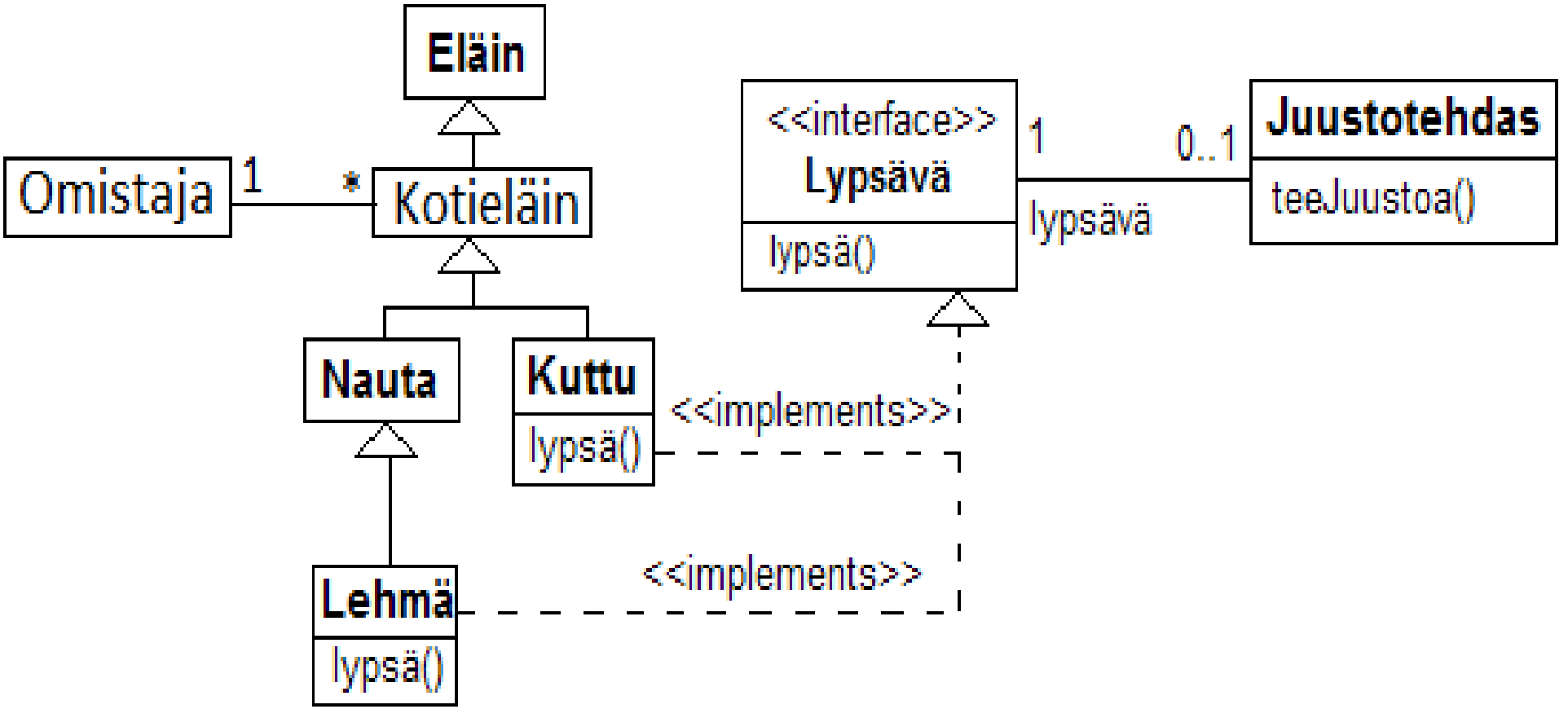
Ohjelmistojen mallintaminen

Luento 8, 26.11.

Kertaus: yleistys-erikoistus ja perintä

- Nauta, Lehmä ja Kuttu ovat Kotieläimiä, Kotieläimet Eläimiä
- Kotieläimillä (siis myös Naudoilla, Lehmillä ja Kutuilla) on Omistaja
- Kuttu ja Lehmä toteuttavat rajapinnan Lypsävä, eli ne lupaavat toteuttaa metodin lypsä()
- Juustotehdas tuntee jonkun Lypsävä-rajapinnan toteuttavan eläimen



Kertaus: oliosuunnittelun periaatteita

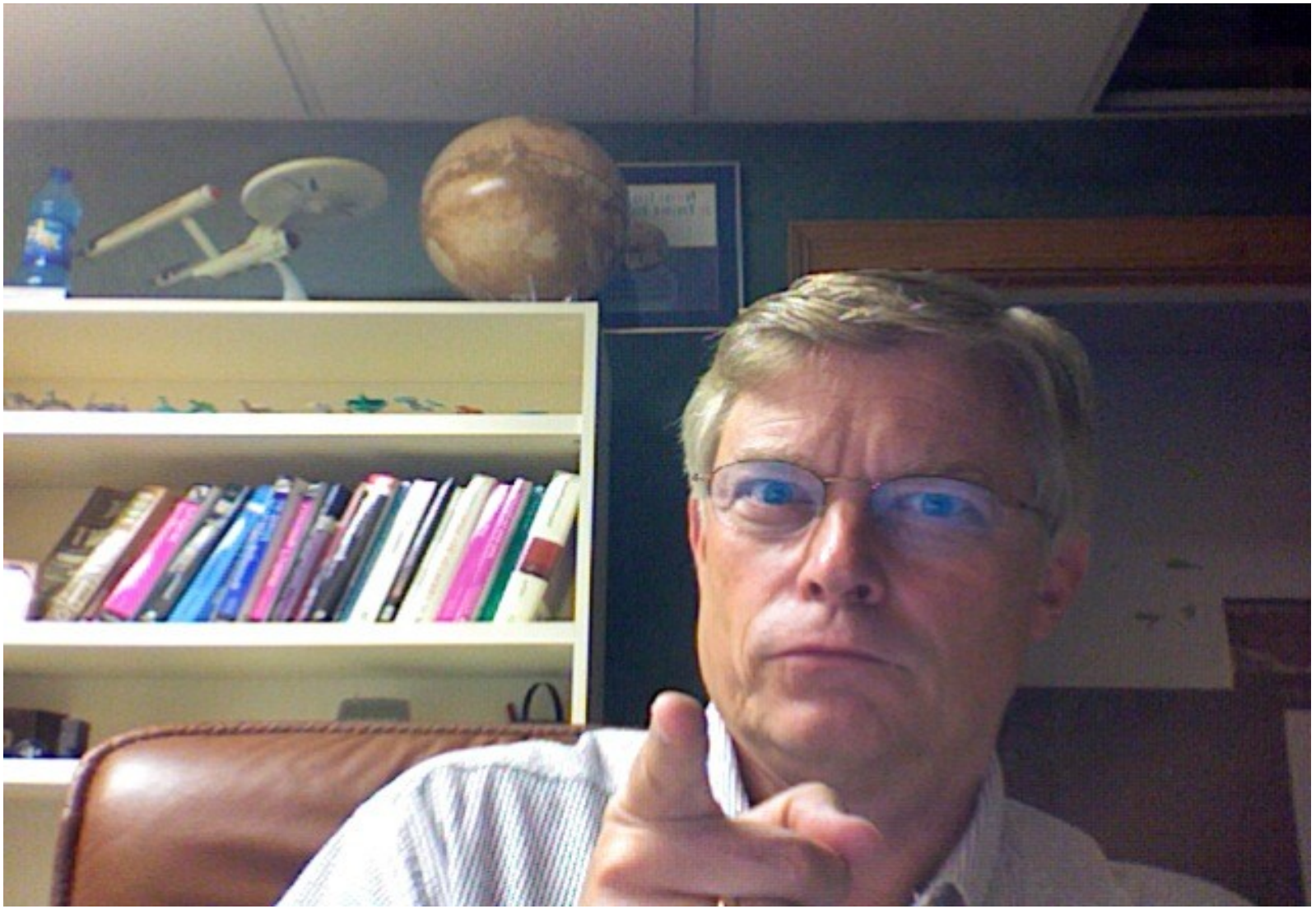
- **Single responsibility**
  - Jokaisella luokalla vain yksi selkeä vastuu
- **Favour composition over inheritance**
  - Älä liikakäytä perintää
  - Kohta esimerkki joka kuvaa liikakäytön ongelmia
- **Program to an interface, not to an Implementation**
  - Tee luokat mahdollisimman riippumattomiksi toisistaan
  - Tunne vain rajapinta (kuten Juustotehdas!)
- Motivaationa ohjelman muokattavuuden ja uusiokäytettävyyden parantaminen
- ”ikiaikaisia periaatteita”, systematisoijina mm

Martin Fowler

Erich Gamma

Kent Beck

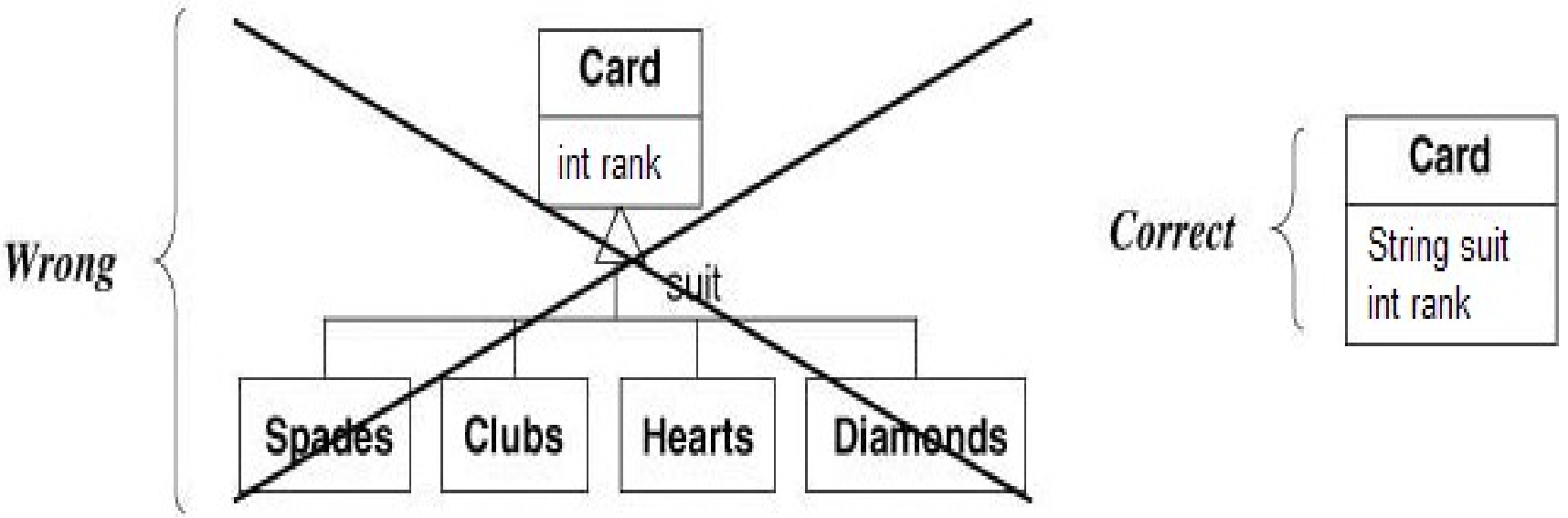
ja ennenkaikkia uncleBob Martin



- **Muitakin periaatteita on, niihin palataan syksyllä 2011 kurssilla *Ohjelmistokehitys***

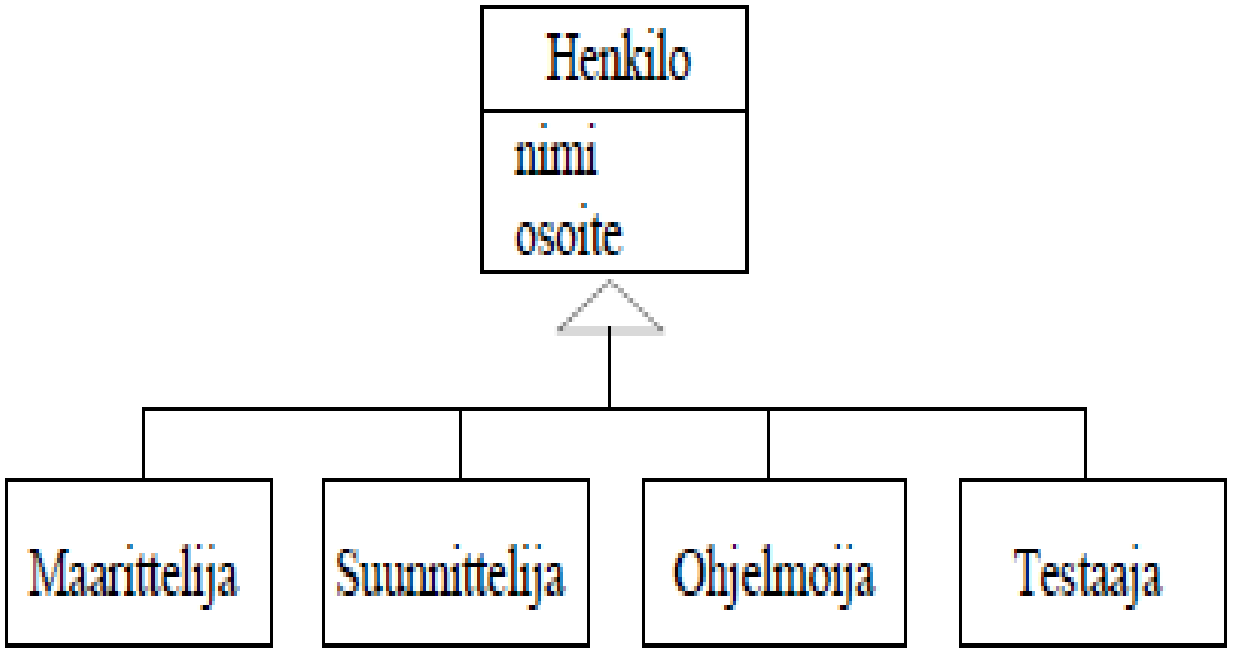
Milloin ei kannata periä

- Pelikorteilla on maa (suit) ja arvo (rank)
- Yksi mahdollisuus olisi luoda yliluokka pelikortti, johon liittyy arvo ja periä siitä aliluokat Pata, Ruutu, Risti ja Hertta
- Jos korttipakkaa käytetään peliin, jossa eri maiden kortit käyttäytyvät suunnilleen samalla tavalla (esim. pokeri), ei kannata perimistä kannata käyttää
  - Kortin maa voidaan ilmaista attribuuttina samoin kuin arvo
- Toisaalta jos pelissä eri maihin liittyy radikaalisti erilaista käyttäytymistä voikin olla viisasta kuvata maat erillisinä aliluokkina
  - Eli perinnän käyttämisen järkevyyskin on tilannekohtaista



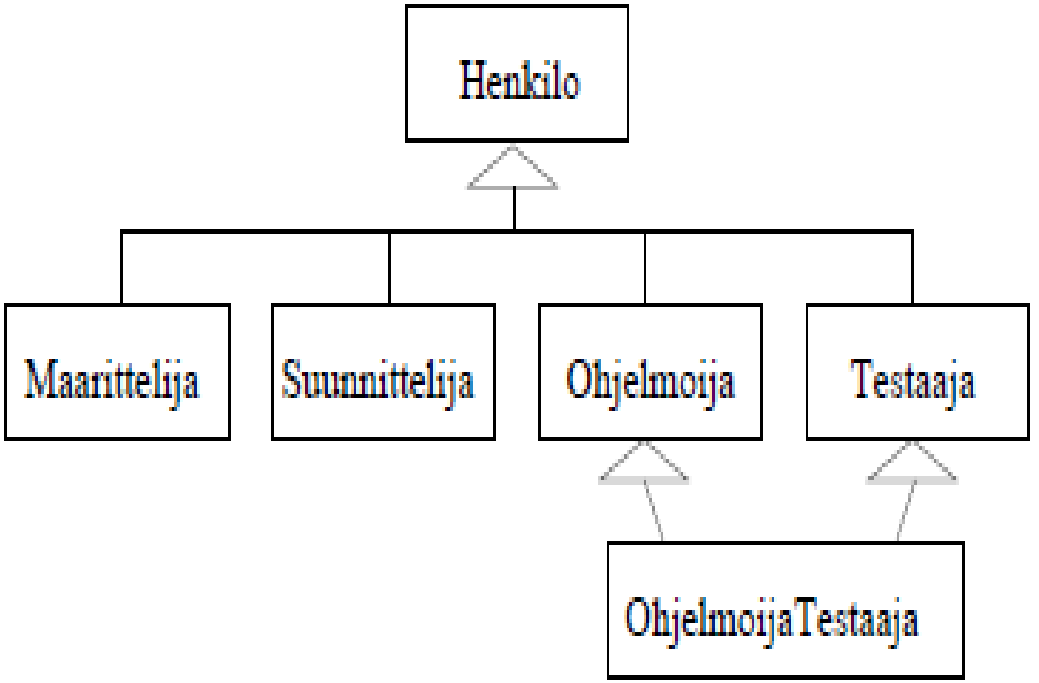
Esimerkki periytymisen virheellisestä käyttöyrityksestä

- 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...



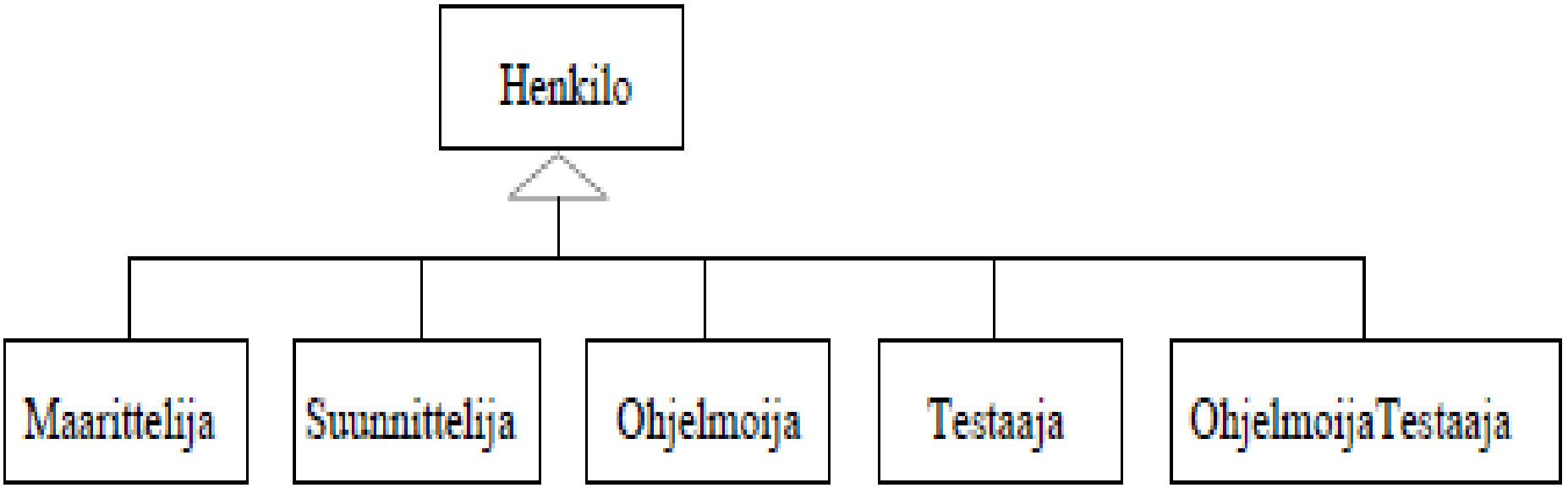
Ongelmia

- 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 tarvittaisiin10 kappaletta
  - Kuten mainittua, esim. Javassa ei ole moniperintää



Huono ratkaisuyritys

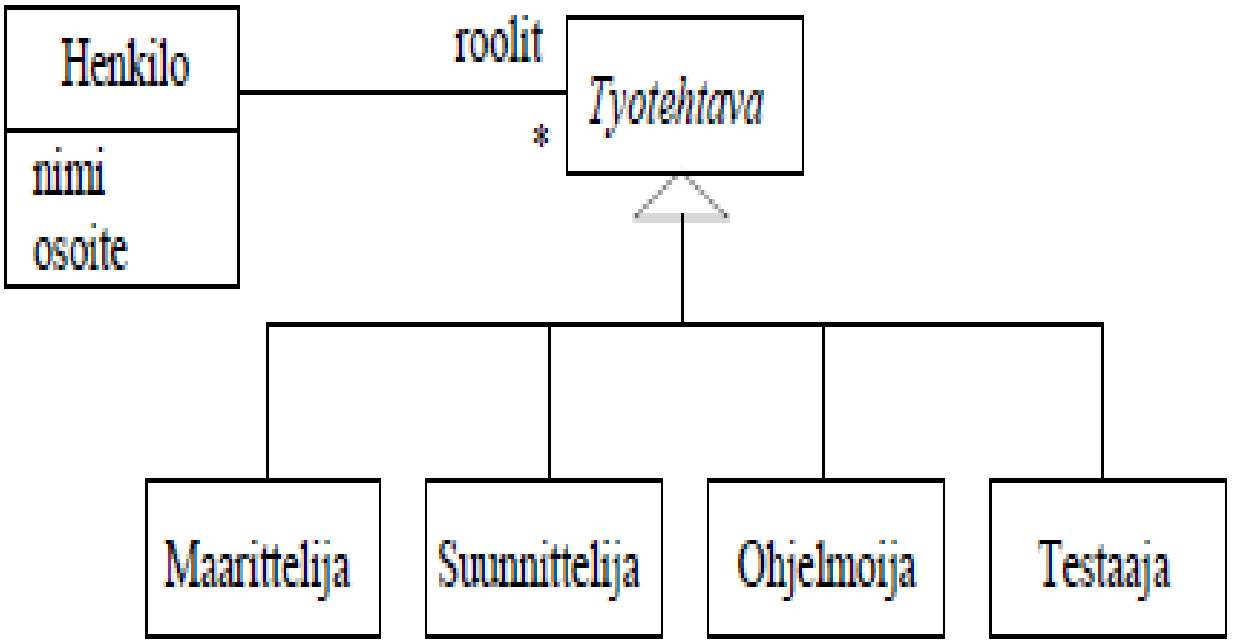
- Jos toteutuskieli ei tue moniperintää, yksi vaihtoehto on jokaisen työyhdistelmän kuvaaminen omana suoraan Henkilön alla olevana aliluokkana
  - Erittäin huono ratkaisu: nyt esim. OhjelmoijaTestaaja ei peri ollenkaan Ohjelmoija- eikä Testaaja-luokkaa
    - Seurauksena se, että samaa esim. Ohjelmoija-luokaan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



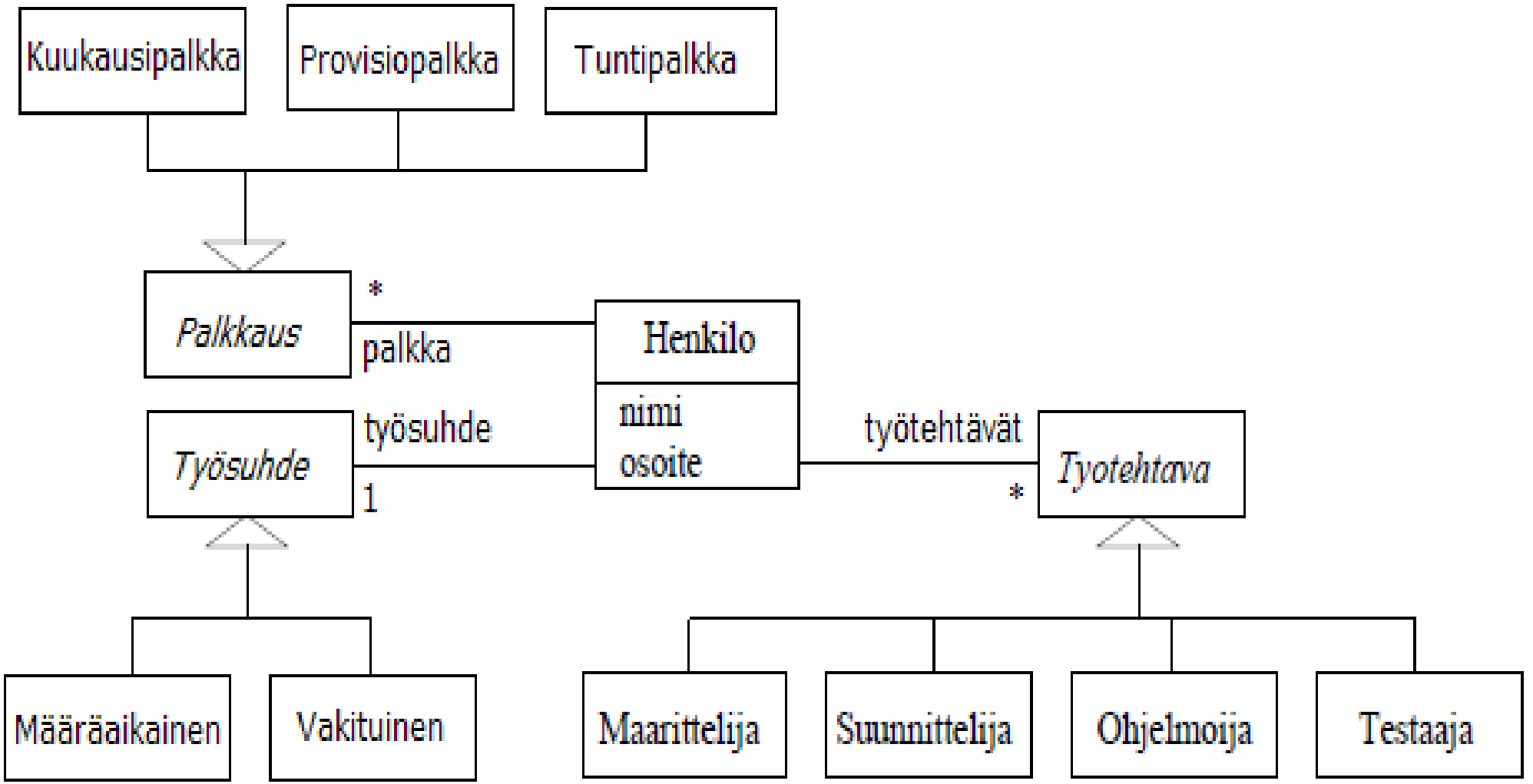
- Tyotehtava on nyt abstrakti luokka, sillä se on pelkkä käsite, jonka merkitys konkretisoituu vasta aliluokissa, esim. Ohjelmoija osaa koodata...
- Ratkaisun ”hinta”, on luokkien määrän kasvu
  - yhtä käsitettä, esim. ohjelmointia tekevää työntekijää kuvataan usealla oliolla: Henkilö-olio ja siihen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

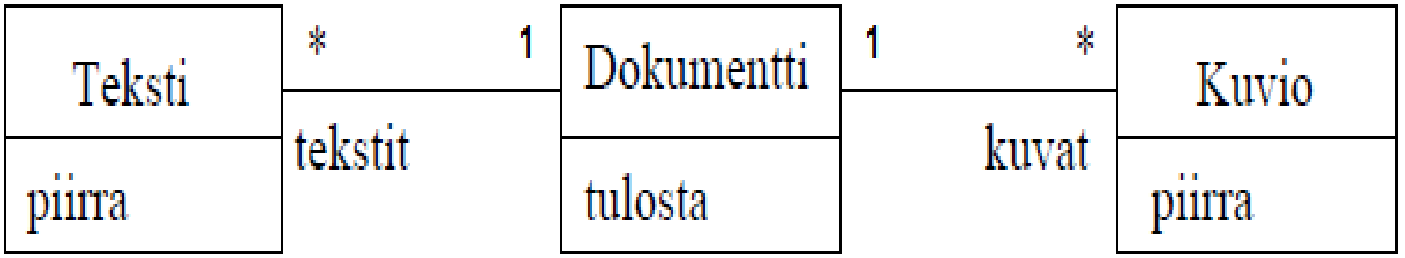
- Henkilöön voisi liittyä muitakin rooleja kun työntekijärooleja, esim.
  - työsuhteen laatua kuvaava rooli (vakinainen, määräaikainen)
  - palkkausta kuvaava rooli (tuntipalkka, kuukausipalkka, …)
  - ks. seuraava sivu



- 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

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

Tarkennetaan kuvioa

- Kuvio voi siis olla

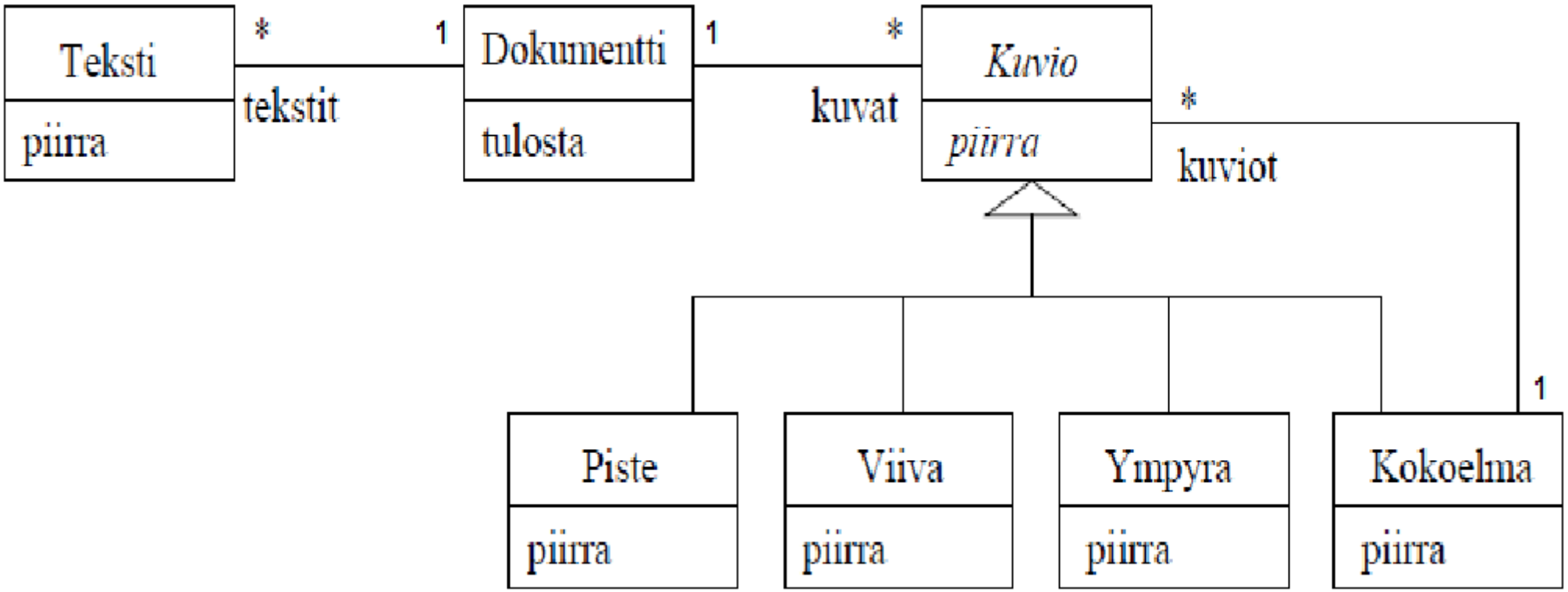
  piste, viiva tai ympyrä, tai
  - Edellisistä koostuva monimutkaisempi kuvio

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

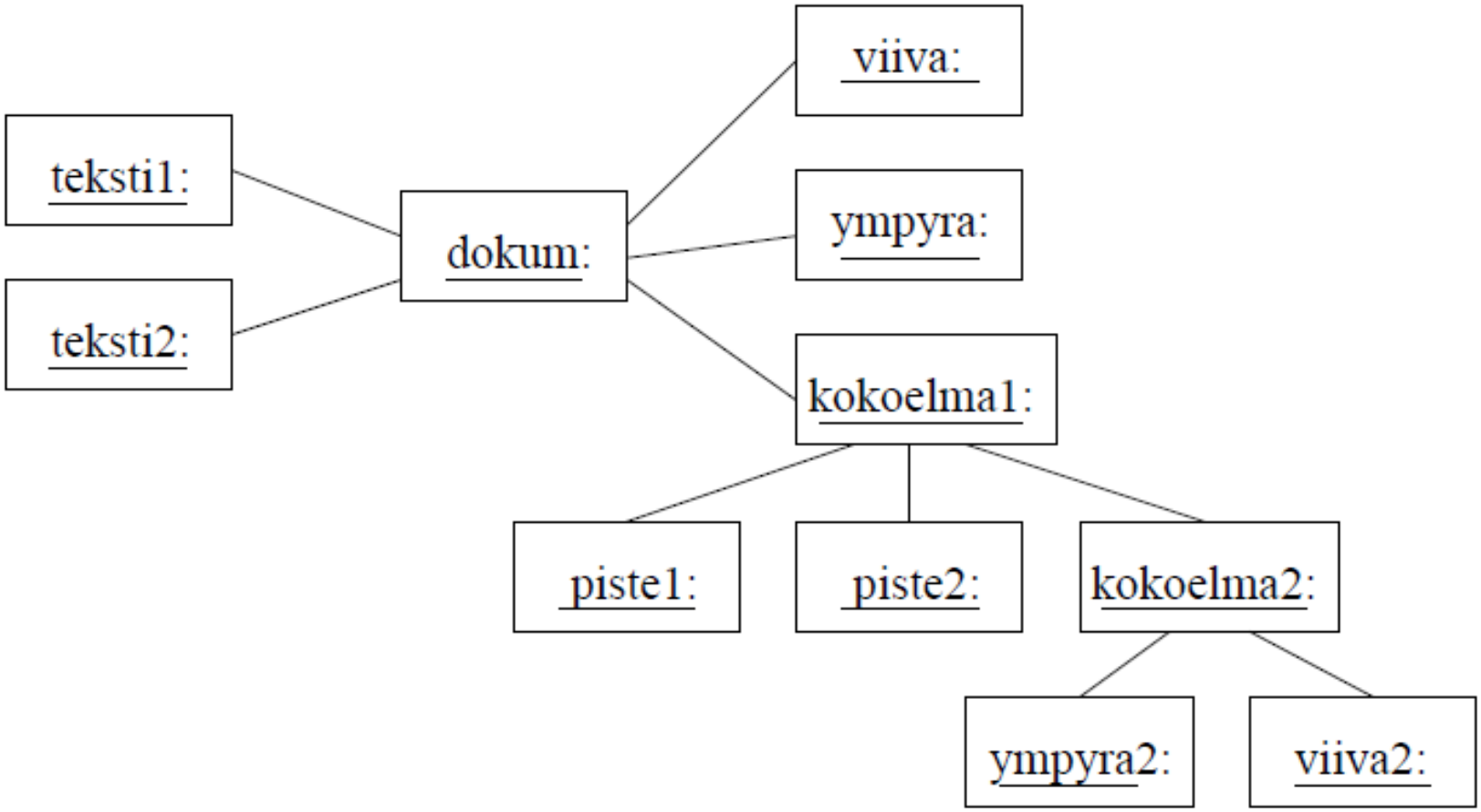
- Tarkennetaan määritelmää. Kuvio on, joko
  - piste,
  - viiva,
  - ympyrä tai
  - kokoelma kuvioita
- Luokkakaavio seuraavalla sivulla

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

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

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

Yleistetään mallia vielä hiukan

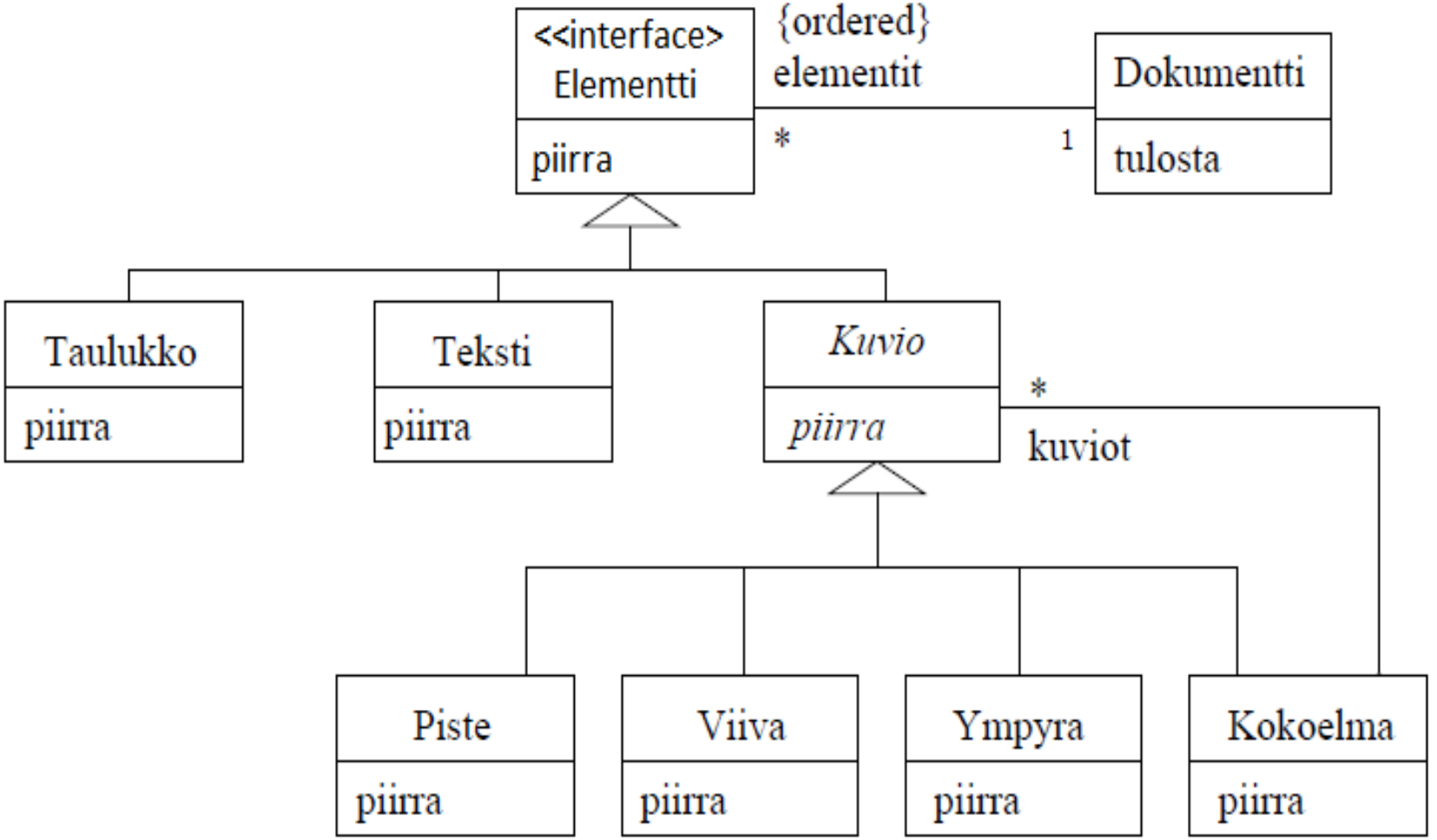
- Dokumentissa voisi olla muunkinlaisia rakenneosia kuin tekstiä ja kuvia, esim. taulukoita
- Mallia onkin järkevä yleistää, ja määritellä kaikille dokumentin rakenneosille yhteinen rajapinta Elementti, jolle on määritelty metodi piirrä jonka kaikki konreettiset elementit toteuttavat
- Dokumentti tuntee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

```
class Dokumentti {

   ArrayList<Elementti> elementit;

   void tulosta(){

      for ( i=0 ; i < elementit.length; i++ )

           elementit[i].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*

Menetelmä

- UML tarjoaa joukon enemmän tai vähemmän hyödyllisiä kaaviota
  - Standardi ei kuitenkaan neuvo milloin ja miten kaavioita kannattaisi käyttää

- Vuosien saatossa on kehitetty lukuisia **menetelmiä** (engl. method) ohjelmistojen kehittämiseen
- Menetelmä kertoo
  - miten missäkin ohjelmiston kehittämisen vaiheessa (määrittely, suunnittelu, ohjelmointi, testaus) tulisi edetä
  - Mitä tekniikoita, strategioita, apuvälineitä ja mallinnustekniikoita (esim. UML-kaavioita) tulisi käyttää ohjelmiston kehittämisen tukena
  - Miten esim. mallinnustekniikoita kannattaa soveltaa

- Tällä kurssilla olemme kiinnostuneita **oliomenetelmistä**
  - Menetelmät hyödyntävät usein UML:ää

Oliomenetelmät

- Oliomenetelmiä on olemassa lukuisia
  - Varhaisimmat (kehittäjinä mm. Booch, Jacobson, Rumbaugh, Yourdon) 80-90-luvun vaihteesta
  - Menetelmissä paljon samaa, joitakin vivahde-eroja
- Melkein kaikki oliomenetelmät sisältävät seuraavat vaiheet
  - **Olioanalyysi:** määrittelyvaiheen luokkamallin laatiminen
  - **Oliosuunnittelu: l**uokkamallin jalostaminen teknisemmäksi
  - Toteutus **olio-ohjelmointikielellä**
- Tutustutaan pintapuolisesti erääseen oliomenetelmään esimerkkisovelluksen avulla
  - Esitys perustuu *Craig Larmanin* kirjaan *Applying UML and Patterns*
  - Haastavaa on *oliosuunnittelu*
  - Larmanin kirjassa sovelletaan ns. *vastuupohjaista* (engl. responsibility driven) tapaa suunnitella oloita [kehittäjä Wirfs-Brock]
  - Vastuupohjainen oliosuunnittelu luonnollisesti noudattaa kaikkia yleisiä oliosuunnitteluperiaatteita, niitäkin kolmea jotka tällä kurssilla jo mainittu

Varoituksen sana

- Oliosuunnittelu on hyvin vaikea aihepiiri
- Erilaisia lähestymistapoja oliosuunnitteluun on monia
- Menetelmän käyttökelpoisuus riippuu sovelluskohteesta ja käyttäjästä
- *Mikään menetelmä (esim. vastuupohjainen oliosuunnittelu) ei sovellu kaikkiin tilanteisiin tai kaikille ihmisille*
- Eli seuraavassa (aiheeseen käytetään ainakin 2.5 luentoa) esiteltävä menetelmä on vain yksi tapa muiden joukossa
  - Vaikka kaikki eivät varmasti pidä Larmanin lähtökohtaa parhaana mahdollisena, uskoisin, että edes johonkin oliosuunnittelutapaan tutustuminen on aloittelijalle parempi vaihtoehto kuin pään pistäminen pensaaseen
  - Yhden menetelmän tuntevan on helpompi omaksua muita
- Koska asia on laaja, emme voi perehtyä siihen kovin syvällisesti ja jotain mutkia joudutaan oikomaan
  - Larmanin kirja on ehdottoman suositeltava jos aihepiiri kiinnostaa tarkemmin
  - Larmanin kirjakin on vain *johdanto* aihepiiriin...

Suurempi esimerkki: Kirjastojärjestelmä

- Tavoitteena on määritellä ja suunnitella tietojärjestelmä, jonka avulla hallitaan kirjaston lainaustapahtumia.
- Järjestelmä on alkuvaiheessa tarkoitettu ainoastaan yksittäisen kirjaston käyttöön
- Järjestelmä toteutetaan ketterien menetelmien hengessä eli *iteratiivisesti*
  - ensimmäisessä iteraatiossa toteutetaan perustoiminnallisuus, johon jokainen myöhempi iteraatio tuo lisää toiminnallisuutta
  - Järjestelmä saadaan käyttöön jo muutaman iteraation päästä
- Ensimmäisen tuotantoversion valmistumisen jälkeen mahdolliset myöhemmät iteraatiot voivat vielä laajentaa järjestelmän toiminnallisuutta asiakkaan haluamalla tavalla.
- *Luentomonisteessa käydään läpi ainoastaan ensimmäisen iteraation vaatimusmäärittely- ja suunnitteluvaihe*
  - Luennoilla aiheeseen käytössä noin 5 tuntia, eli kaikkea ei keritä käymään läpi
  - **Asiat ovat siis esitetty täsmällisemmin luentomonisteessa**

Asiakkaan asettamia vaatimuksia

- Kirjasto lainaa alussa vain kirjoja, myöhemmin ehkä muitakin tuotteita, kuten CD- ja DVD-levyjä
- Yksittäistä kirjanimikettä voi olla useampia kappaleita
- Kirjastoon hankitaan uusia kirjoja ja kuluneita tai hävinneitä kirjoja poistetaan
- Kirjastonhoitaja huolehtii lainojen, varausten ja palautusten kirjaamisesta
- Kirjastonhoitaja pystyy ylläpitämään tietoa lainaajista sekä nimikkeistä
- Nimikkeen voi varata jos yhtään kappaletta ei ole paikalla
- Varaus poistuu lainan yhteydessä tai erikseen peruttaessa
- Lainaajat voivat selata valikoimaa kirjastossa olevilla päätteillä
- Kirjauduttuaan omalla kirjastokortin numerolla, on lainaajan myös mahdollista selailla omia lainojaan
- Järjestelmän on oltava laajennettavissa usean kirjaston laajuiseksi ja mahdollistettava asiakkaiden käytössä olevat lainaus- ja palautusautomaatit

Käsiteanalyysi - sanasto

- Tuttuun tapaan alamme etsiä tekstikuvauksesta (joka on tällä kertaa lista vaatimuksia) luokkakandidaatteja
- Kaikki luokkakandidaatiksi valitut termit eivät ole merkitykseltään aivan itsestäänselviä
- Usein on hyödyllistä laatia *sanasto*, jossa käsitteiden merkityksiä tarkennetaan
- Seuraavassa osa kirjastojärjestelmän sanastosta

- **Nimike**
  - Samaa kirjaa voi kirjastossa olla useita kappaleita. Nimikkeellä tarkoitetaan tietoja yhden nimisestä kirjasta
  - Eli esim. varaus kohdistuu tiettyyn nimikkeeseen, josta lainaaja saa itselleen tietyn kirjan
- **Kirja**
  - Hyllyssä sijaitseva "fyysinen" kirja, jonka asiakas lainaa. Jokaisella yksittäisellä kirjalla on todennäköisesti yksikäsitteinen tunniste, joka erottaa sen muista saman nimikkeen kirjoista

Sanasto jatkuu

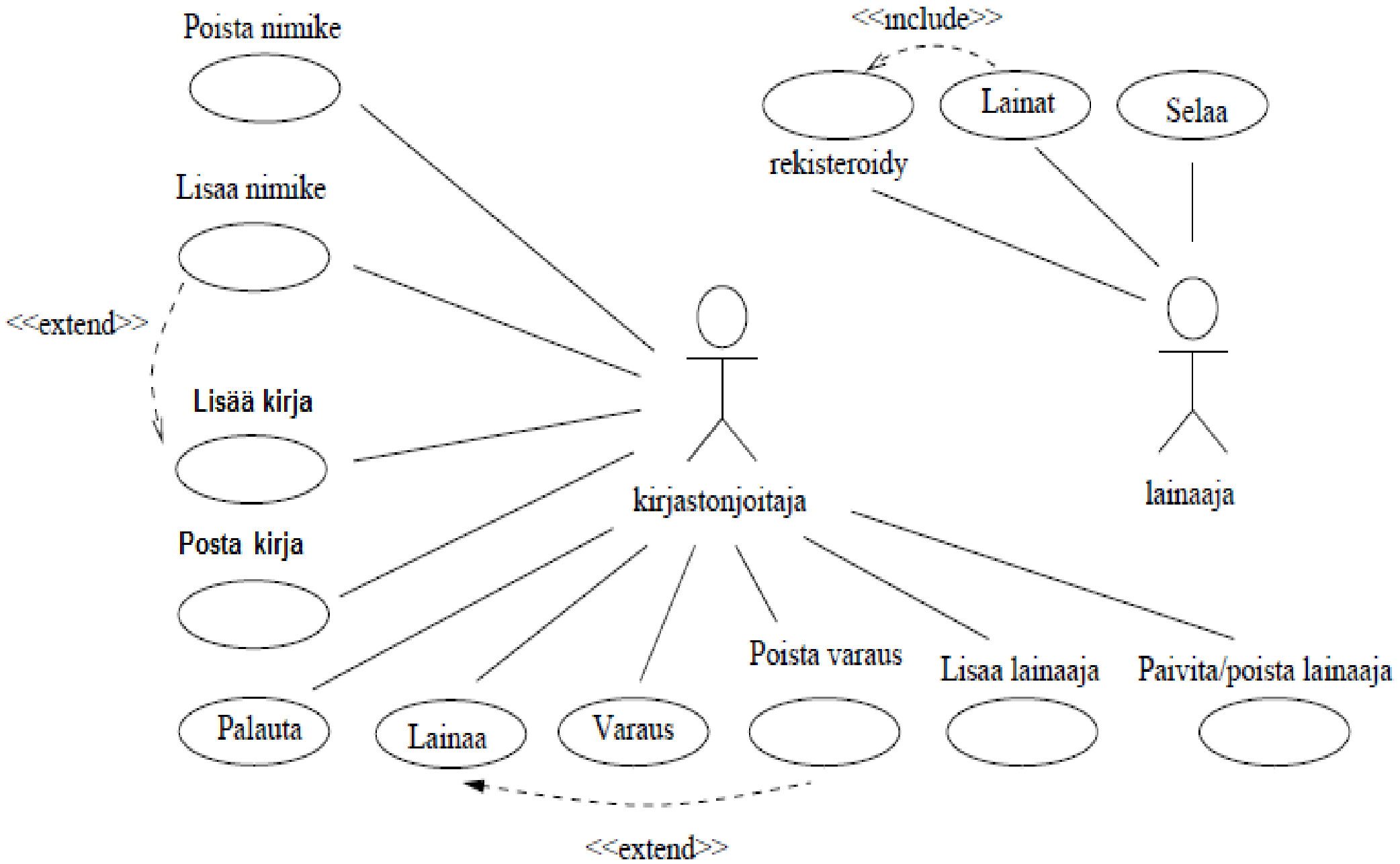
- **Kirjastonhoitaja**
  - Järjestelmän käyttäjä, tekee lainat, varaukset ja palautukset
- **Laina**
  - Lainaus kohdistuu tiettyyn fyysiseen kirjaan
- **Varaus**
  - Tiettyyn nimikkeeseen kohdistuva varaus
- **Lainaaja**
  - Kirjaston asiakas, hoitaa lainauksen kirjastohoitajan kautta, mutta voi käyttää joitakin järjestelmän palveluja päätteen tai automaatin avulla.
- **Pääte**
  - Mahdollistaa asiakkaalle nimikekokoelman selailun ja rekisteröityneille asiakkaille omien lainojen selailun
- **Automaatti**
  - Tulevaisuudessa asiakas voi suorittaa lainauksen ja palautuksen suoraan automaatin avulla

- Vaikka sanasto sisältääkin käytännössä kaikki luokkakandidaatit, ennen luokkamallin tekemistä määrittelemme järjestelmän käyttötapaukset

Käyttötapaukset

- Järjestelmän *käyttäjät* ovat selvästi *kirjastonhoitaja* ja *lainaaja*
- Järjestelmän käyttötapauksiksi tunnistetaan seuraavat
  - Lainaa kirja
  - Palauta kirja
  - Varaa nimike
  - Poista varaus
  - Lisää nimike
  - Päivitä/poista nimike
  - Lisää kirja
  - Poista kirja
  - Lisää lainaaja
  - Päivitä/poista lainaajatiedot
  - Selaa valikoimaa
  - Rekisteröidy
- Käyttötapauskaavio seuraavalla sivulla

Käyttötapauskaavio



Vaatimusmäärittely, ensimmäinen iteraatio

- Valitaan ensimmäisessä iteraatiossa toteutettavaksi seuraavat käyttötapaukset:
  - Lainaa kirja
  - Palauta kirja
  - Lisää nimike
  - Lisää kirja
  - Lisää lainaaja
- Ensimmäiseen iteraatioon kannattaa yleensä valita toteutettavaksi järjestelmän ydintoiminnallisuus
  - Näin asiakas näkee jo varhaisessa vaiheessa eteneekö järjestelmän kehitys oikeaan suuntaan
- Seuraavaksi tarkennetaan iteraatioon valitut käyttötapaukset
  - yhden iteraation aikana ei ole välttämätöntä toteuttaa käyttötapauksien toimintaa täydessä laajuudessaan
  - Seuraavassa määritellä käyttötapauksesta perustoiminnallisuus, jota myöhemmissä iteraatioissa todennäköisesti tarkennetaan ja laajennetaan

Käyttötapaus 1: Lainaa kirja

Tavoite: Lainaaja tulee lainattavan kirjan kanssa tiskille ja antaa kirjastokortin ja kirjan virkailijalle, joka kirjaa lainan kirjastojärjestelmän avulla.

Käyttötapauksen kulku:

1. Syötetään lainaajan tunniste eli kirjastokortin numero
2. Järjestelmä tunnistaa lainaajan ja tulostaa lainaajan tiedot
3. Syötetään lainattavan kirjan koodi
4. Järjestelmä tunnistaa kirjan ja tulostaa kirjan tiedot
5. Pyydetään järjestelmää rekisteröimään laina
6. Järjestelmä kertoo lainan eräpäivän

Poikkeuksellinen toiminta:

Lainaaja voi olla lainauskiellossa tai kirjan lainaus estetty, tällöin lainaa ei rekisteröidä. Nämä erikoistapaukset eivät kuitenkaan sisälly vielä tämän iteraation aikana toteutettavaan toiminnallisuuteen.

Käyttötapaus 2: Palauta kirja

Tavoite: Lainaaja tuo lainassa olleen kirjan virkailijalle, virkailija kirjaa palautuksen järjestelmään.

Käyttötapauksen kulku:

1. Syötetään palautettavan kirjan koodi
2. Järjestelmä tunnistaa palautettavan kirjan ja tulostaa sen tiedot
3. Merkitään kirja palautetuksi

Käyttötapaus 3: Lisää nimike

Tavoite: Kokoelmaan lisätään uuden nimikkeen tiedot. Tässä käyttötapauksessa ei kirjata yksittäisten kirjan tietoja, vaan ainoastaan kirjaa koskevat nimiketiedot esim. tekijät, nimi, ISBN-numero, aiheluokitus, kustantaja, painovuosi, painos.

Käyttötapauksen kulku:

1. Pyydetään luomaan uusin nimike annetuilla tiedoilla
2. Järjestelmä tulostaa luodun nimikkeen tiedot

Poikkeuksellinen toiminta:

Vastaava nimike voi jo olla järjestelmässä. Tässä tapauksessa ei sallita nimikkeen luomista uudelleen.

Käyttötapaus 4: Lisää kirja

Tavoite: Luodaan uudelle lainattavissa olevalle kirjalle yksikäsitteinen tunniste ja talletetaan tiedot järjestelmään. Tämä edellyttää, että vastaavan nimikkeen tiedot löytyvät jo järjestelmässä.

Käyttötapauksen kulku:

1. Syötetään kirjan nimi, kirjoittaja ja ISBN-koodi
2. Järjestelmä tunnistaa kirjaa vastaavan nimikkeen ja tulostaa nimikkeen tiedot
3. Pyydetään uudelle kirjalle yksikäsitteinen tunniste
4. Talletetaan uusi kirja järjestelmään

Poikkeuksellinen toiminta:

Jos vastaavaa nimikettä ei ole järjestelmässä, suoritetaan käyttötapaus *Lisää nimike*. Käyttötapauskaaviossa käytetty on käytetty extend-merkintää, eli Lisää kirja -käyttötapauksen yhteydessä suoritetaan tarvittaessa Lisää nimike -käyttötapaus.

Käyttötapaus 5: Lisää lainaaja

Tavoite: Uuden lainaajan nimi ja osoite kirjataan järjestelmään.

Käyttötapauksen kulku:

1. Kirjataan järjestelmään uuden lainaajan tiedot
2. Järjestelmä palauttaa uuden lainaajan tiedot, erityisesti lainaajanumeron, joka toimii lainaajan yksikäsitteisenä tunnisteena

Poikkeuksellinen toiminta:

Jos vastaavaa lainaaja jo on olemassa, ei uutta lainaajaa luoda.

Muut vaatimukset

- Käyttötapaukset ovat yksi tapa kuvata järjestelmän *toiminnallisia vaatimuksia* (engl. functional reguirements)
- Kuten luennolla 1 mainittiin, on ohjelmistolla aina myös ei-toiminnallisia (engl. non functional requierements) vaatimuksia, kuten esim.
  - Käytettävyyteen liittyvät vaatimukset, esim:
    - Helppo käyttää myös vasenkätisille
  - Suorituskykyyn liittyvät vaatimukset, esim:
    - Kykenee tallettamaan ainakin 10000 kirjan tiedot
    - Selviää viidestä yhtäaikaisesta lainaustapahtumasta
  - Toteutusympäristöön liittyvät vaatimukset, esim:
    - Ohjelmointi Javalla, tietokantana MySQL
- Lisäksi järjestelmään voi liittyä toiminnallisuutta, joka ei liity erityisesti mihinkään käyttötapaukseen
  - Esim. vaatimus, että kaikki tapahtumat tallennetaan lokiin

Sovelluksen kohdealueen luokkamalli

- Teemme luokkamallin joka kattaa järjestelmän ensimmäisen iteraation aikana toteutettavan toiminnallisuuden
- Tutkimalla muutaman sivun takaista sanastoa sekä viittä ensimmäistä käyttötapausta, päädymme seuraaviin luokkiin:
  - Lainaaja
  - Kirja
  - Laina
  - Nimike
  - Järjestelmä
- Seuraavalla sivulla oleva luokkamalli perustuu seuraaviin havaintoihin:
  - Järjestelmä pitää kirjaa sekä lainaajista että lainattavissa olevista nimikkeistä. Yksittäistä nimikettä kohti on useita kirjoja
  - Yhdellä lainaajalla saattaa olla yhtä aikaa useita lainoja
  - Laina kohdistuu aina tiettyyn kirjaan
  - Kirja ei välttämättä ole lainassa, mutta jos kirja on lainassa, liittyy se kerrallaan vain yhteen lainaan ja on vain yhdellä lainaajalla kerrallaan.

Järjestelmän kohdealueen luokkamalli, ensimmäinen iteraatio

- Kohdealueen luokkamalli on siis vielä täysin toteutusriippumaton malli, jonka tarkoitus on jäsentää sovellusalueen käsitteistöä ja käsitteiden suhteita.
- Tässä vaiheessa ei kannata vielä yrittääkään miettiä mitä operaatioita luokilla on
- Luokat saavat operaatiot kohta alkavassa oliosuunnitteluvaiheessa