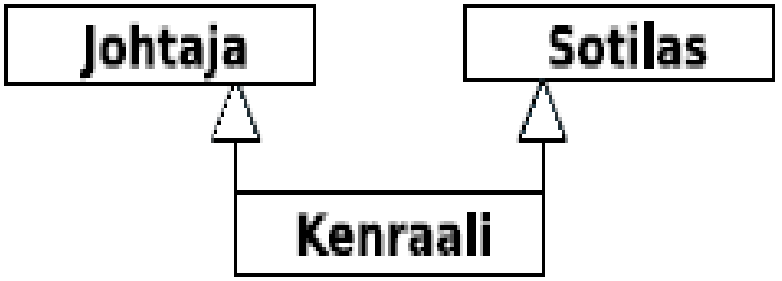


Lisää perinnästä

moniperintä

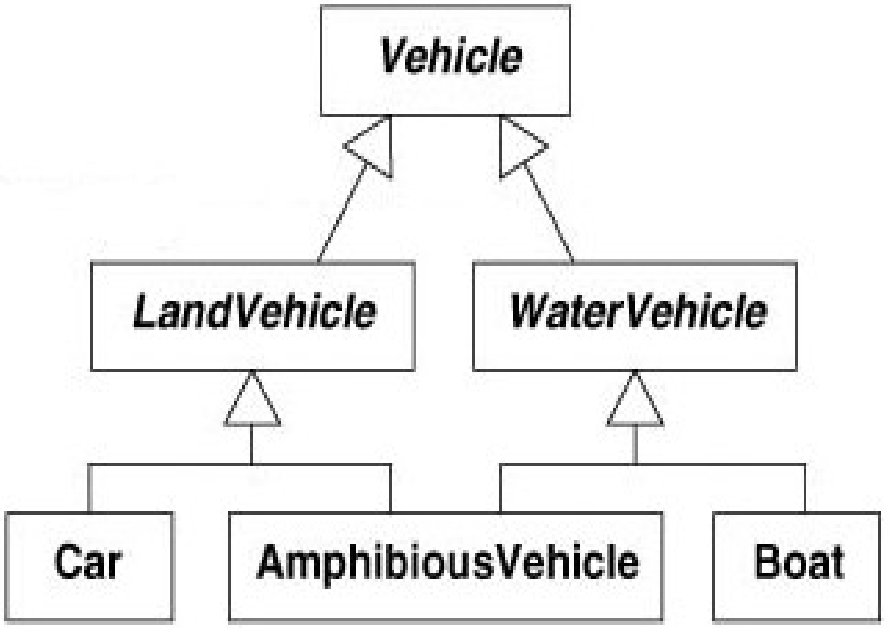
Moniperintä

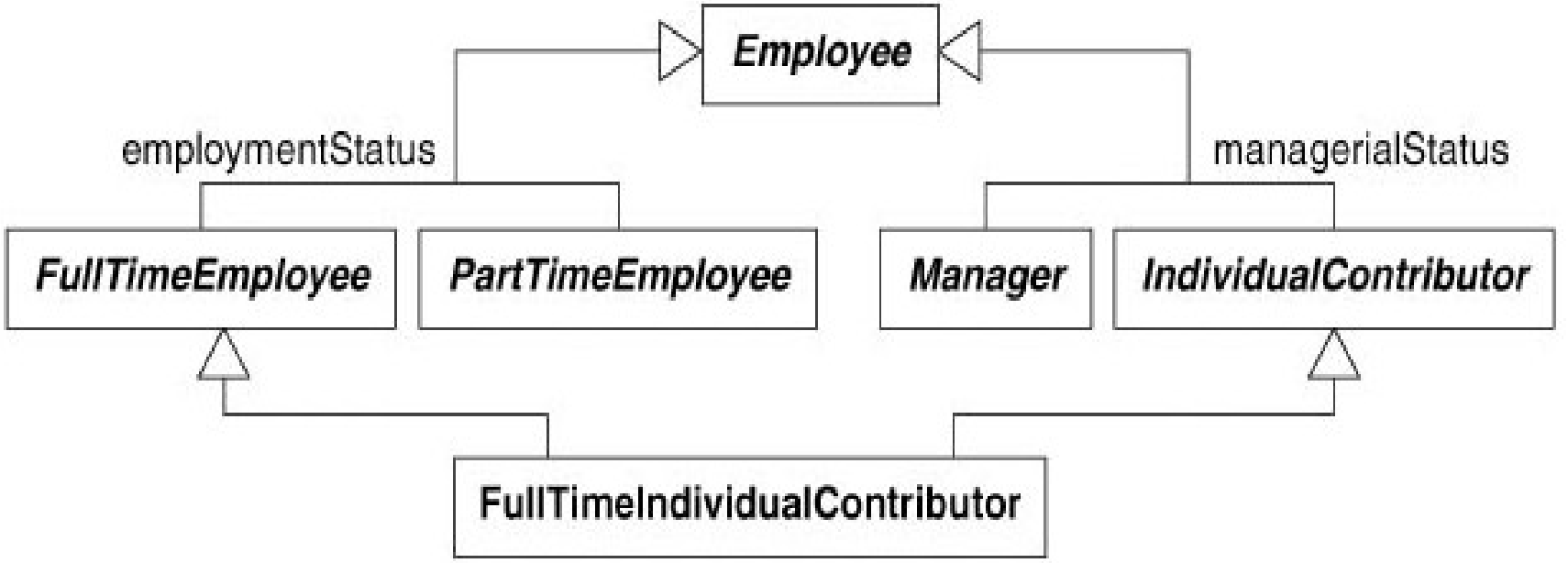
- Joskus tulee esiin tilanteita, joissa yhdellä luokalla voisi kuvitella olevan useita yliluokkia
- Esim. kenraalilla on sekä sotilaan, että johtajan ominaisuudet
  - Voitaisiin tehdä luokat Johtaja ja Sotilas ja periä Kenraali näistä
- Kyseessä *moniperintä* (engl. multiple inheritance)



- 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





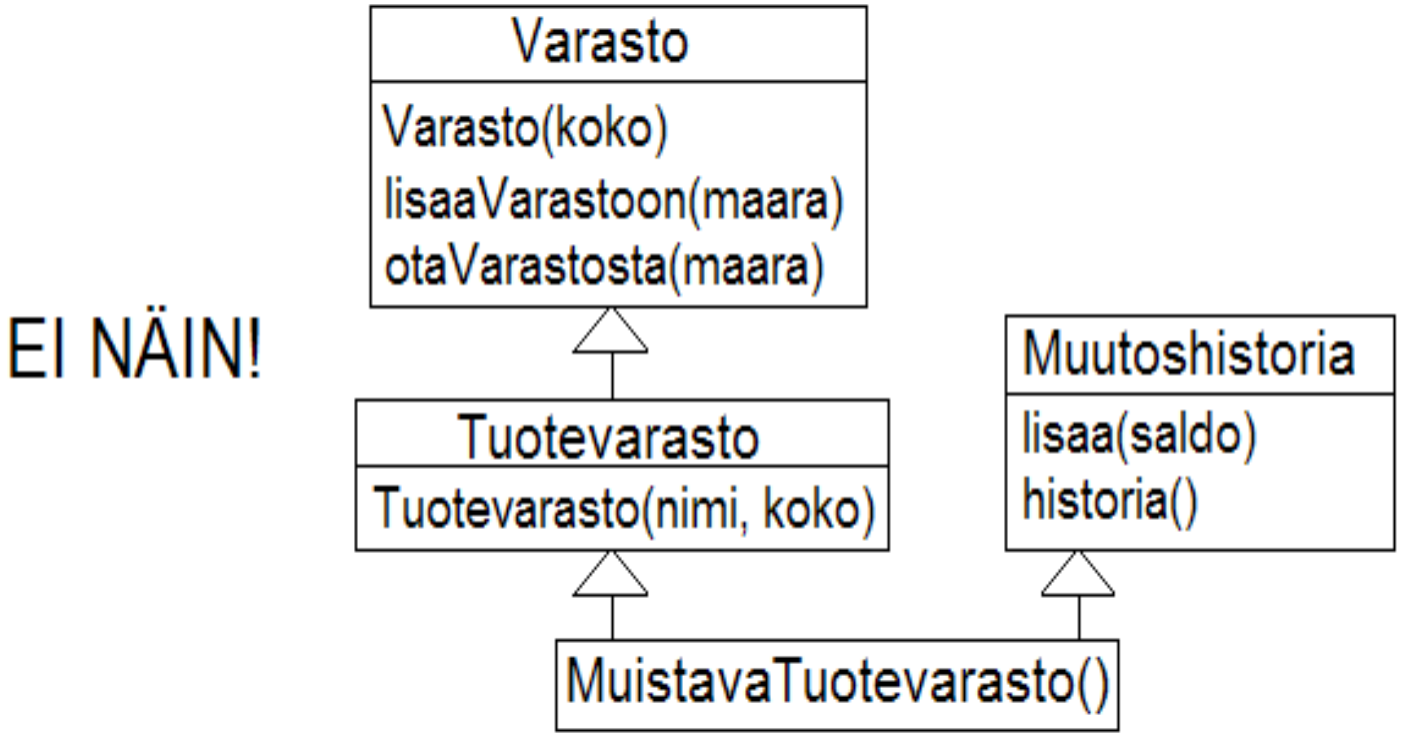
Moniperinnän ongelmat

- 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-mekanismi, joka mahdollistaa ”hyvinkäyttäytyvän” moniperintää vastavan mekanismin
- Monissa tilanteissa 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
  - Javan rajapinnan toimivat joissain tapauksessa moniperinnän korvikkeena varsinkin kun rajapinnat tukevat Java 8:n ilmestymisen jälkeen metodien oletusarvoisia toteutuksia
- Ohjelmoinnin jatkokurssin viikon 4 (tai viikon 10) laskareissa, ks http://www.cs.helsinki.fi/group/java/s15-materiaali/viikko11/#155varastointia

  toteutetaan MuistavaTuotevarasto, joka on luokka johon lisätään toiminnallisuutta perimisen sijaan *liittämällä* siihen toinen olio

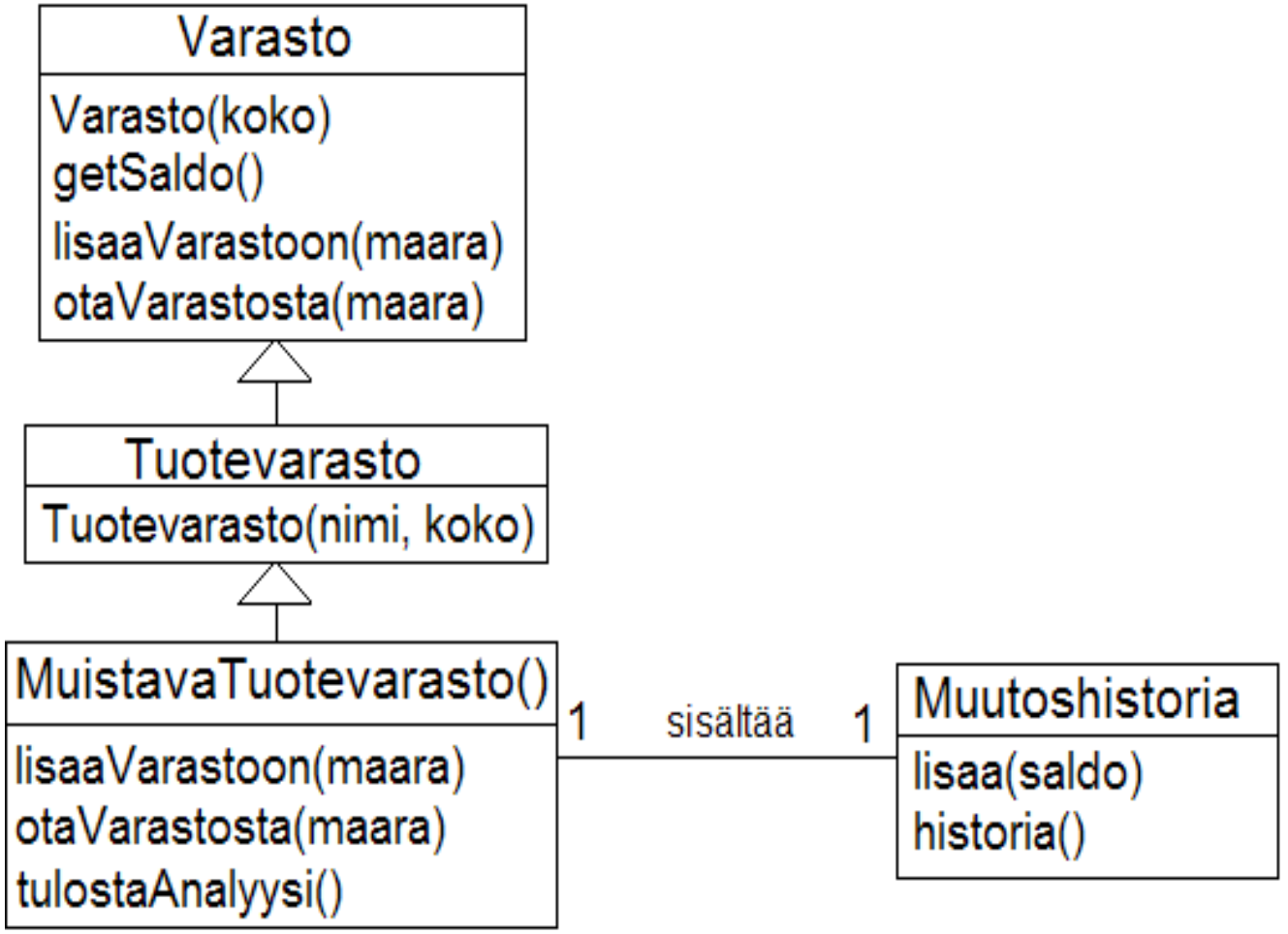
Muistava tuotevarasto

- 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:



Muistava tuotevarasto

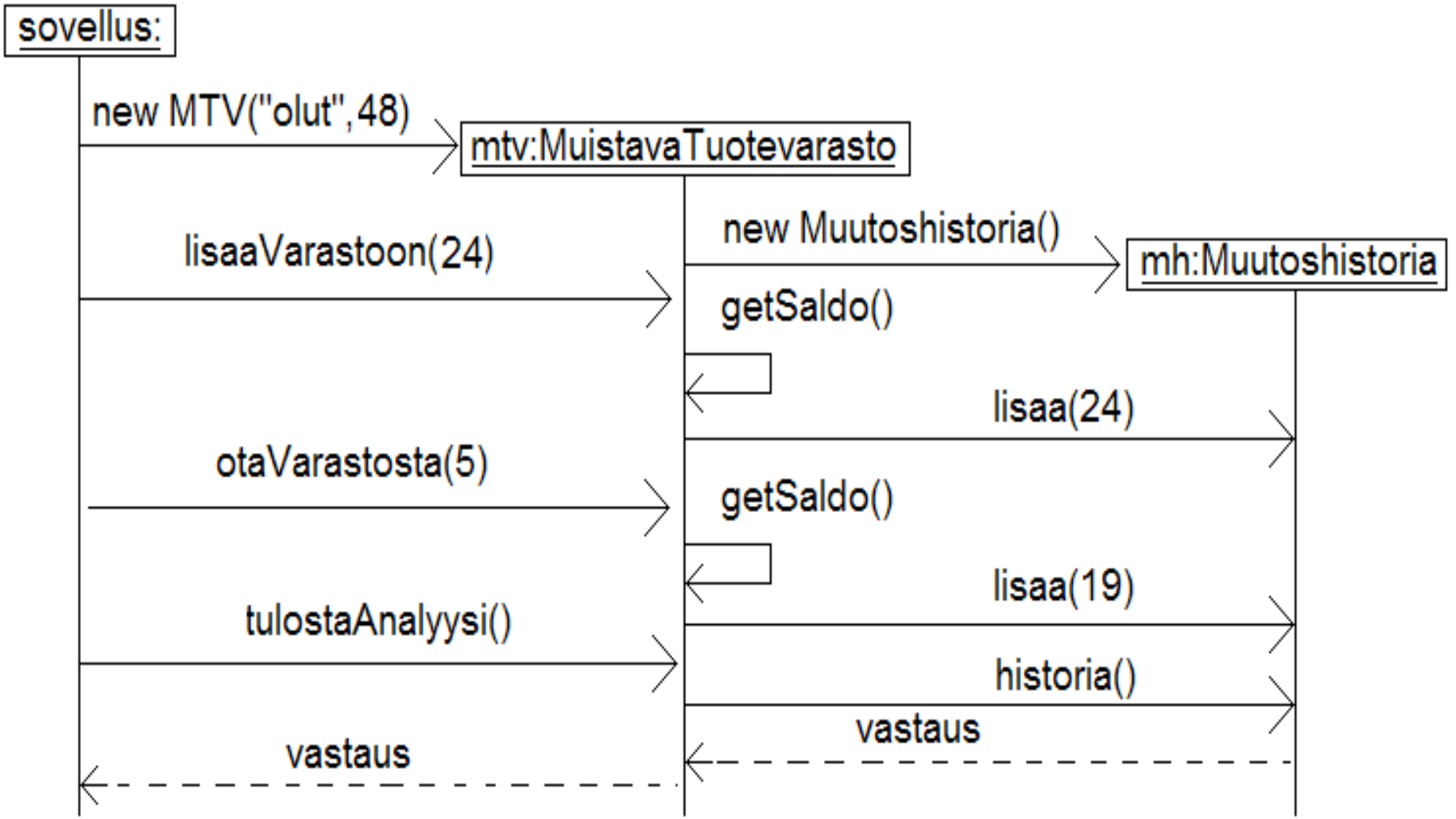
- Javassa ei moniperintää ole, ja vaikka olisikin, on parempi liittää ”muistamistoiminto” muistavaan tuotevarastoon erillisenä oliona:



- 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
  - Kaaviosta on jätetty pois seuraavalla sivulla koodissa näkyvät super-kutsut
- 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
    }
}
```

Oliosuunnittelun periaatteita

Miten olioita tulisi käyttää?

- Ohjelmointikielet tarjoavat paljon erilaisia mekanismeja, mm. ohjelmoinnin jatkokurssillakin edellisinä viikoilla tarkastelun alla olleet perinnän ja rajapinnat
- Aloittelevalle ohjelmoijalle ei kuitenkaan ole ollenkaan selvää miten kielen mekanismeja olisi järkevä käyttää, eli minkälaista on ”hyvä” ja toisaalta ”huono” koodi
- Lähtökohtana on tietysti se, että koodi toteuttaa ohjelmalle asetetut vaatimukset, eli
  - ohjelmalla on ne ominaisuudet, joita asiakas haluaa
  - ohjelma on riittävässä määrin virheetön
  - ohjelma on riittävän tehokas asiakkaan tarpeisiin
- Ohjelman *sisäinen laatu*, eli se mitä suunnittulratkaisuja koodia kirjoitettaessa on käytetty on myös tärkeää
- Jos ohjelma on sisäiseltä laadultaan huonoa, ohjelman ylläpito- ja laajennuskustannukset voivat nousta niin suuriksi että ohjelmisto muuttuu jossain vaiheessa käyttökelvottomaksi

Oliosuunnittelun periaatteita

- Aikojen saatossa on huomattu, että sisäiseltä laadultaan hyvissä ohjelmissa on tiettyjä samankaltaisia piirteitä, ja näitä tutkimalla on päädytty joukkoon *hyvän oliosuunnittelun periaatteita*
- Periaatteita on useita, tarkastellaan tänään neljää
  - Single responsibility
  - Favour composition over inheritance
  - Program to an interface, not to an Implementation
  - Riippuvuuksien minimointi
- **Single responsibility**
  - Tarkoittaa karkeasti ottaen, että **oliolla tulee olla vain yksi vastuu** eli yksi asiakokonaisuus, mihin liittyvästä toiminnasta luokka itse huolehtii
  - Äskeinen esimerkkimme *MuuttuvaTuotevarasto* toteuttaa periaatetta, sillä sen vastuulla on vain varaston nykyisen tilanteen ylläpito
  - Se *delegoi* vastuun aikaisempien varastosaldojen muistamisesta Muutoshistoria-oliolle

Oliosuunnittelun periaatteita

- **Favour composition over inheritance**
  - eli **suosi yhteistoiminnassa toimivia olioita perinnän sijaan**
  - Perinnällä on paikkansa, mutta sitä tulee käyttää harkiten
  - MuistavaTuotevarasto käyttää myös perintää järkevästi
    - Jos olisi moniperitty Tuotevarasto ja Muutoshistoria, olisi muodostettu luokka, joka rikkoo single responsibility – eli yhden vastuun periaatteen
- **Program to an interface, not to an Implementation**
  - 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 mahdollistaa myös rajapinnan takana olevan luokan korvaamisen kokonaan uudella luokalla
- **Riippuvuuksien minimointi**
  - Älä tee spagettikoodia, jossa kaikki oliot tuntevat toisensa
  - Pyri elimonoimaan riippuvuudet siten, että luokat tuntevat mahdollisimman vähän muita luokkia, ja mielellään nekin vain rajapintojen kautta

Esimerkki huonosta koodista

- Alla ote viime viikon laskareiden MongoDB:n kanssa keskustelleesta opiskelijoiden ilmottautumisia käsitelleestä sovelluksesta

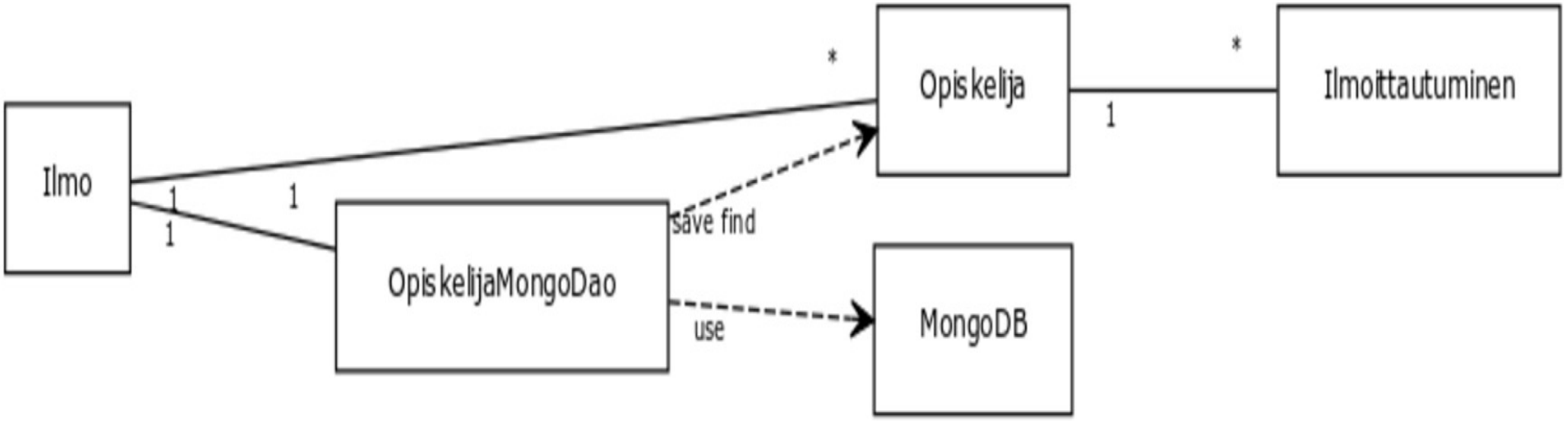
```
public class Ilmo {

    private Datastore store;
    private Scanner lukija;

    public Ilmo() {
        Morphia morphia = new Morphia();
        MongoClient mc = new MongoClient("80.69.172.211:27017");
        store = morphia.createDatastore(mc, "otm4");
    }

    public void suorita() {
        // …
       System.out.println(”minkä kurssin ilmottautumiset haluat nähdä: ”);
       String kurssi = lukija.nextLine();

        Query<Opiskelija> query = store.createQuery(Opiskelija.class);
       List<Opiskelija> ilmot = query.field("ilmoittautumiset.kurssi").equal(kurssi).asList();

        for ( Opiskelija o: ilmot) {
           System.out.prinltn(o.getNimi());
        }
    }
}
```

Esimerkki huonosta koodista

- Mitä ongelmia koodissa on?
- Koodi keskittyy yhtä aikaa useampaan asiaan
  - käyttäjän kanssa käytävä interaktio
  - olioiden talletus ja hakeminen tietokannasta
  - ”sovelluslogiikka”, eli esim. opiskelijoiden ilmottaminen kursseille, osakuntien jäsenten tietojen hallinta, ...
- Sovellus siis rikkoon *single responsibility* -periaatetta
- Sovelluksella on myös todella paljon konkreettisia riippuvuuksia
  - Scanner ja neljä Mongoon liittyvää luokkaa
- Mitä haittaa näistä ongelmista on?
  - Jos päätetään siirtyä käyttämään graafista käyttöliittymä, on luokan koodia hankala uusiokäyttää, sillä sovelluslogiikkaa ja käyttäjän kanssa kommunikointia on vaikea eriyttää koodista
  - Vastaavasti jos siirryttäisiin MongoDB:stä johonkin toiseen tallennusratkaisuun, tulisi ongelmia, sillä tietokantaoperaatiot on kirjoitettu muun koodin sekaan

Vastuiden eriyttäminen

- Koodi tulisikin eriyttää pienemmiksi, yhden vastuun periaatetta noudattaviksi, toistensa kanssa kommunikoiviksi luokiksi
- Aloitetaan rakenteen parantaminen eriyttämällä tietokantaoperaatiot omaksi luokakseen.



- Kuvassa tietokantaoperaatiot on eristetty luokkaan *OpiskelijaMongoDao* (eli data access object), joka kapseloi kaikki tietokantaan liittyvät riippuvuudet ja tarjoaa muulle ohjelmalle selkeät metodit Opiskelija-olioiden tallettamiseen ja eri kriteerein tapahtuvaan tietokannasta hakemiseen
- Seuraavilla sivuilla OpiskelijaMongoDao ja sitä käyttävä Ilmo

OpiskelijaMongoDAO

```
public class OpiskelijaMongoDao {
	private Datastore store;

	public OpiskelijaMongoDao(String palvelin, String tietokanta) {
		Morphia morphia = new Morphia();
		MongoClient mc = new MongoClient(palvelin);
		store = morphia.createDatastore(mc, tietokanta);
	}

	private Query<Opiskelija> query() {
		return store.createQuery(Opiskelija.class);
	}


	public List<Opiskelija> kaikki() {
		return query().asList();
	}


	public List<Opiskelija> ilmottautujat(String kurssi) {
		return query().field("ilmoittautumiset.kurssi").equal(kurssi).asList();
	}

	public Opiskelija haeNimellä(String nimi) {
		return query().field("nimi").equal(nimi).get();
	}

}
```

Ilmo jolla ei ole riippuvuuksia Mongoon

```
public class Ilmo {

    private OpiskelijaMongoDao opiskelijaMongoDao;

    public Ilmo(OpiskelijaMongoDao opiskelijaMongoDao) {
        this.opiskelijaDao = opiskelijaDao;
    }

    public void suorita() {

        System.out.println(”minkä kurssin ilmottautumiset haluat nähdä: ”);
        String kurssi = lukija.nextLine();

        List<Opiskelija> ilmot = opiskelijaDao.ilmottautujat(kurssi);
        for ( Opiskelija o: ilmot) {
            System.out.prinltn(o.getNimi());
        }

        // ...
        System.out.println(”kenen tiedot haluat nähdä: ”);
        String nimi = lukija.nextLine();

        Opiskelija opiskelija = opiskelijaDao.haeNimella(nimi);
        System.out.prinltn(opiskelija);

    }
}
```

Paranneltu Ilmo

- Sovellus käynnistään nyt seuraavasti:

```
OpiskelijaMongoDao dao = new OpiskelijaMongoDao(”ohtu.jamo.fi”, ”myDb”);
Ilmo sovellus = Ilmo(dao);
sovellus.suorita();
```

- Eli sovelluslogiikalle annetaan konstruktorin parametrina olio, jonka kautta se on yhteydessä tietokanaan
- Voisimme yleistää ratkaisua siten, että tekisimme rajapinnan, joka määrittelee tietokantayhteyden tarjoavan luokan metodit

```
public interface OpiskelijaDao {
    void talleta(Opiskelija o);
    List<Opiskelija> kaikki();
    List<Opiskelija> ilmottautujat(String kurssi);
    Opiskelija haeNimellä(String nimi);
    // …
}
```

- ja muuttaa Ilmo käyttämään tietokantayhteyttä, jonka tyyppinä rajapinta

```
public class Ilmo {
    public Ilmo(OpiskelijaDao opiskelijaDao) { … }

    // ...
}
```

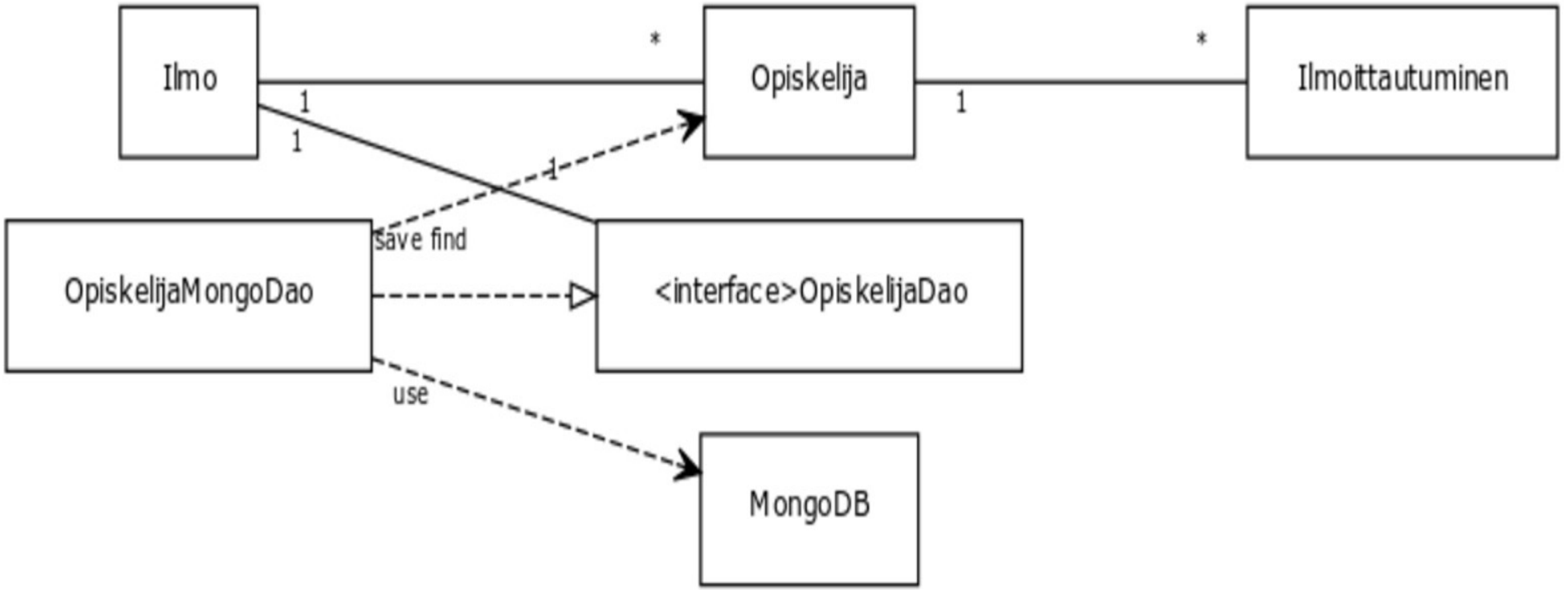
Paranneltu Ilmo: tietokantayhteys rajapinnan taakse

- Nyt voisimme määritellä vaihtoehtoisen, mongon sijaan esim. relaatiotietokantaa käyttävän olioiden talletusratkaisun

  public OpiskelijaMongoDao implements OpiskelijaDao { … }

  public OpiskelijaSqliteDao implements OpiskelijaDao { … }

- Sovelluslogiikasta huolehtiva luokka Ilmo tuntee ainoastaan rajapinnan, joten talletustavan muutos ei vaikuta muun sovelluksen toimintaan
- Tilanne vielä luokkakaaviona



Paranneltu Ilmo: tietokantayhteys rajapinnan taakse

- Noudatimmekin tässä oliosuunnittelun periaatetta **program to an interface, not to an Implementation**
  - Jos Ilmo tuntee ainoastaan rajapinnan, voidaan tallennusratkaisu helposti tulevaisuudessa vaihtaa
- Kuten tässäkin esimerkissä, laajennettavuuden kannalta ei ole hyvä idea olla riippuvainen konkreettisista luokista
  - Parempi on tuntea vain rajapintoja (tai abstrakteja luokkia) ja olla tietämätön siitä mitä rajapinnan takana on
  - Näin muutokset konkreettisissa luokissa eivät haittaa niin kauan kun rajapinta säilyy muuttumattomana
- Programming to an interface -periaate 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 tai vaikka korvata uudella luokalla vaikuttamatta luokan käyttäjiin

Oliosuunnittelun periaatteista

- Onko näissä periaatteissa järkeä? Kyllä, sillä niiden noudattaminen lisäävät ohjelmien ylläpidettävyyttä
- Kannattaako periaatteita noudattaa: useimmiten
  - joskus kuitenkin voi olla jonkun muun periaatteen nojalla viisasta rikkoa jotain toista periaatetta...
  - Jos kyseessä ”kertakäyttökoodi”, ei luonnollisesti kannata panostaa ylläpidettävyyteen
- ”ikiaikaisia periaatteita”, motivaationa ohjelman muokattavuuden, uusiokäytettävyyden ja testattavuuden parantaminen

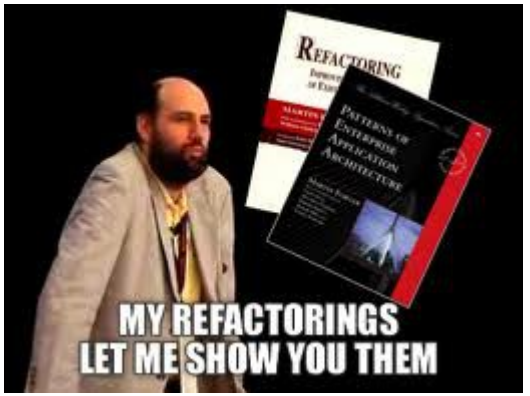
- Huonoa oliosuunnittelua on verrattu *velan* (engl. technical debt) ottamiseen
- Piittaamattomalla ja laiskalla ohjelmoinnilla/suunnittelulla saadaan ehkä nopeasti aikaan jotain, mutta hätäinen ratkaisu tullaan maksamaan korkoineen takaisin myöhemmin ***jos*** ohjelmaa on tarkoitus laajentaa tai muuttaa
- Joissain tilanteissa tosin velan ottaminen kannattaa
  - Voi olla elintärkeää saada tuote julkaistua nopeasti, muuten esim. saatetaan menettää markkina-asema ja tuote voi muuttua turhaksi

Oliosuunnittelun tärkeitä nimiä

- Oliosuunnittelun periaatteet siis ikiaikaisa, periaatteita ovat systematoisoineet mm. seuraavat henkilöt

Erich Gamma

Robert ”uncle bob” Martin



Martin Fowler

Kent Beck

Koodi haisee: merkki huonosta suunnittelusta

- Seuraavassa alan ehdoton asiantuntija Martin Fowler selittää mistä on kysymys **koodin hajuista**:
  - **A code smell is a surface indication that usually corresponds to a deeper problem in the system.** The term was first coined by Kent Beck while helping me with my Refactoring book.
  - The quick definition above contains a couple of subtle points. Firstly **a smell is by definition something that's quick to spot** - or sniffable as I've recently put it. *A long method is a good example of this - just looking at the code and my nose twitches if I see more than a dozen lines of java.*
  - The second is that smells don't always indicate a problem. Some long methods are just fine. You have to look deeper to see if there is an underlying problem there - smells aren't inherently bad on their own - they **are often an indicator of a problem rather than the problem themselves.**
  - One of the nice things about smells is tha**t it's easy for inexperienced people to spot them**, even if they don't know enough to evaluate if there's a real problem or to correct them. I've heard of lead developers who will pick a "smell of the week" and ask people to look for the smell and bring it up with the senior members of the team. Doing it one smell at a time is a good way of gradually teaching people on the team to be better programmers.

Koodihajuja

- Koodihajuja on hyvin monenlaisia ja monentasoisia
- Aloittelijankin on hyvä oppia tunnistamaan ja välttämään tavanomaisimpia
- Internetistä löytyy paljon hajulistoja, esim:
  - http://sourcemaking.com/refactoring/bad-smells-in-code
  - http://c2.com/xp/CodeSmell.html
  - http://wiki.java.net/bin/view/People/SmellsToRefactorings
  - http://www.codinghorror.com/blog/2006/05/code-smells.html
- Muutamia esimerkkejä aloittelijallekin helposti tunnistettavista hajuista:
  - Duplicated code (eli koodissa copy pastea...)
  - Methods too big
  - Classes with too many instance variables
  - Classes with too much code
  - Uncommunicative name
  - Comments

Koodin refaktorointi

- Lääke koodihajuun on *refaktorointi* eli muutos koodin rakenteeseen, joka kuitenkin pitää koodin toiminnan ennallaan
- Erilaisia koodin rakennetta parantavia refaktorointeja on lukuisia
  - ks esim. http://sourcemaking.com/refactoring
- Muutama hyvin käyttökelpoinen ja nykyaikaisessa kehitysympäristössä (esim NetBeans, Eclipse, IntelliJ) automatisoitu refaktorointi:
  - **Rename method** (rename variable, rename class)
    - Eli uudelleennimetään huonosti nimetty asia
  - **Extract method**
    - Jaetaan liian pitkä metodi erottamalla siitä omia apumetodejaan
  - **Extract interface**
    - Luodaan automaattisesti rajapinta perustuen jonkun luokan metodeihin ja korvataan suora riippuvuus luokkaan riippuvuudella luotuun rajapintaan

Koodin refaktorointi

- Pari sivua sitten tekemämme Ilmo-luokan muokkaaminen siten, että eristimme luokasta kaiken MongoDB-spesifisen koodin omaan luokkaansa oli myös refaktorointia, sillä koodin toiminnallisuus säilyi rakenteen muutoksesta huolimatta muuttumattomana

- Seuraavalla kahdella sivulla esimerkki vanhasta Ohjelmoinnin jatkokurssin tehtävästä
  - Koodi lisää luvun suuruusjärjestyksessä olevaan taulukkoon, jos luku ei ole ennestään taulukossa
- Ensin sotkuinen kaiken tekevä metodi
- Seuraavaksi refaktoroitu versio, jossa jokainen lisäämiseen liittyvä vaihe on erotettu omaksi selkeästi nimetyksi metodikseen
  - Osa näin tunnistetuista metodeista tulee käyttöön muidenkin metodien kuin lisäyksen yhteydessä
  - Lopputuloksena koodin rakenteen selkeys ja copy-pasten eliminointi joiden seurauksena ylläpidettävyys paranee

```
public boolean lisaa(int lisattava) {
    boolean loytyiko = false;
    for ( int i=0; i<alkioidenLkm; i++ )
        if ( taulukko[i]==lisattava ) loytyiko = true;
    if ( !loytyiko ) {
        if (alkioidenLkm == taulukko.length) {
            int[] uusi = new int[taulukko.length + kasvatuskoko];
            for (int i = 0; i < alkioidenLkm; i++) uusi[i] = taulukko[i];
            taulukko = uusi;
        }
        for( int i=0; i<alkioidenLkm; i++ )
            if ( i==alkioidenLkm || taulukko[i]>=lisattava ) {
                for ( int j=alkioidenLkm-1; i>=i; j-- )  taulukko[j+1] = taulukko[j];
                taulukko[i] = lisattava;
                alkioidenLkm++;
            }
        return true;
    }
    else return false;
}
```

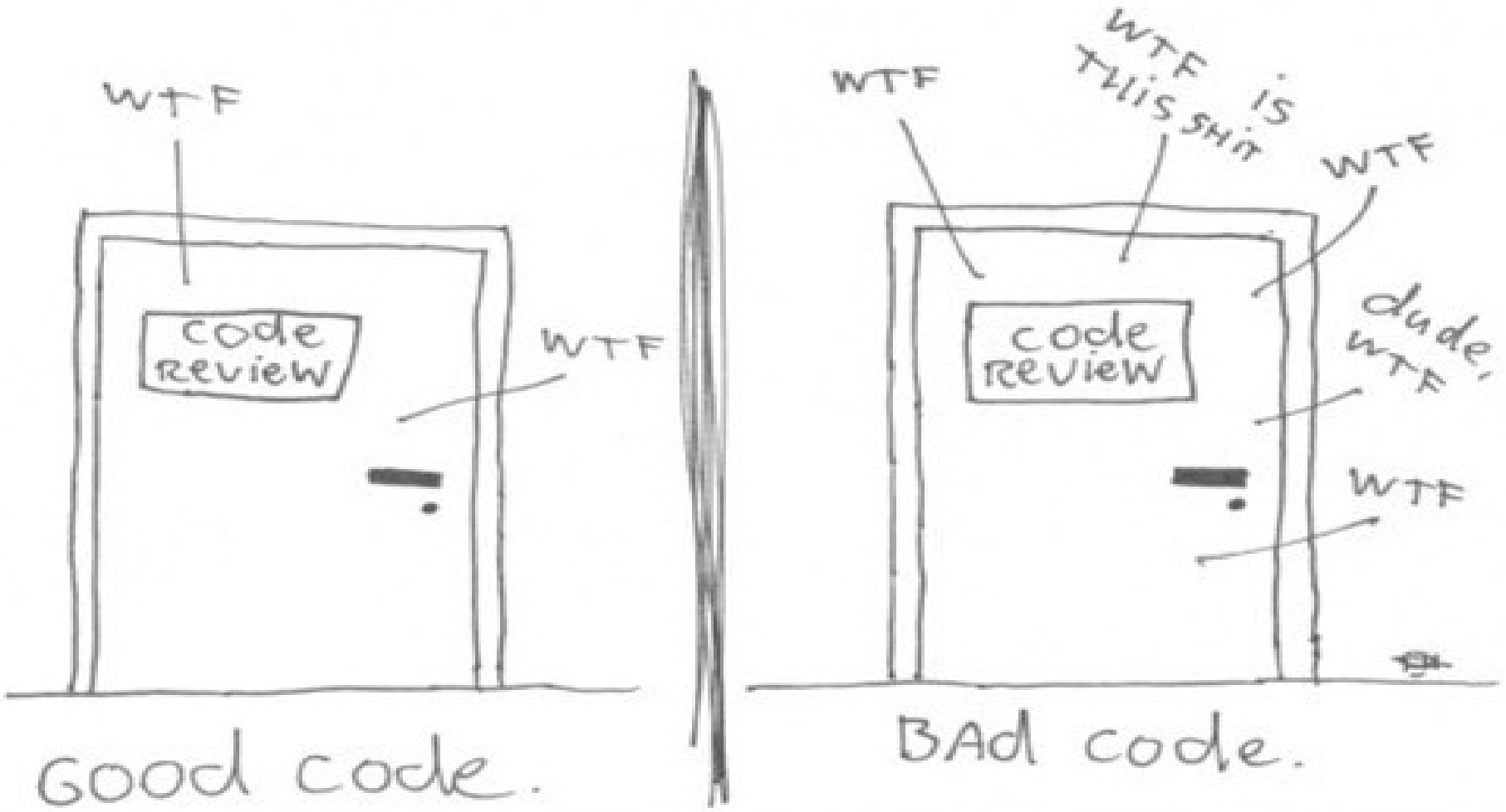
```
public boolean lisaa(int lisattava) {
    if ( onTaulukossa(lisattava) ) {
        return false;
    }

    if ( alkioidenLkm == taulukko.length ) {
        kasvataTaulukkoa();
    }

    int lisayskohta = etsiLisayskohta(lisattava);
    siirraEteenpainAlkaenKohdasta(lisayskohta);

    taulukko[lisayskohta] = lisattava;
    alkioidenLkm++;

    return true;
}
```

- Jokainen tässä kutsuttava metodi tekee yhden pienen asian
- Apumetodit ovat lyhyitä, helppoja ymmärtää ja tehdä
- Apumetodit on helppo testata oikein toimivaksi
- Koodi kannattaa kirjoittaa osin jo alusta asti suoraan ”puhtaaksi ja hajuttomaksi”
  - Helpompaa tehdä ja saada toimimaan oikein
- Toisen vuoden kurssilla *Ohjelmistotuotanto* palataan tarkemmin refaktorointiin ja puhtaan koodin kirjoittamiseen

Miten refaktorointi kannattaa tehdä

- Refaktoroinnin melkein ehdoton edellytys on kattavien yksikkötestien olemassaolo
  - Refaktoroinninhan on tarkoitus ainoastaan parantaa luokan tai komponentin sisäistä rakennetta, ulospäin näkyvän toiminnallisuuden pitäisi pysyä muuttumattomana
- Kannattaa ehdottomasti edetä pienin askelin
  - Yksi hallittu muutos kerrallaan
  - Testit on ajettava mahdollisimman usein ja varmistettava, että mikään ei mennyt rikki
- Refaktorointia kannattaa suorittaa lähes jatkuvasti
  - Koodin ei kannata antaa ”rapistua” pitkiä aikoja, refaktorointi muuttuu vaikeammaksi
  - Lähes jatkuva refaktorointi on helppoa, pitää koodin rakenteen selkeänä ja helpottaa sekä nopeuttaa koodin laajentamista
- Osa refaktoroinneista, esim. metodien tai luokkien uudelleennimentä tai pitkien metodien jakaminen osametodeiksi on helppoa, aina ei näin ole
  - Joskus on tarve tehdä isoja refaktorointeja joissa ohjelman rakenne eli arkkitehtuuri muuttuu

The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE



Test Driven Development

Yksikkötestien kirjoittaminen valmiille koodille on työlästä ja tylsää

- Kirjoittamalla testejä ainoastaan valmiille koodille jää huomattava osa yksikkötestien hyödyistä saavuttamatta
  - Esim. refaktorointi edellyttäisi testejä
- JUnit ei ole alunperin tarkoitettu jälkikäteen tehtävien testien kirjoittamiseen, JUnitin kehittäjällä Kent Beckillä oli alusta asti mielessä jotain paljon järkevämpää ja mielenkiintoisempaa

Test Drive it!

TDD eli Test Driven Development

- TDD:ssä ohjelmoija (eikä siis erillinen testaaja) kirjoittaa testikoodin
- Testit laaditaan ennen koodattavan luokan toteutusta, yleensä jo ennen lopullista suunnittelua
- Sovelluskoodi kirjoitetaan täyttämään testien asettamat vaatimukset
  - Testit määrittelevät miten ohjelmoitavan luokan tulisi toimia
  - Testit toimivatkin osin koodin dokumentaationa, sillä testit myös näyttävät miten testattavaa koodia käytetään
- Testaus ohjaa kehitystyötä eikä ole erillinen toteutuksen jälkeinen laadunvarmistusvaihe
  - Oikeastaan TDD ei ole testausmenetelmä vaan ohjelmiston kehitysmenetelmä, joka tuottaa sivutuotteenaan automaattisesti ajettavat testit
- Testien on ennen toteutuksen valmistumista epäonnistuttava
  - Näin pyritään varmistamaan, että testit todella testaavat haluttua asiaa
- Toiminnallisuus katsotaan toteutetuksi, kun testit menevät läpi

Oikeaoppinen TDD-sykli

(1) tehdään **yksi** testitapaus

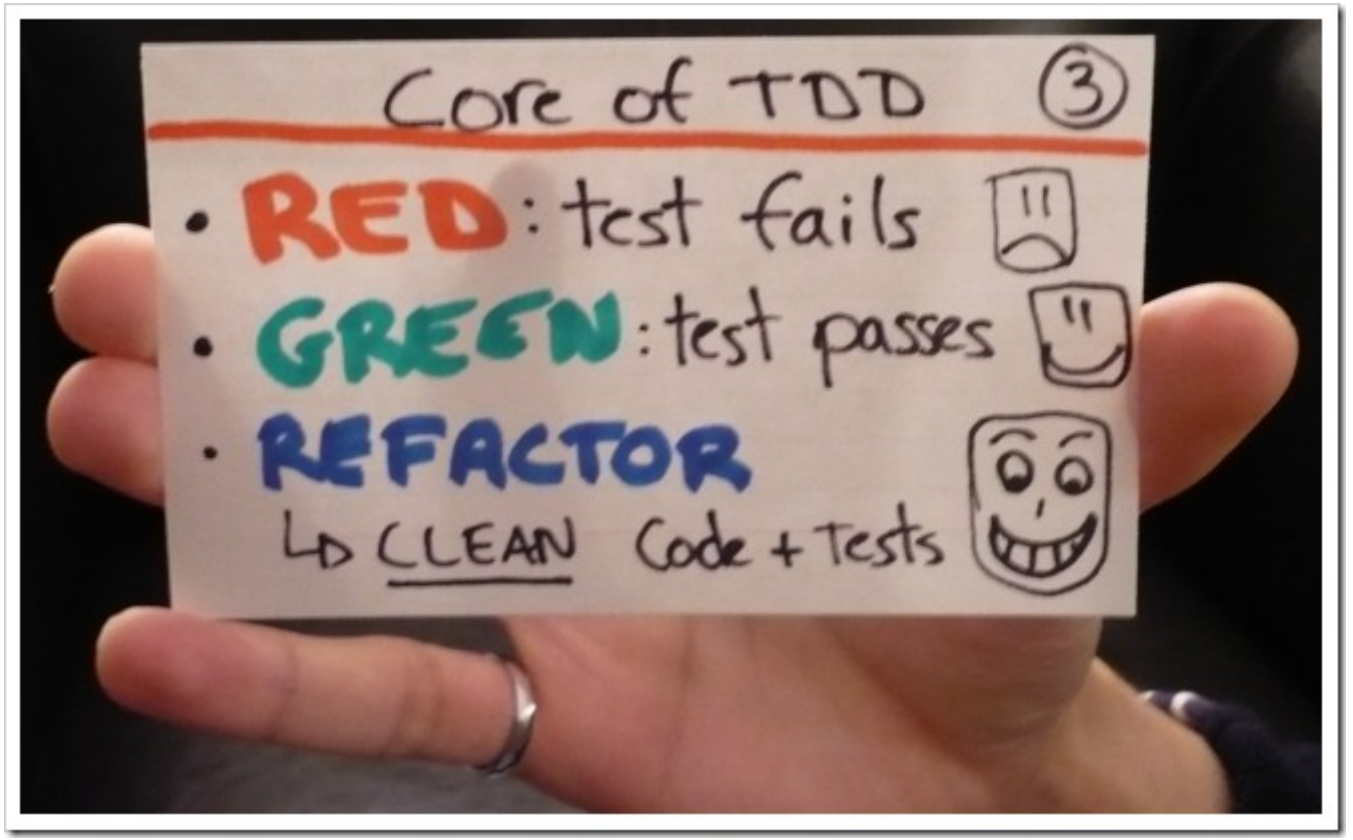
- testitapaus testaa ainoastaan yhden ”pienen” asian

(2) Tehdään koodi joka läpäisee testitapauksen

(3) **refaktoroidaan** koodia, eli parannellaan koodin laatua ja struktuuria

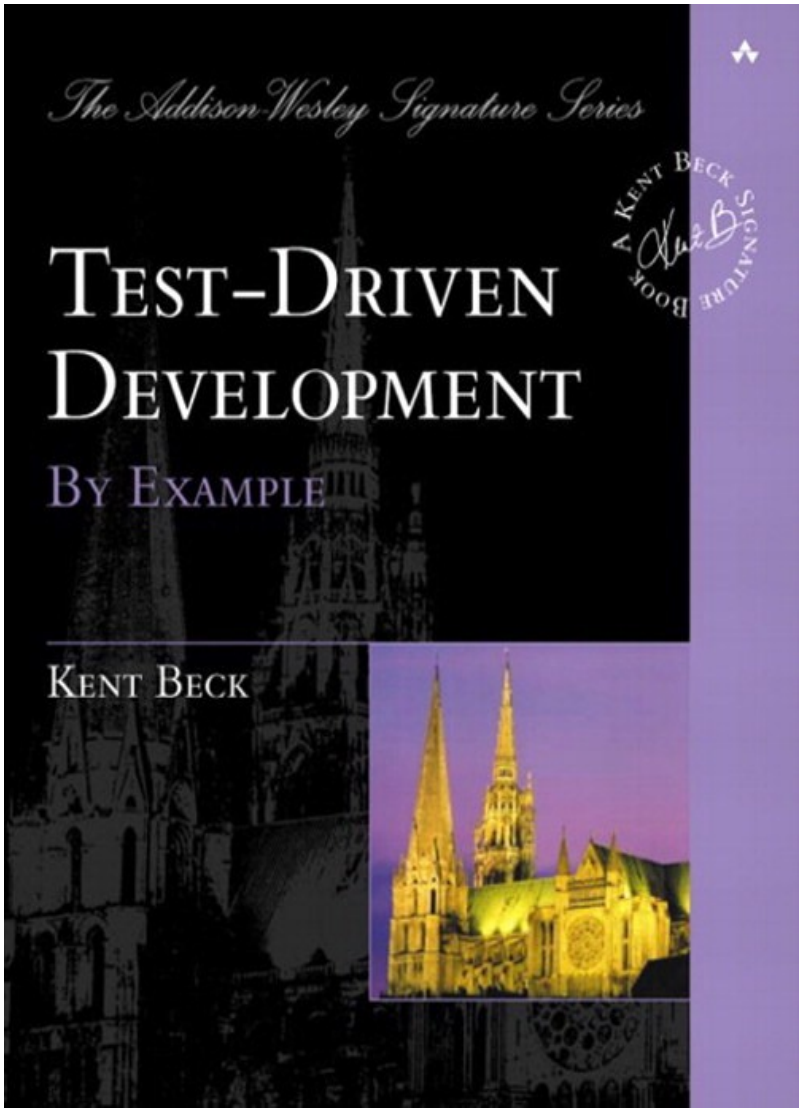
- Testit varmistavat koko ajan ettei mitään mene rikki

Kun koodin rakenne on kunnossa, palataan vaiheeseen (1)



TDD

- Automaattinen testaus ja TDD ovat usein osana ketterää ohjelmistokehitystä
- Mahdollistaa turvallisen refaktoroinnin
  - Koodi ei rupea haisemaan
  - Ohjelman rakenne säilyy laajennukset mahdollistavana
- Seuraavan viikon laskareiden paikanpäällä tehtävässä tehtävässä pääsemme itse kokeilemaan TDD:tä

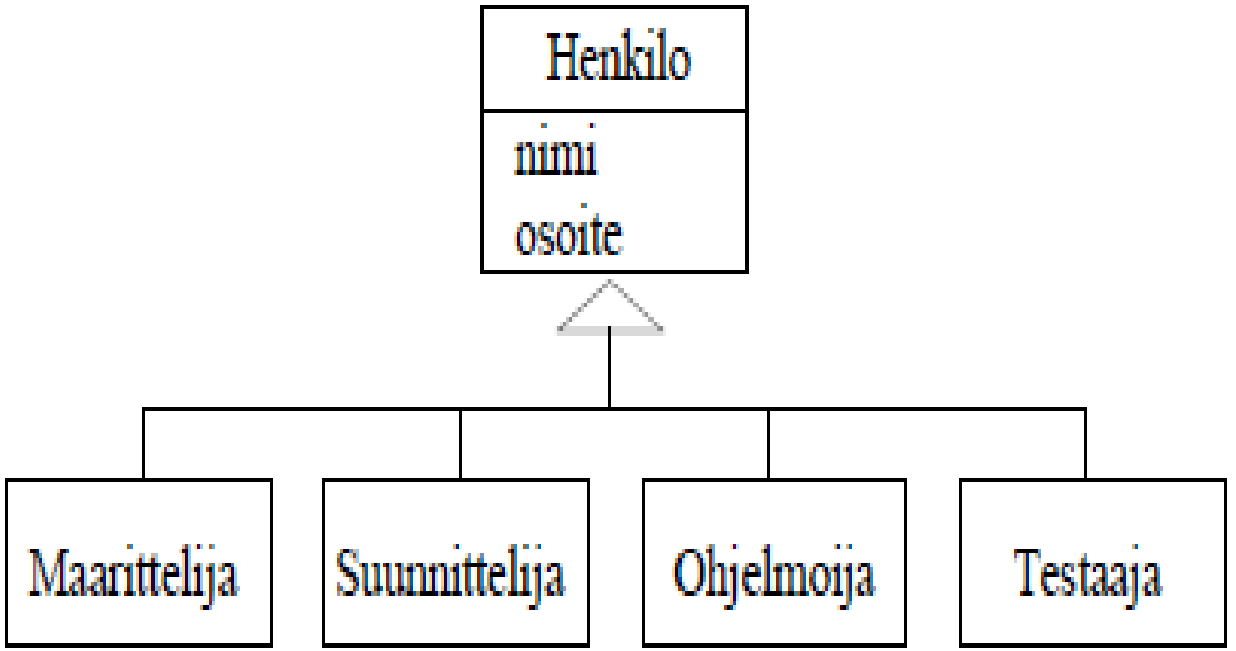


Esimerkkejä oliosuunnittelusta

perimisen virheellisiä ja oikeaoppisia käyttötapoja

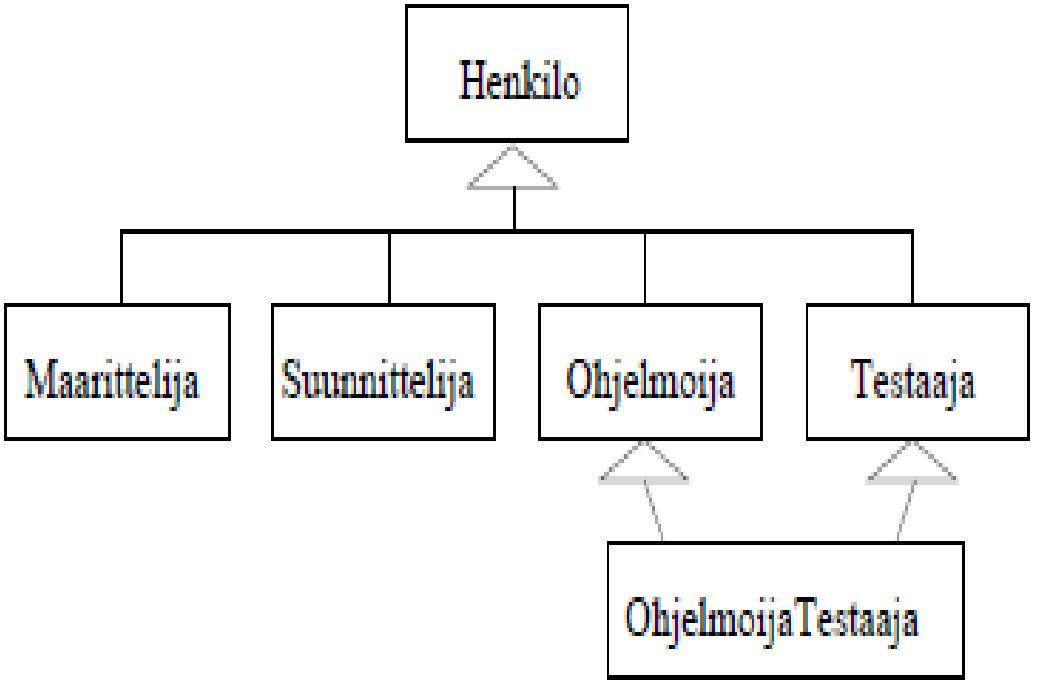
Esimerkki periytymisen virheellisestä käyttöyrityksestä

- 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...



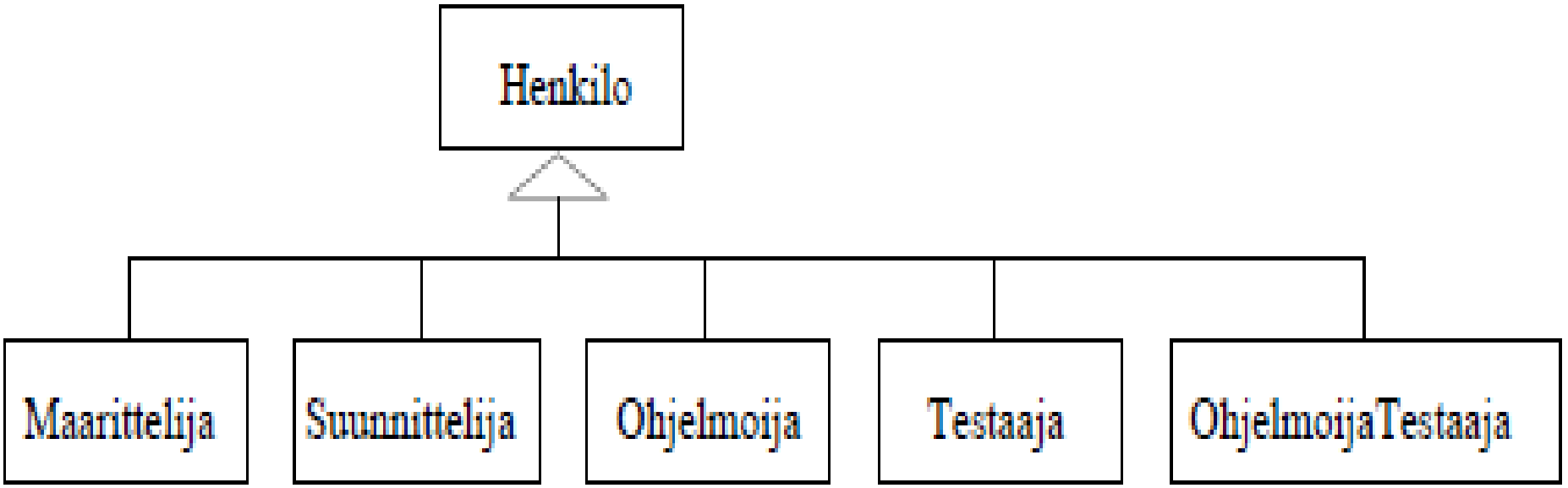
Ongelmia

- 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

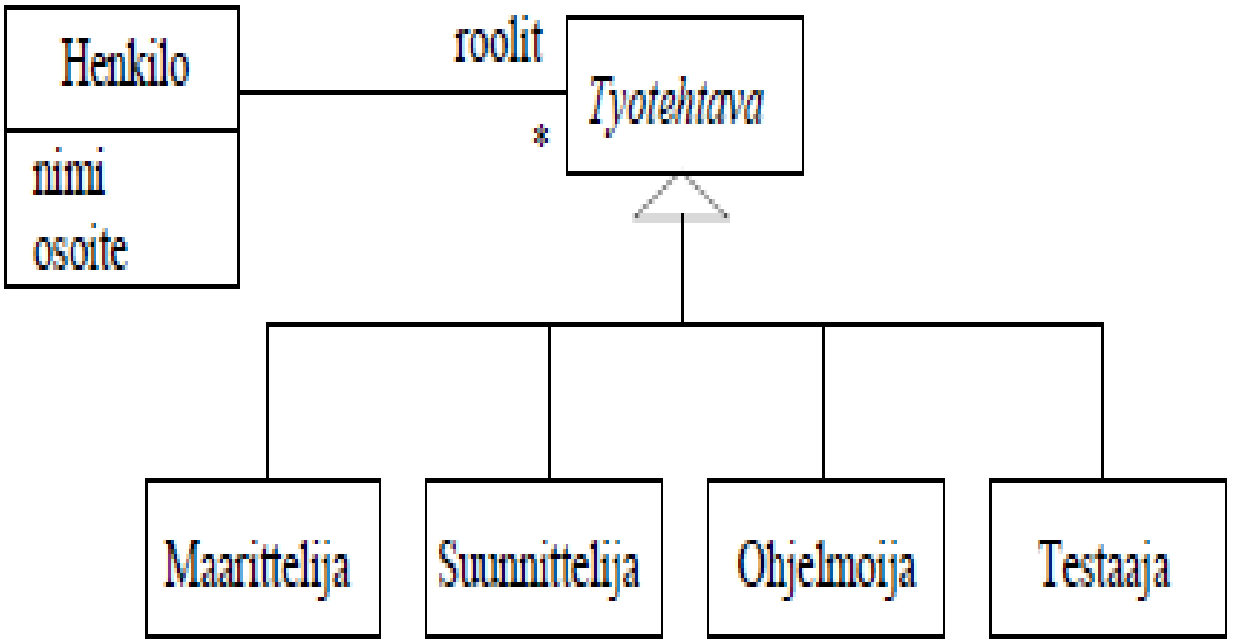
- Jos toteutuskieli ei tue moniperintää, yksi vaihtoehto on jokaisen työyhdistelmän kuvaaminen omana suoraan Henkilön alla olevana aliluokkana
  - Erittäin huono ratkaisu: nyt esim. OhjelmoijaTestaaja ei peri ollenkaan Ohjelmoija- eikä Testaaja-luokkaa
    - Seurauksena se, että samaa esim. Ohjelmoija-luokaan 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



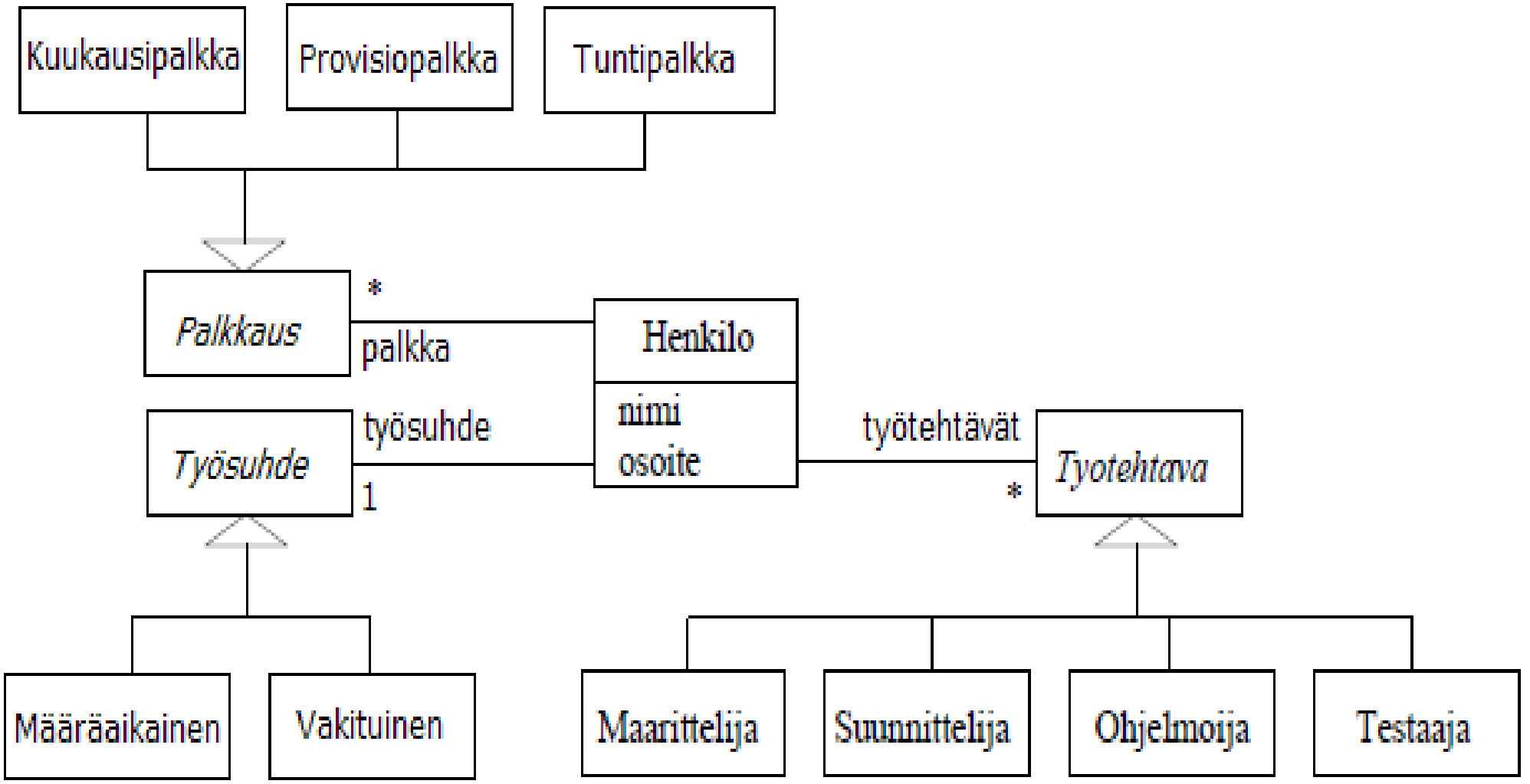
- Tyotehtava on nyt abstrakti luokka, sillä se on pelkkä käsite, jonka merkitys konkretisoituu vasta aliluokissa, esim. Ohjelmoija osaa koodata...
- Ratkaisun ”hinta”, on luokkien määrän kasvu
  - yhtä käsitettä, esim. ohjelmointia tekevää työntekijää kuvataan usealla oliolla: Henkilö-olio ja siihen 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...

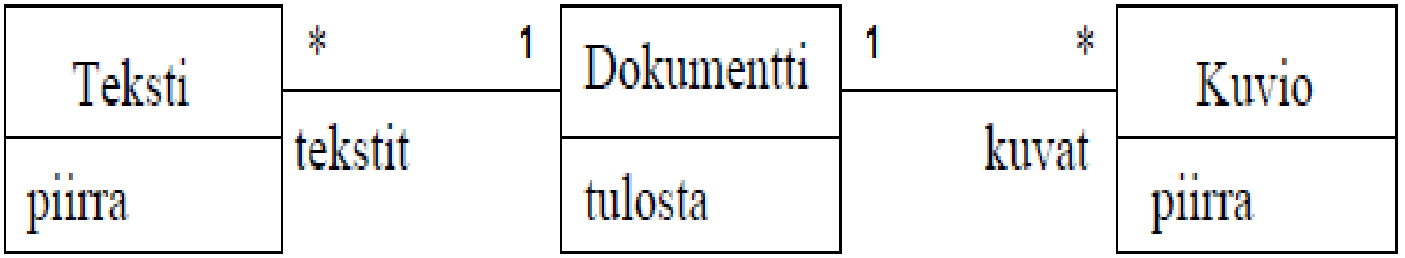
- Henkilöön voisi liittyä muitakin rooleja kun työntekijärooleja, esim.
  - työsuhteen laatua kuvaava rooli (vakinainen, määräaikainen)
  - palkkausta kuvaava rooli (tuntipalkka, kuukausipalkka, …)
  - ks. seuraava sivu



- 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

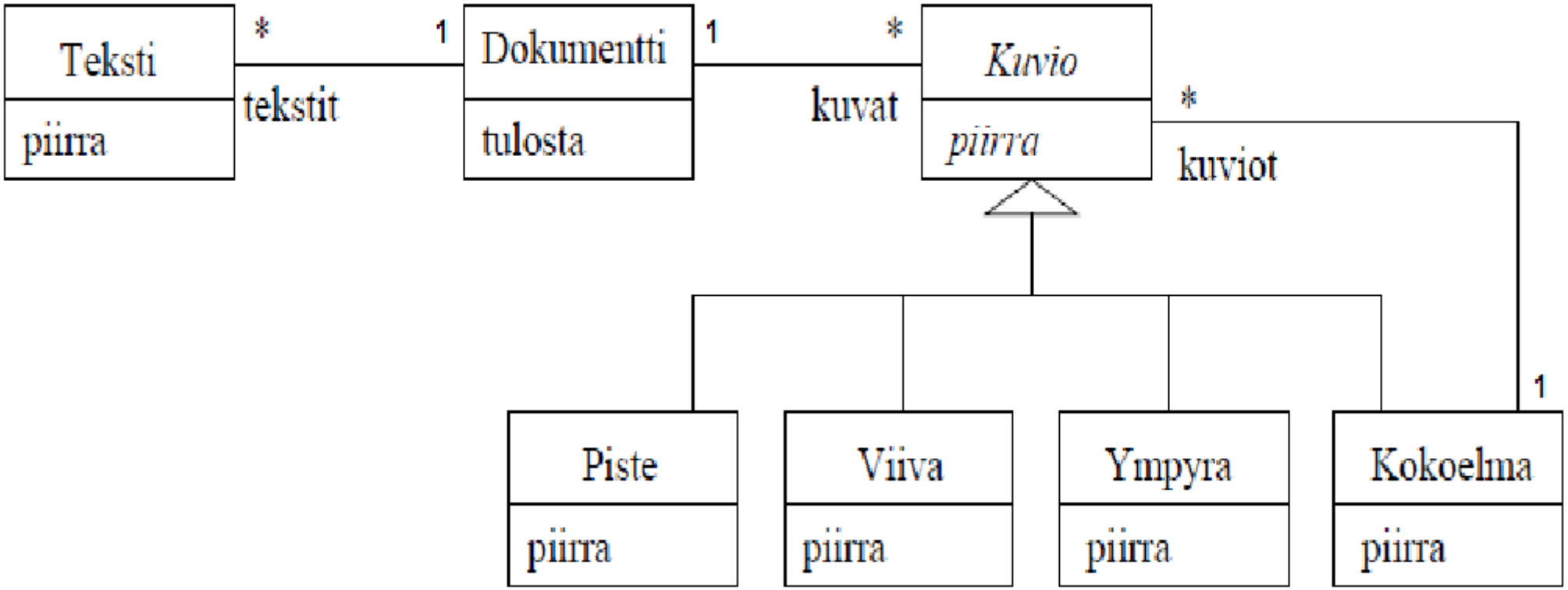
  piste, viiva tai ympyrä, tai
  - Edellisistä koostuva monimutkaisempi kuvio

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

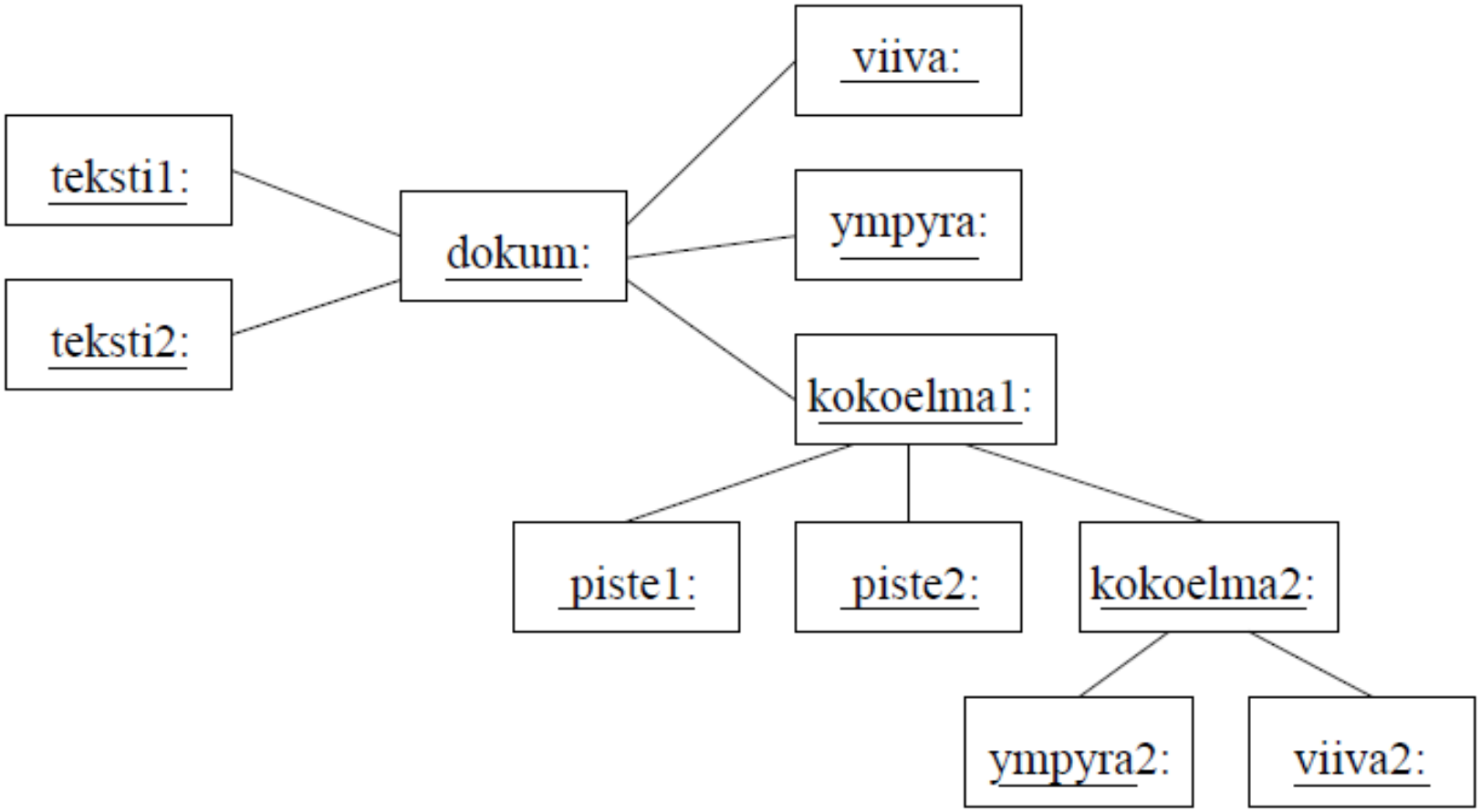
- Tarkennetaan määritelmää. Kuvio on, joko
  - piste,
  - viiva,
  - ympyrä tai
  - kokoelma kuvioita
- Luokkakaavio seuraavalla sivulla

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

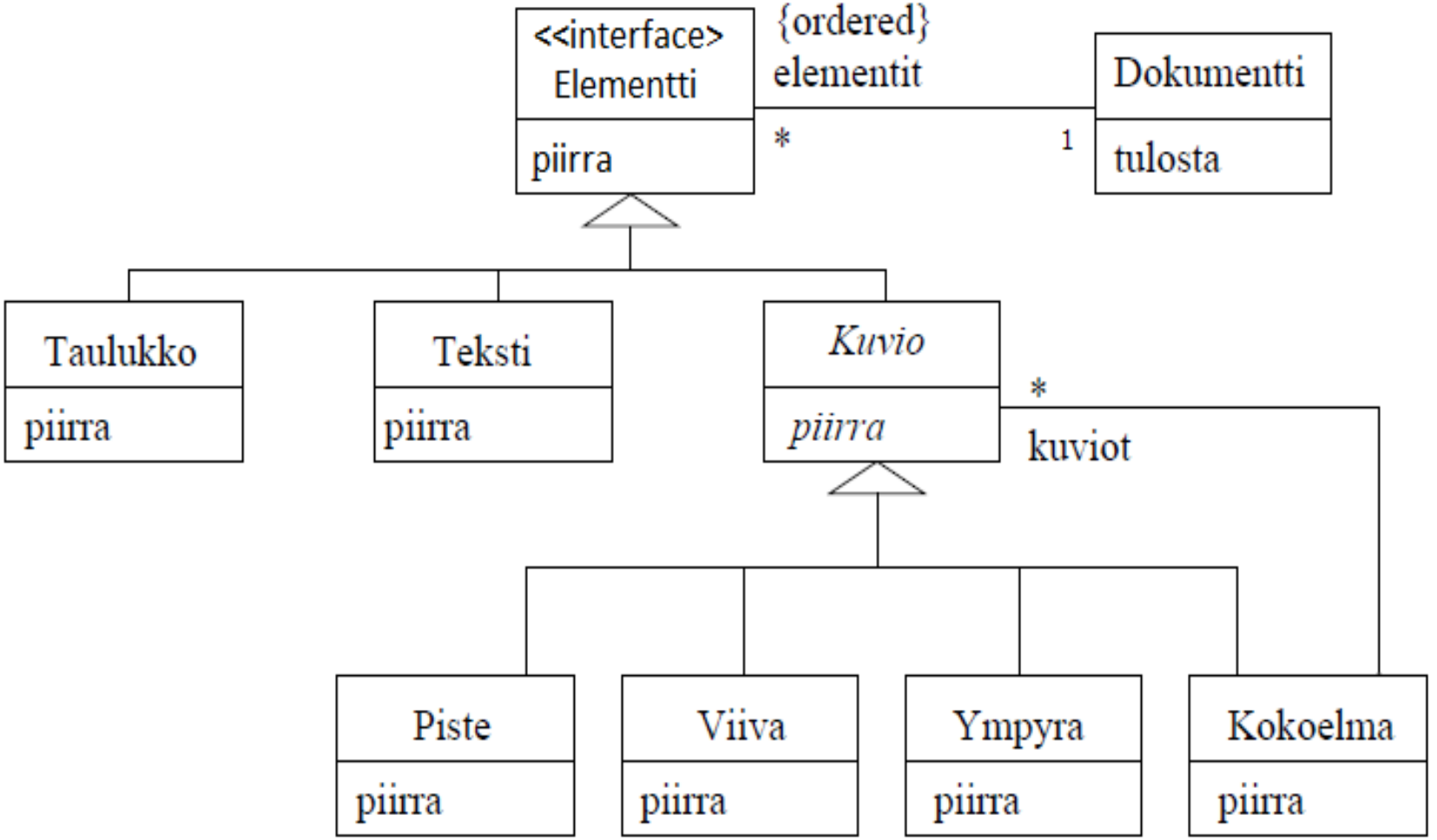
- Dokumentissa voisi olla muunkinlaisia rakenneosia kuin tekstiä ja kuvia, esim. taulukoita
- Mallia onkin järkevä yleistää, ja määritellä kaikille dokumentin rakenneosille yhteinen rajapinta Elementti, jolle on määritelty metodi piirrä jonka kaikki konreettiset elementit toteuttavat
- Dokumentti tuntee 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*