Ohjelmistojen mallintaminen

Luento 10, 3.12.

Kertaus

- **Menetelmä**: miten edetään ohjelmistoprosessin eri vaiheissa ja mitä apuvälineitä kannattaa missäkin tilanteessa käyttää
- Käymme läpi *erästä* olioperustaista menetelmää *kirjastoesimerkin* avulla
- Aluksi vaatimusmäärittely
  - Käyttötapaukset ja kohdealueen luokkamalli
- Sen jälkeen ohjelmiston suunnittelu
  - Arkkitehtuurisuunnittelu
  - Oliosuunnittelu
- Iteratiivinen lähestymistapa: eli määritellään, suunnitellaan ja toteutetaan vähän kerrallaan
- Viimeksi ohjelmiston **arkkitehtuuri**
  - Järjestelmän jakautuminen isompiin rakennekomponentteihin
  - Arkkitehtuuri voidaan kuvata esim. UML:n *pakkauskaaviolla*
  - Kirjastojärjestemä noudattaa *kerrosarkkitehtuuria*
  - Järkevää eristää käyttöliittymä sovelluslogiikasta
- Nyt aiheena se kaikkein haastavin eli **oliosuunnittelu**

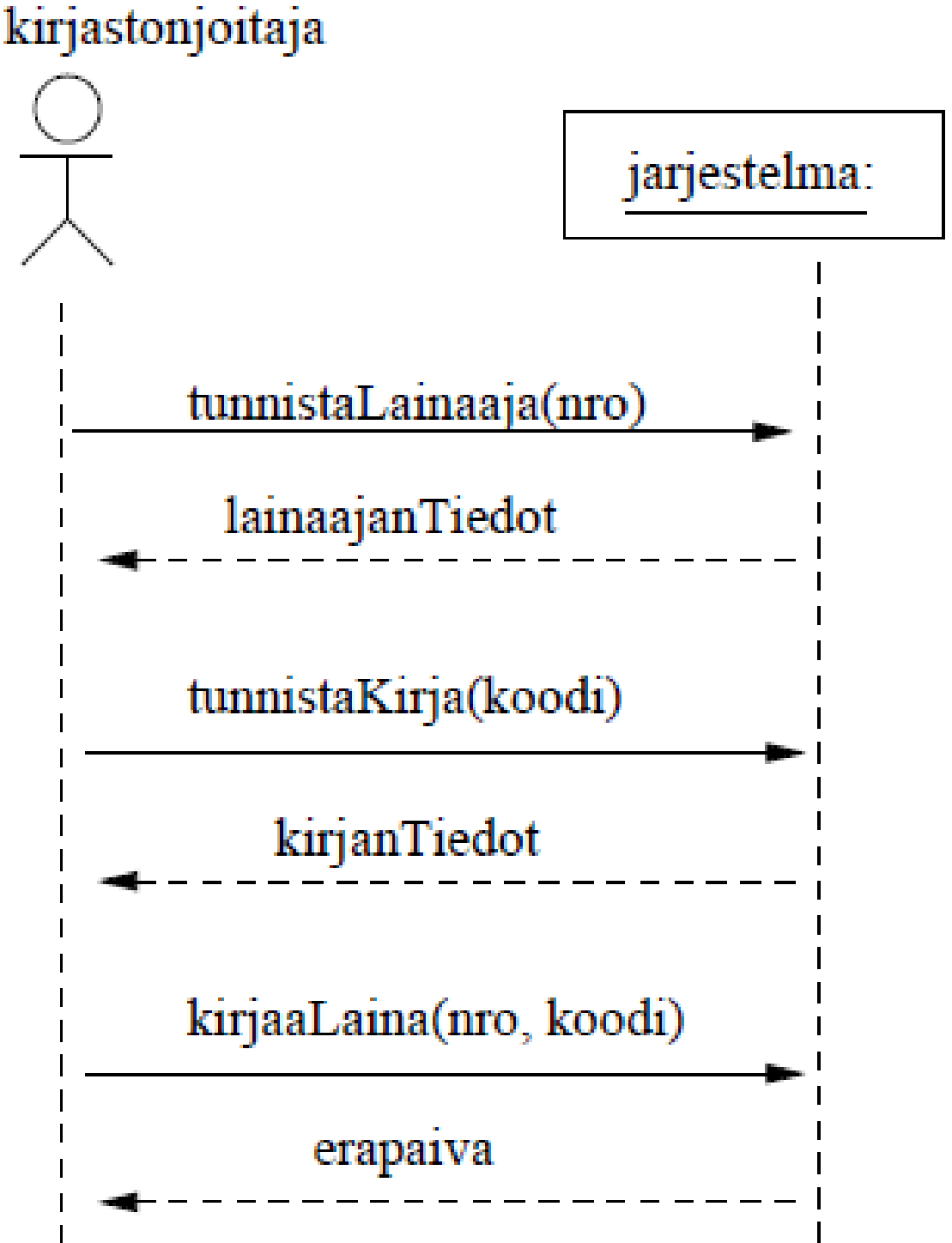
Missä olimme menossa kirjastoesimerkissä

- Ennen oliosuunnittelua, palautetaan mieleen mitä ehdimme jo tehdä kirjastoesimerkin suhteen
- Määrittelimme järjestelmän *ensimmäisessä iteraatiossa* toteutettavan toiminnallisuuden *käyttötapauksina*
  - Lainaa kirja
  - Palauta kirja
  - Lisää lainaaja
  - Lisää kirja
  - Lisää nimike
- Täsmennettiin käyttötapausten kulku *järjestelmätason sekvenssikaavioiden* avulla
  - Näistä käy hyvin selville, mitä operaatiota järjestelmän on toteutettava toimiakseen käyttötapausten määrittelemällä tavalla
- Tehtiin sovelluksen *kohdealueen luokkamalli* ensimmäisen iteraation osalta
  - Käsitteet ja niiden suhteet
  - Pohja suunnittelutason luokkamallille

Lainaa kirja – käyttötapauksen kulku ja sekvenssikaavio

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

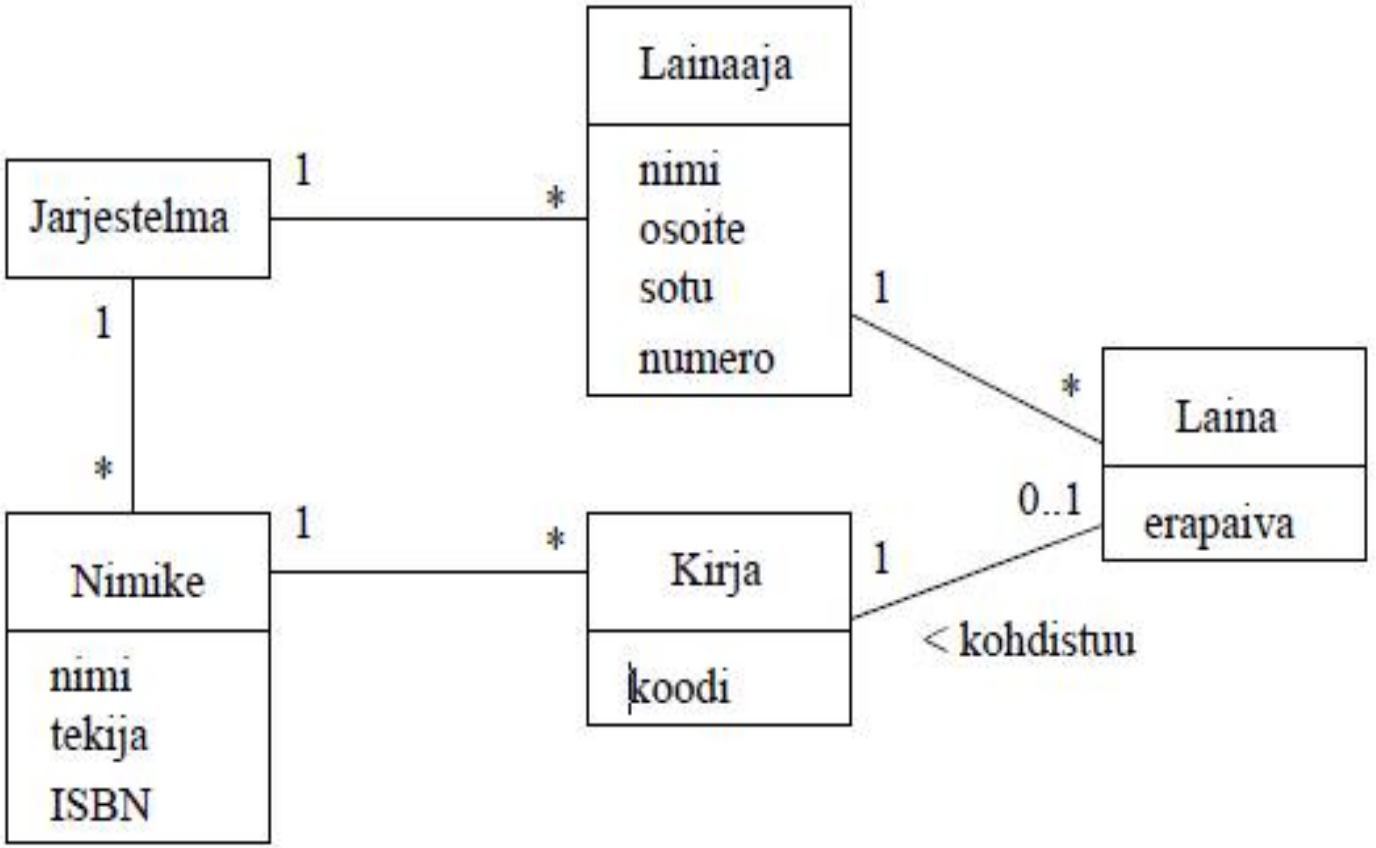


Operaatiot, jotka järjestelmän on toteutettava ensimmäisessä iteraatiossa

- Edellisellä sivulla käyttötapauksen *Lainaa kirja* kulku tekstinä ja sekvenssikaavion avulla ilmaistuna
- Käymällä läpi kaikkia käyttötapauksia kuvaavat sekvenssikaaviot saadaan *lista operaatiosta, jotka järjestelmän pitää toteuttaa*:
  - tunnistaLainaaja(nro)
  - tunnistaKirja(koodi)
  - kirjaaLaina(nro, koodi)
  - merkitsePalautus(koodi)
  - lisaaLainaaja(nimi, os, sotu)
  - lisaaNimike(nimi, kirj, ISBN)
  - tunnistaNimike(nimi, kirj, ISBN)
  - luoTunniste(nimi, kirj, ISBN)
  - lisaaKirja(nimi, kirj, ISBN, tunniste)
- Seuraavalla sivulla vielä järjestelmän kohdealueen luokkamalli

Kohdealueen luokkamalli, ensimmäinen iteraatio

- Kohdealueen luokkamalli on siis vielä täysin toteutusriippumaton malli, jonka tarkoitus on jäsentää sovellusalueen käsitteistöä ja käsitteiden suhteita
- Kohdealueen luokkamallin olioille ei laiteta vielä operaatiota
- Luokat saavat operaatiot oliosuunnitteluvaiheessa, jonka nyt aloitamme



Oliosuunnittelu

- Ohjelman on siis toteutettava käyttötapauksista johdetut operaatiot toimiakseen vaatimustensa mukaisesti
- Ohjelmalta vaadittavien operaatioiden voidaan ajatella olevan *ohjelman vastuita* (engl. responsibilities).
  - hoitamalla vastuunsa ohjelma toimii kuten sen odotetaan toimivan
- Ohjelma toteuttaa vastuunsa olioiden yhteistyön avulla
  - Haasteena oliosuunnittelussa siis on löytää sopivat oliot, joille ohjelman vastuut jaetaan
- Tyypillisesti mikään yksittäisen olio ei toteuta yhtä ohjelman vastuuta itse, vaan *jakaa vastuun pienemmiksi alivastuiksi ja delegoi alivastuiden hoitamisen muille olioille*
- Yritetään seuraavassa hahmotella muutamaa periaatetta, jotka auttavat ohjelman vastuut toteuttavien olioiden löytämisessä
  - Periaatteet noudattavat Larmanin kirjan tapaa soveltaa *vastuupohjaista oliosuunnittelua*
  - Vastuupohjaisuus (engl. responsibility driven design) ei ole Larmanin keksintö, takana mm. Wirfs-Brock, Beck ja Cunningham

Vastuupohjaisen oliosuunnittelun periaatteita

- Tehdään sama huomautus kuin viimeksi, eli vaikka esittelemme erään menetelmän oliosuunnitteluun, niin
  - mikään yksittäinen menetelmä ei toimi kaikenlaisiin ongelmiin vaan antaa ainoastaan virikkeitä alkuun pääsemiseen
  - oliosuunnittelussa tarvitaan aina luovuutta
- Suunnittelumenetelmässä on muutamia periaatteita, joita pyritään pitämään mielessä suunnittelun edetessä
  - Kirjassaan Larman listaa 9 periaatetta
  - Larman käyttää periaatteistaan nimeä *general responsibility assignment software patterns* eli *GRASP-patterns*
  - me käsittelemme niistä ainoastaan kuutta ensimmäistä
- Periaatteiden (paitsi yhden) nimet kankeana suomennoksina ovat:
  - Ohjausperiaate
  - Eksperttiperiaate
  - Luontiperiaate
  - Riippuvuusten vähäisyyden periaate
  - Toiminnallisen yhtenäisyyden periaate

Vastuupohjaisen oliosuunnittelun periaatteita

- Ensimmäinen periaate liittyykin siihen, *kuka ottaa vastaan käyttöliittymäolioilta* tulevat komennot
  - Esim. kun kirjastokortin numero syötetään käyttöliittymään, miten numero välitetään sovelluslogiikalle?
  - Vastauksen tuo **ohjausperiaate**, jonka sisältö tarkentuu pian
- Vastuita on pääpiirteittäin kahdenlaisia, tietämistä (engl. knowing) ja tekemistä (engl. doing) koskevia
  - *Tietämistä* edustaa esim. vastuu *tunnistaLainaaja(nro)* eli tunnistetaan lainaaja kirjastokortin numeron perusteella
  - *Tekemistä* edustaa esim. vastuu *lisaaLainaaja(nimi,osoite,sotu),* joka siis lisää järjestelmään uuden lainaajan tiedot
- **Ekspertti-periaatteen** mukaan *vastuun hoitaminen kannattaa antaa sille oliolle, jolla on parhaat tiedot vastuun suorittamiseksi*
  - Tämä saattaa kuulostaa tässä vaiheessa typerän itsestäänselvältä...
- Usein mikään olio ei yksistään osaa hoitaa koko vastuuta
  - **Yksittäinen vastuu pitää jakaa osavastuihin, jotka kokonais-vastuun omaava olio delegoi osavastuiden eksperteille**

- Kaikki oliot täytyy luoda. **Luontiperiaatteen** mukaan olion luo se, joka
  - sisältää tai säilyttää olion
  - pitää kirjaa oliosta tai
  - tuntee olion alustustuksessa tarvittavan datan
- Oliolla saattaakin näiden periaatteiden nojalla olla useita luojakandidaatteja
- Eri ratkaisuista pitää valita tarkoituksenmukaisin vertailemalla niitä esim. kahden seuraavaksi esitettävän periaatteen kriteerein
- Kaikissa suunnitteluratkaisuissa tulee pääsääntöisesti pyrkiä olioiden välisten **riippuvuuksien vähäisyyteen** (engl. low coupling)
  - Järjestelmän erilaisten komponenttien välisten riippuvuuksien minimointia perusteltiin jo kerrosarkkitehtuurin yhteydessä
  - Erityisesti riippuvuuksia konkreettisiin luokkiin kannattaa välttää (aiemmin mainittu oliosuunnittelun periaate program to interfaces, not to concrete implementations sanoo juuri tämän)
- Pelkkä riippuvuuksien minimointi ei tuota kaikin osin hyvää ratkaisua
- Oliosuunnittelussa tulee pyrkiä myös **toiminnallisuudeltaan yhtenäisiin** (engl. high cohesion) olioihin
  - Olion julkisten metodien tulee liittyä mahdollisimman saman asian tekemiseen

Oliosuunnittelun kulmakivet: riippuvuuksien vähäisyys ja toiminnallinen yhtenäisyys

- Toiminnallisuudeltaan epäyhtenäinen olio olisi esim. vastuussa sekä lainaajien lisäämisestä että lainojen kirjaamisesta järjestelmään
  - Koska lainaajat ja lainat ovat kaksi selkeästi erillistä konseptia, on parempi, että sama olio ei hoida molempiin liittyviä vastuita
- Toiminnallisessa yhtenäisyydessä kyse asiasta, joka mainittu kurssilla jo useasti:
  - oliosta kannattaa tehdä pieniä ja selkeitä, hyvin yhden asian osaavia oliota
  - kyseessä käytännössä (melkein sama asia kuin) **single responsibility** -periaate
- Kaikkia suunnitteluratkaisuja tulee punnita olioiden välisten riippuvuuksien ja olioiden toiminnallisen yhtenäisyyden kannalta ja pyrkiä näiden periaatteiden kannalta mahdollisimman hyviin ratkaisuihin
  - Nämä kaksi periaatetta ilmenevät lähes kaikkien olioasiantuntijoiden kirjoituksista joko suoraan tai epäsuorasti
  - Englanniksi periaatteet siis ovat nimeltään **low coupling** ja **high cohesion**, huomattavasti ytimekkäämpiä nimiä kun käännökset
- Termin toiminnallinen yhtenäisyys (high cohesion) sijasta käytämme jatkossa meille jo tutumpaa termiä **single responsibility**

- Osa suunnittelutason luokista saadaan vaatimusmäärittelyn aikana tehdyn kohdealueen luokkamallin pohjalta
  - todennäköisesti kirjastojärjestelmään otetaan suunnittelutasolle mukaan ainakin luokat lainaaja, laina, kirja ja nimike
- Jos suunnittelutasolle ei oteta muita luokkia kuin kohdealueen luokkamallista omaksutut, joudutaan helposti huonoihin ratkaisuihin
  - Seurauksena olioiden sekavat vastuut ja liialliset riippuvuudet ja näinollen mm. single responsibility -periaate rikkoutuu
- Tällaisissa tilanteissa otetaan mukaan uusia, ”keinotekoisia” luokkia, joita vastaavia käsitteitä ei välttämättä sovellusalueella ole
  - Tämä onkin **kuudes periaate**, nimeä periaatteelle on vaikea antaa
- Uudet mukaan tuotavat luokat ovat usein *teknisen tason luokkia*, joilla voi olla monia erilaisia käyttötarkoituksia, esim.
  - Käyttöliittymän toteutus
  - Tietokantayhteyksien hoitaminen
  - Sovellusolioiden yhteyksien toteuttaminen
  - Osajärjestelmien piilottaminen (fasadioliot)
  - Sovelluksen toiminnan ohjaus ja koordinointi
  - Abstraktit yliluokat ja rajapinnat

Oliosuunnittelun vaikeus

- Todettakoon edelliseen kalvoon liittyen, että ”kuudennen periaatteen” soveltaminen, eli sopivien teknisten apuluokkien keksiminen on äärimmäisen haastavaa
  - Kokemus, luovuus ja tieto auttavat
- Monissa *suunnittelumalleista* (engl. design patterns) on kyse juuri sopivien teknisten ratkaisujen ja abstraktioiden mukaantuomisesta
  - Luennoilla ja materiaalissa olemme törmänneet jo muutamaan suunnittelumalliin
- Valitettavasti suunnittelumalleja ja muuta haasteellista ja mielenkiintoisia oliosuunnitteluun liittyvää ei tällä kurssilla juuri ehditä käsittelemään
  - Syksyn 2011 kurssilla Ohjelmistokehitys tutustutaan asiaan tarkemmin
  - Asiasta kiinnostunut voi jo ennen kurssia suunnata kirjallisuuden ääreen, Larmanin lisäksi suositeltavia mm:
    - Robert Martin: Agile and iterative development
    - Freeman et. all.:Head first design patterns
    - Gamma et all.: Design patterns

Suunnittelun eteneminen: käyttötapaus kerrallaan

- Yksi tapa tehdä suunnittelua on edetä käyttötapauksittain
  - Otetaan yksi käyttötapaus kerrallaan tarkasteluun
  - Suunnitellaan oliot tai mukautetaan jo suunniteltujen olioiden vastuita ja yhteistyötä siten, että tarkastelussa olevan käyttötapauksen tarvitsemat operaatiot saadaan toteutetuksi
  - Usein käy niin, että uuden toiminnallisuuden lisääminen aiheuttaa jo olemassa olevaan suunnitelmaan pieniä muutoksia

- Aloitamme nyt kirjastojärjestelmän oliosuunnittelun käyttötapauksesta **Lainaa kirja**
- Monisteessa käsitelty kaikki ensimmäisen iteraation käyttötapaukset

- Samalla kun etenemme oliosuunnittelussa näemme miten edellisillä sivuilla olevia suunnitteluperiaatteita sovelletaan käytännössä

- Kertauksena vielä käyttötapauksen **Lainaa kirja** 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

- **Operaatiot**, jotka käyttötapauksen suorittamisessa tarvitaan:
    - **tunnistaLainaaja(nro)**
        - Kirjastokortissa olevaa lainaajanumeroa vastaavan lainaajan tiedot tulostetaan
    - **tunnistaKirja(koodi)**
        - Tunnistetaan yksittäinen kirja ja tulostetaan sen tiedot
    - **kirjaaLaina(nro, koodi)**
        - Luodaan Laina-olio, joka kiitetään lainaajaan, jonka kirjastokortin numero nro ja kirjaan, jolla tunnisteena koodi

Alustavat luokat suunnittelutason luokkamalliin

- Kuten jo mainittu, useimmiten *sovelluksen kohdealueen luokkamallin luokista ainakin osa siirtyy myös suunnittelutasolle*
- Käyttötapaus *Lainaa kirja* liittyy kohdealueen luokista ainakin seuraaviin:
  - Lainaaja, Kirja, Laina
- Jokainen Kirja-olio liittyy tiettyyn Nimike-olioon
  - Myös Nimike otetaan mukaan
- Eli *päädymme alustavasti ottamaan kohdealueen luokkamallin suoraan suunnittelun pohjaksi*

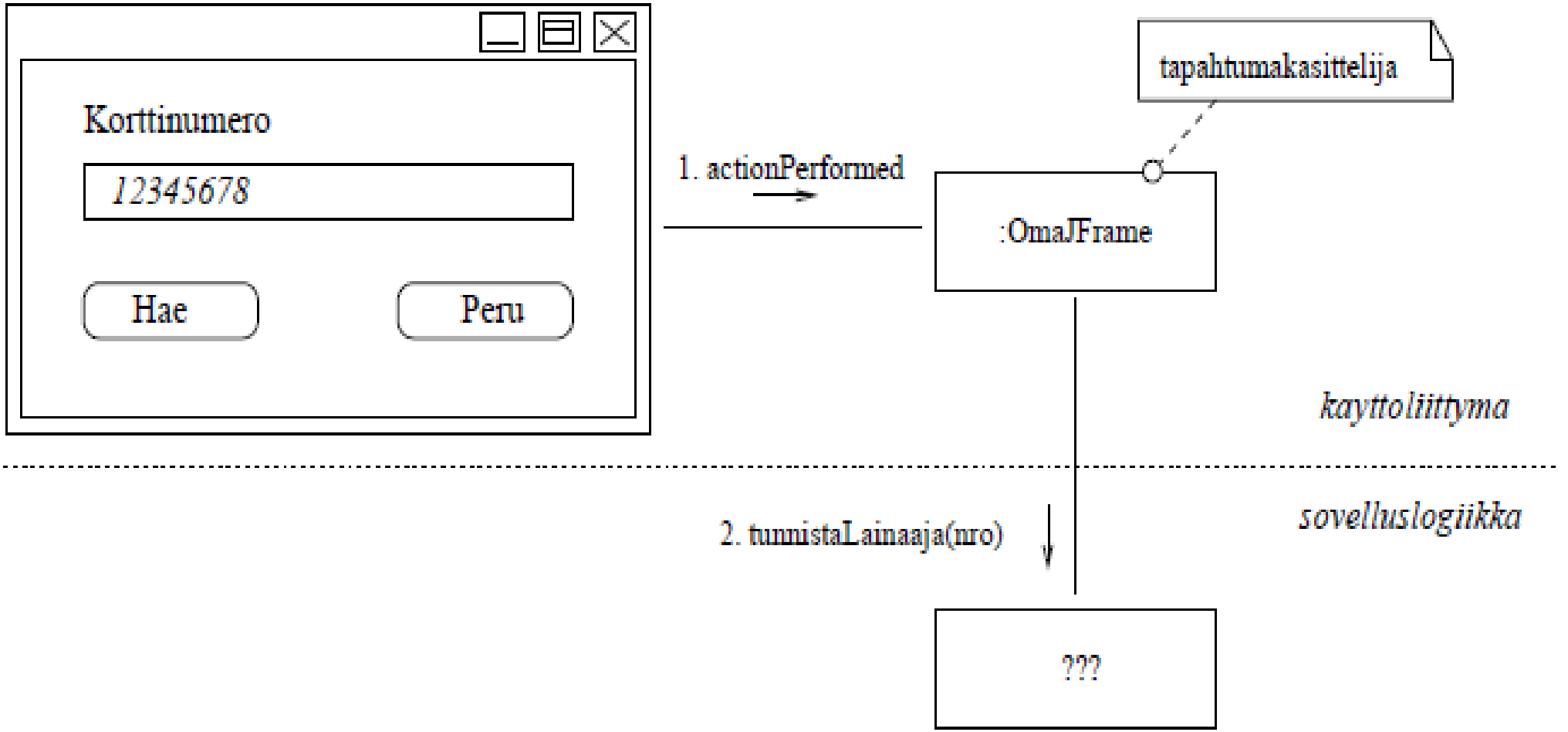
- Koko käyttötapauksen *Lainaa kirja* toiminnallisuuden toteuttaminen vaatii siis kolmen operaation toteuttamista
  - Aloitamme operaatiosta tunnistaLainaaja(nro)
  - On siis mietittävä *minkä luokan tai luokkien vastuulle operaation tunnistaLainaaja(nro) suoritusvastuu annetaan*
- Ensin kuitenkin tarkastellaan käyttöliittymän ja sovelluslogiikan yhteyttä

Käyttöliittymän ja sovelluslogiikan erottaminen

- Käyttöliittymä ja sovelluslogiikka on siis hyvä erottaa, eli
  - **Käyttöliittymän toteuttaviin olioihin ei sisällytetä ollenkaan ohjelman sovelluslogiikkaa**
  - Käyttötapaukset toteuttavat operaatiot (mm. tunnistaLainaaja, tunnistaKirja, lisaaLaina) suoritetaan kokonaisuudessaan sovelluslogiikan puolella
  - Käyttöliittymä ainoastaan kutsuu näitä operaatioita kun käyttäjä suorittaa käyttötapaukseen liittyvää toiminnallisuutta
- Periaate ilmenee seuraavan sivun kuvasta
  - Käyttötapauksen Lainaa kirja alussa tapahtuu lainaajan tunnistus, eli virkailija syöttää dialogi-ikkunaan kirjastokorttinumeron, jotta lainaaja voidaan tunnistaa
  - Oletetaan, että käyttöliittymä on toteutettu Javan Swing-komponenteilla
  - Painikkeen klikkaaminen tuottaa herätteen (action performed) ohjelmoijan toteuttamalle tapahtumakäsittelijäoliolle
  - Tapahtumakäsittelijä (joka siis on osa käyttöliittymää) kutsuu *jonkun* sovelluslogiikan olion toteuttamaa metodia tunnistaLainaaja(nro)

Käyttöliittymän ja sovelluslogiikan yhteys

- **Minkä olion tulisi ottaa vastuulleen käyttöliittymän kutsuman operaation suoritus**, eli mikä on kuvan kysymysmerkein nimetty luokka?
- Tähän vastauksen tuo muutama sivu sitten mainittu **ohjausperiaate**



Ohjausperiaate

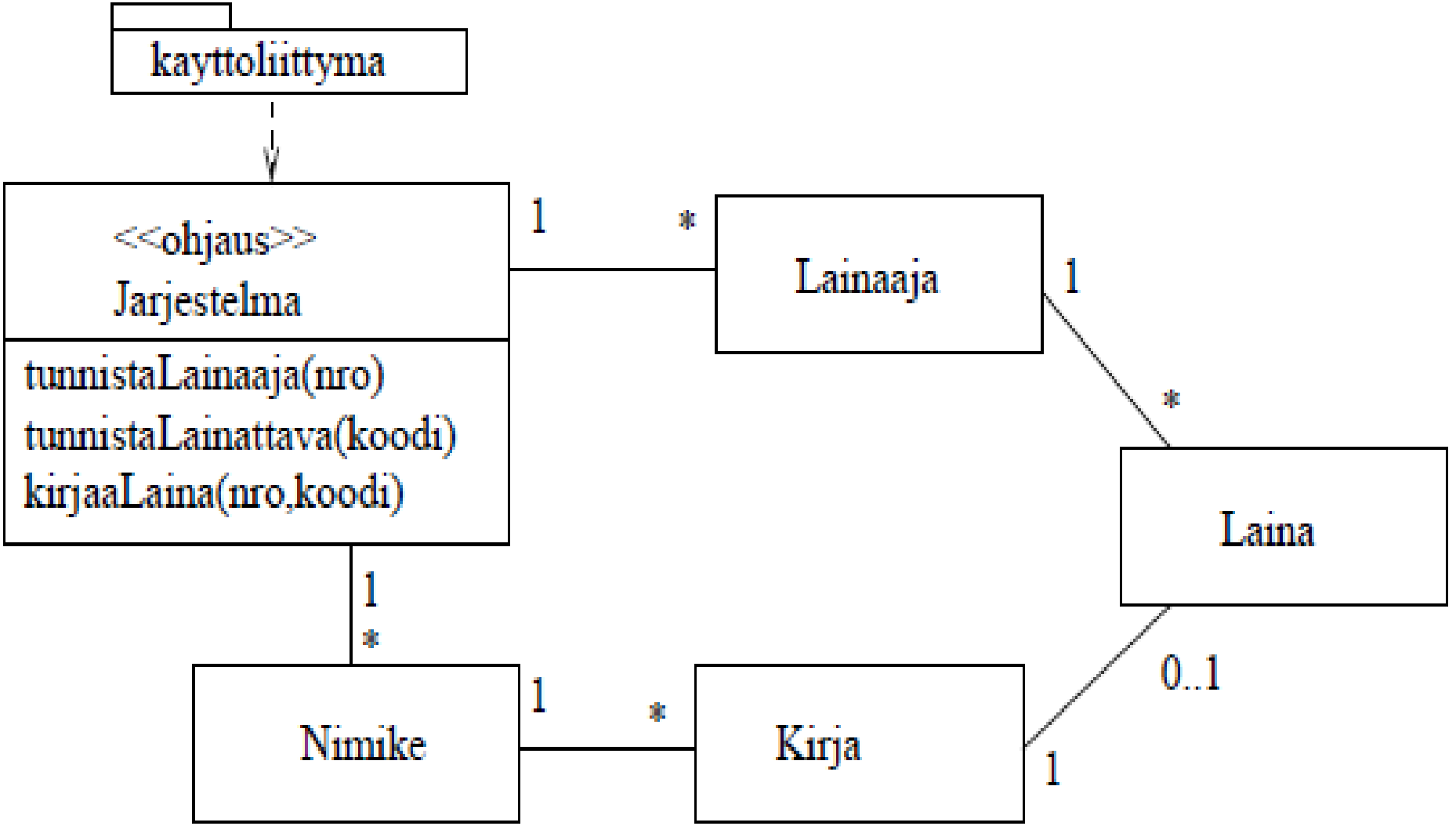
- Periaatteena on asettaa käyttöliittymän ja varsinaisten sovellusolioiden väliin **ohjausolio**, joka
  - ottaa vastaan käyttöliittymästä tulevan operaatiokutsun ja kutsuu edelleen sovelluslogiikan olioita, jotka suorittavat varsinaisen tehtävän
- Järjestelmätason sekvenssikaavioista ilmenneet operaatiot siis tulevat ohjausolioiden operaatioiksi joita käyttöliittymä kutsuu
- Ohjausolio voi olla ohjelman **kaikkien operaatioiden yhteinen** ohjausolio tai vaihtoehtoisesti voidaan käyttää **käyttötapauskohtaisia** ohjausolioita
  - Välimuodotkin ovat mahdollisia eli joillain käyttötapauksella voi olla oma ohjausolio ja jotkut taas käyttävät yhteistä ohjausolioa
  - Yksinkertaisissa järjestelmissä selvitään ehkä yhdellä ohjausoliolla
  - Jos järjestelmässä on paljon toiminnallisuutta, kannattanee kullakin käyttötapauksella olla oma ohjausolionsa
  - Näin ohjausoliot säilyvät vastuiltaan selkeämpinä ja helpommin ylläpidettävinä
- Käyttötapauskohtaisten ohjausolioiden etuna on myöskin se, että niiden avulla voidaan helposti hallita monimutkaisen käyttötapauksen etenemistä

Ohjausolion valinta

- *Yksinkertaisessa ohjelmassa* yhteisenä *ohjausoliona* voi joissain tapauksissa toimia *sovellusalueen olio, joka vastaa koko järjestelmää*
- Päädymme kirjastojärjestelmässä *alustavasti* kaikille käyttötapauksille *yhteisen ohjausolion* käyttöön
  - ohjausolioksi valitsemme koko järjestelmää vastaavan olion, jonka nimi on Jarjestelma
- Seuraavan sivun luokkakaavio dokumentoi tilanteen
  - Selvyyden vuoksi ohjausluokan rooli on merkitty kuvaan stereotyypin eli tarkenteen «ohjaus» avulla
- Käyttöliittymä on näytetty ainoastaan pakkauksena, joka on riippuvainen ohjausluokasta
- Kuvassa on myös mukana myös muut sovellusalueen luokkamallista omaksutut alustavat luokkaehdokkaat
  - Luokkien yhteyksien laadut tulevat tarkentumaan suunnittelun edetessä
- Ensimmäisessä iteraatiossa ei vielä oteta kantaa olioiden tietokantaan tallettamiseen, eli tallennuspalvelupakkausta ei ole kuvaan merkitty

Suunnitteluvaiheen luokkamallin ensimmäinen versio

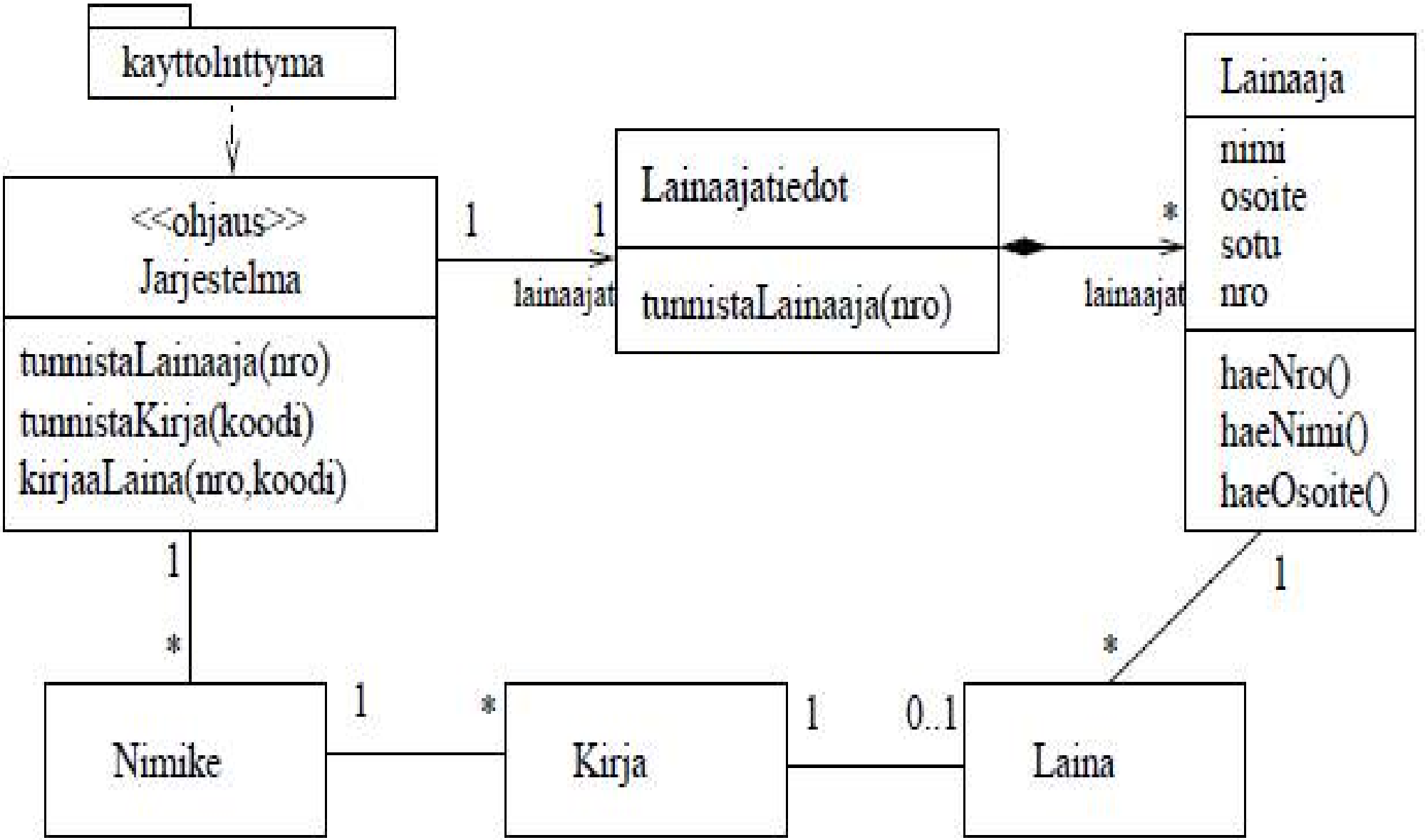
- Luokka järjestelmä toimii ohjausoliona



Operaation tunnistaLainaaja(nro) suunnittelu

- Aloitetaan jakamaan operaatioiden suoritusvastuita luokille samalla tarkentaen tarpeen mukaan ohjelman luokkarakennetta
- Koska Jarjestelma-olio tuntee Lainaaja-oliot, voisi se **eksperttiperiaatteen** mukaan ottaa vastuulleen operaation tunnistaLainaaja(nro) suorittamisen
- Periaatteen mukaanhan vastuu tulee antaa oliolle, jolla on parhaat tiedot operaation suorittamiseksi, eli sille kuka *tuntee lainaajat*
  - Kukin Lainaaja-olio tuntee ainoastaan yhden lainaajan eli itsensä, joten se ei tule kyseeseen
  - Nimike, Kirja ja Laina eivät selvästikään tule kyseeseen
  - Jarjestelma-olio tuntee kaikki lainaajat, joten se on siinä mielessä sovelias operaation suorittajaksi
- Tarkastellessa muitakin operaatioita (esim. lisaaLainaaja ja lisaaNimike), huomataan, että Jarjestelma-oliosta uhkaa muodostua olio, joka suorittaa suuren joukon operaatioita, jotka eivät toiminnallisesti liity toisiinsa
  - Näin ollen rikottaisiin **single responsibility** -periaatetta
- Ohjelman luokkarakennetta kannattaa muuttaa siten, että saadaan aikaan pienempiä, enemmän yhdelle asialle omistautuneita luokkia

- *Lisätään luokka Lainaajatiedot*, jonka vastuuna on huolehtia yksittäisistä Lainaaja-olioista
- Alla muokattu luokkamalli, myös yhteyksiä tarkennettu (navigointisuunnat, kompositio)



Operaation tunnistaLainaaja(nro) suunnittelu jatkuu

- Operaatio alkaa sillä, että käyttöliittymä kutsuu Jarjestelma-olion (joka siis toimii ohjausoliona) operaatiota tunnistaLainaaja(nro)
- Jarjestelma ei enää tunne lainaajia, mutta se tuntee Lainaajatiedot-olion, joka taas tuntee lainaajat
  - operaation toteutus Jarjestelma-oliossa *delegoi* pyynnön edelleen lainaajatietoja hallinnoivalle Lainaajatiedot-oliolle
  - Vastauksena saadaan viite etsittyyn Lainaaja-olioon
- Tämän jälkeen Jarjestelma-olio kysyy löydetyltä lainaajalta nimen ja osoitteen ja palauttaa ne kutsujalle

- Lainaajatiedot-oliolle on siis delegoitu vastuu annettua kirjastokorttinumeroa vastaavan Lainaaja-olion löytämisestä
- Lainaajatiedot-olio tarkastaa *jokaiselta sisältämältään* Lainaaja-olioilta, onko se etsitty
  - Jos jokin Lainaaja-olioista on etsitty, palautetaan kutsujalle viite löydettyyn Lainaaja-olioon
  - Jos lainaajaa ei tunnisteta, palautetaan null viite, eli viite ei mihinkään

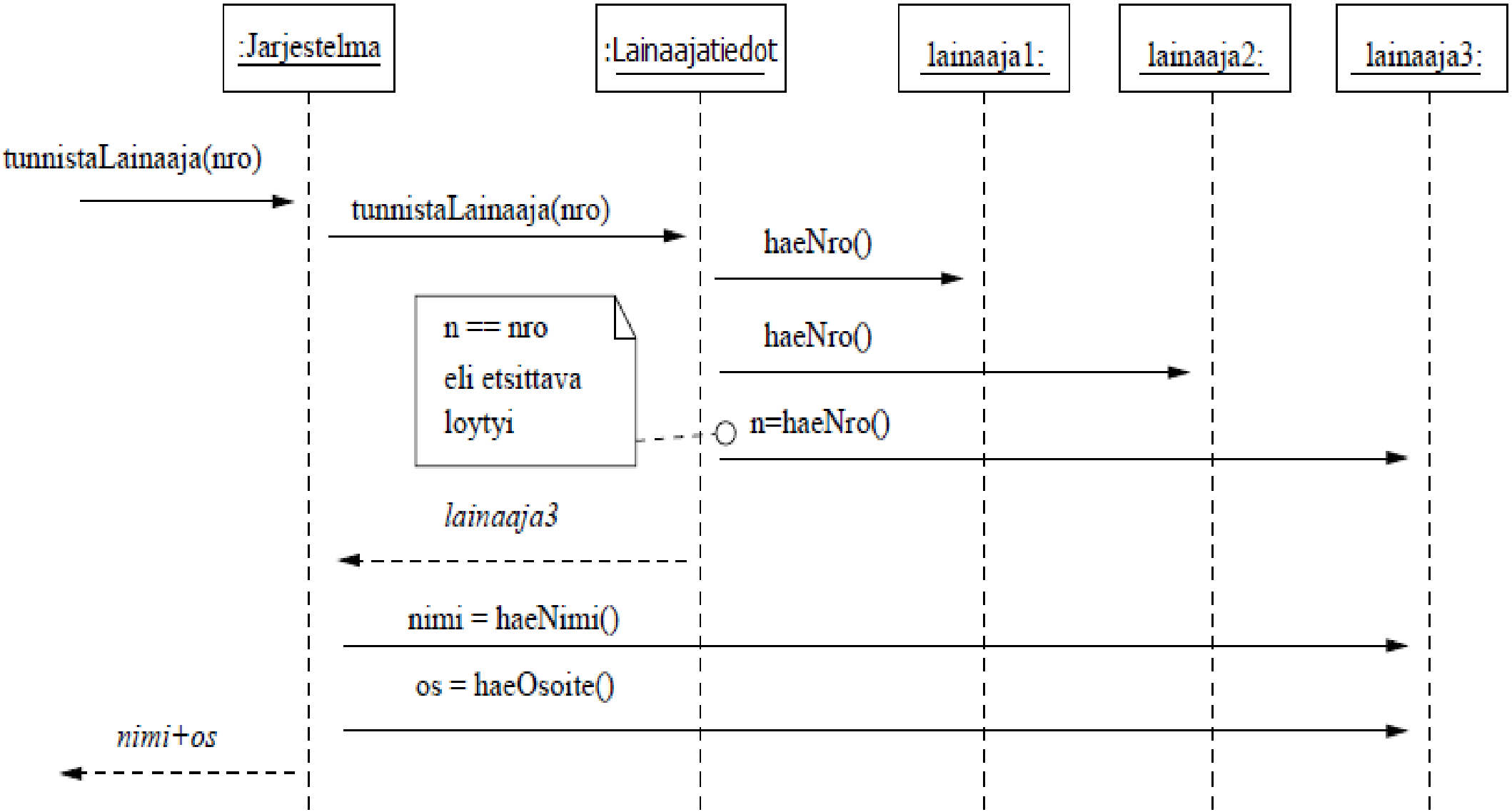
- Alla operaation toteutus pseudokoodina
- Seuraavalla sivulla sekvenssikaavio operaation suorituksen yhdestä mahdollisesta skenaariosta
    - Suunnittelija on päätynyt ratkaisuun, jossa käyttöliittymän kutsuma luokan Jarjestelma operaatio tunnistaLainaajat() palauttaa vastauksen merkkijonona

```
class Jarjestelma {
    String tunnistaLainaaja(int nro){
        Lainaaja vast = lainaajat.tunnistaLainaaja(nro)
        if ( vast == null ) return "tuntematon"
        String nimi = vast.haeNimi()
        String os = vast.haeOsoite()
        return nimi+os;
    }
}
```

```
class Lainaajatiedot {
    Lainaaja tunnistaLainaaja(int etsitty){
        for( Lainaaja lainaaja : lainaajat ) {
            int n =  lainaaja.haeNro();
            if ( etsitty == n )  return lainaaja;
        }

        // ei löytynyt lainaajaa
        return null;
    }
```

- Operaation tunnistaLainaaja(nro) toteutus sekvenssikaaviona
- Kuvattu tilanne, missä kolmas läpikäytävä lainaajaolio (lainaaja3) on etsitty
  - Lue sekvenssikaaviota ja koodia rinnakkain ja varmista, että ymmärrät operaation toteutuksen logiikan



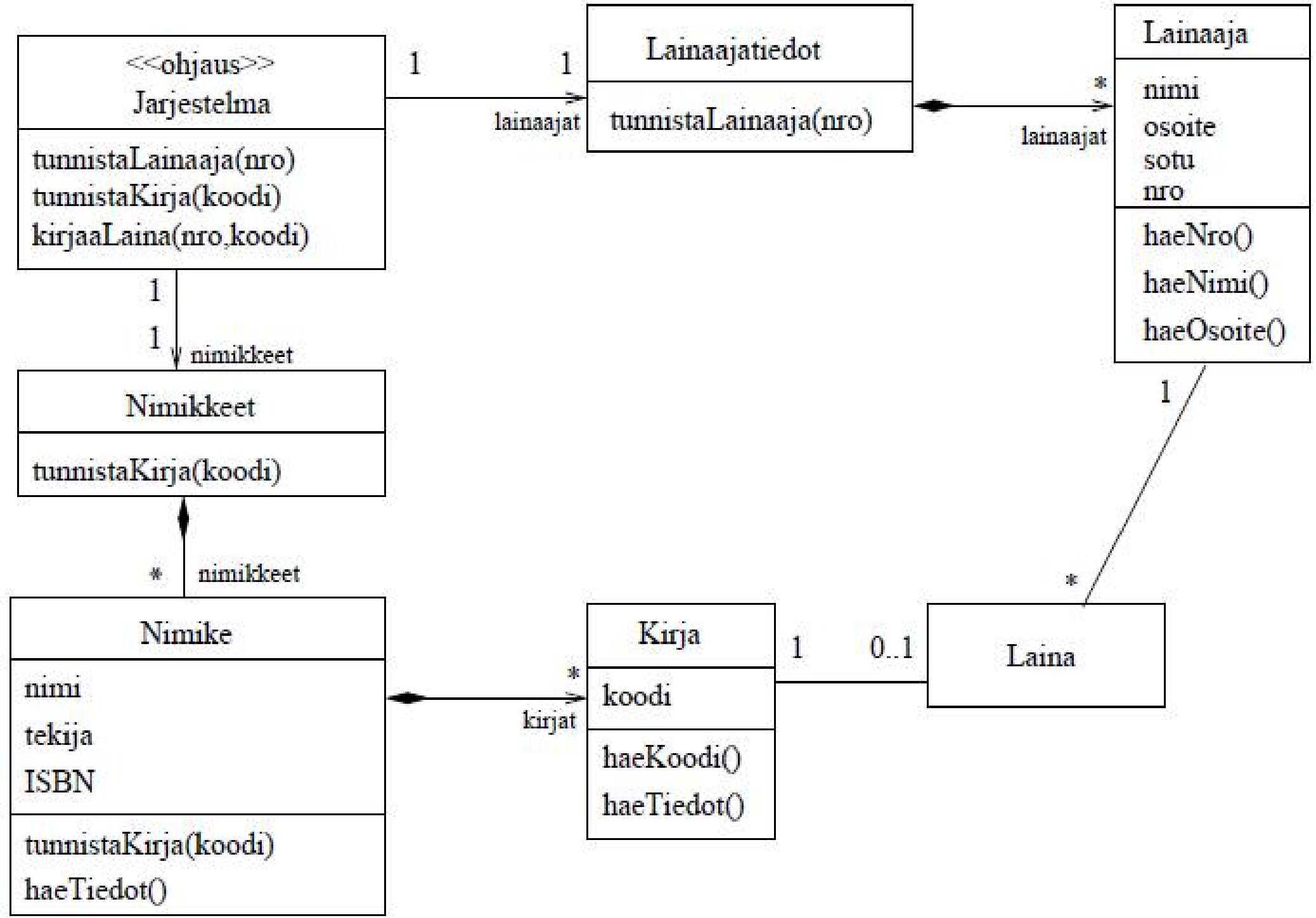
Operaation tunnistaLainaaja(nro) toteutus

- tunnistaLainaaja(nro) operaation toteutus siis tapahtuu usean olion yhteistyönä
- Operaatio suunniteltiin järjen ja muutama sivu sitten listattujen periaatteiden avulla
- **Eksperttiperiaatteen** mukaan aluksi näytti siltä, että luokka Jarjestelma on paras ottamaan operaation vastuulleen
- Järkeilemällä ja **single responsibility -periaatteen** nojalla päädyttiin tuomaan järjestelmään uusi luokka Lainaajatiedot
- Yksittäisen lainaajan tunnistaminen on **eksperttiperiaatteen** mukaan selvästi uuden luokan Lainaajatiedot vastuulla
- Eli ohjausoliona (joka saatiin **ohjausperiaatteen** inspiroimana) oleva Jarjestelma-olio *delegoi* operaation suoritusvastuun Lainaajatiedot-oliolle
  - Käyttöliittymältähän metodikutsu tulee ohjausoliolle
- Koska suunnittelija päättää, että operaatio palauttaa kutsujalle tiedot merkkijonomuodossa, kysyy ohjausolio vielä Lainaaja-oliolta sen tiedot
- Lainaajaoliohan on **eksperttiperiaatteen** mukaan se, joka tuntee omat tietonsa

Operaation tunnistaKirja(koodi) suunnittelu

- Seuraava tarkasteltava operaatio on *tunnistaKirja(koodi)*, jonka tehtävä on etsiä tiettyä koodia vastaava Kirja-olio
- Sovelletaan jälleen ekspertti-periaatetta, eli annetaan vastuu oliolle, jolla on parhaat tiedot operaation suorittamiseksi
- Vastuunalaisen olion täytyy siis tuntea kaikki kirjat
- Tutkimalla luokkamallia huomaamme, että järjestelmään liittyy useita nimikkeitä ja kuhunkin nimikkeeseen liittyy useita kirjoja
  - Ei siis ole olemassa yhtä olioa, joka tuntisi kaikki kirjat
- Olemassaolevia oliota käyttäen ainoa mahdollisuus suorittaa operaatio on selvittää jokaiselta Nimike-oliolta erikseen, liittyykö siihen etsityn koodin omaava kirja
- Koska Jarjestelma-olio tuntee nimikkeet, tulisi sen suorittaa tuo kysely
- Koska pyrkimyksemme on pitää Jarjestelma-olio pelkkänä ohjausoliona, joka lähinnä delegoi tehtäviä muille olioille, tuomme ohjelmaan uuden luokan *Nimikkeet* jonka *vastuulla on Nimike-olioioista huolehtiminen*
- Uusi versio luokkamallista seuraavalla sivulla

Suunnitteluvaiheen luokkamallin kolmas versio



- Operaation toteutus tapahtuu melko monen olion yhteistyönä, joten toteutuksen logiikan seuraaminen voi olla aluksi haasteellista
    - Kannattaa katsoa yhtä aikaa pseudokoodia ja sekvenssikaaviota
- Operaatio alkaa sillä, että käyttöliittymä kutsuu Jarjestelma-olion operaatiota tunnistaKirja(koodi)
- Operaation toteutus delegoi pyynnön edelleen nimiketietoja hallinnoivalle Nimikkeet-oliolle, joka palauttaa viitteen etsittyyn Kirja-olioon
- Saatuaan viitteen Jarjestelma-olio kysyy kirjan tietoja ja palauttaa vastauksen kutsujalle

```
class Jarjestelma{

    String tunnistaKirja(int etsittavanKoodi){

        Kirja k = nimikkeet.tunnistaKirja(etsittavanKoodi)

        If ( k == null ) return "tuntematon"

        String vast = k.haeTiedot()

        return vast

    }

}
```

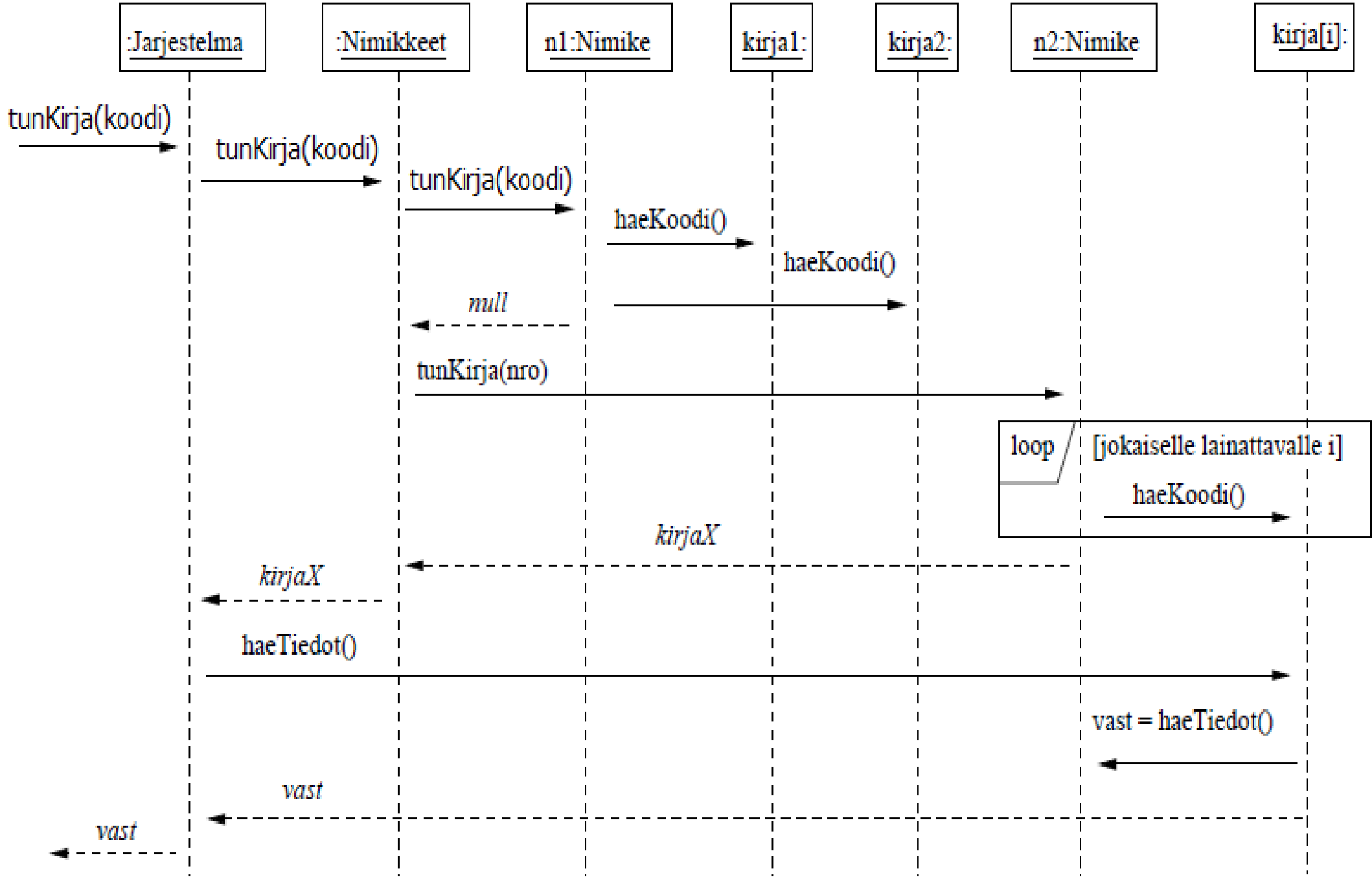
- Suorittaakseen vastuunsa kirjan etsimiseksi, Nimikkeet-olio kysyy jokaiselta tuntemaltaan Nimike-oliolta, tunnistaako se etsityn kirjan
  - Jos jokin Nimike-olioista tunnistaa kirjan (eli kirjalla on etsitty koodi), palauttaa se viitteen löytyneeseen Kirja-olioon
- Nimikkeet-olio siis suorittaa vastuunsa *pilkkomalla sen* yksittäisten Nimike-olioiden *osavastuiksi*
- pseudokoodina:

```
class Nimikkeet{
    Kirja tunnistaKirja(int etsittavanKoodi){
        for ( Nimike n : nimikkeet ) {
            Kirja vast = n.tunnistaKirja(etsittavanKoodi)
            If ( vast != null ) return vast        // etsitty kirja läytyi
        }
        return null                                // koodia vastaavaa kirjaa ei löytynyt
    }
}
```

- Yksittäiseen Nimike-olioon liittyy joukko Kirja-olioita
- Nimike suorittaa vastuunsa tarkastamalla jokaiselta sisältämältään Kirja-olioltaan vastaako sen koodi etsittyä
- Jos etsitty löytyy, palautetaan sen viite. Jos etsittyä kirjaa ei löydy, palautetaan null-viite
- pseudokoodina:

```
class Nimike{
    Kirja tunnistaKirja(int etsittavanKoodi){
        for ( Kirja k : kirjat ) {
            int koodi = k.haeKoodi()
            If ( koodi == etsittavanKoodi ) return k    // etsitty Kirja-olio löytyi
        }
        // jos nimikkeen alta ei löytynyt koodia vastaavaa kirjaa
        return null
    }
}
```

- Operaation tunnistaKirja(nro) toteutus sekvenssikaaviona
  - Käytettyjä merkintätapoja selitetty luentomonisteessa



Huomioita operaatiosta tunnistaKirja(koodi)

- Jarjestelma-olion kuuluu palauttaa etsityn kirjan tiedot merkkijonona
- Saatuaan viitteen etsittyyn Kirja-olioon, Jarjestelma-olio kysyy kirjan tietoja, jotka se palauttaa kutsujalleen
- Kirja-olio ei tiedä itse muuta kuin koodinsa, muut tietonsa (nimi, tekijaä ISBN) se kysyy omalta Nimike-olioltaan

- Operaatio saattaa vaikuttaa aluksi melko monimutkaiselta
- Operaation toteutus vastaa kuitenkin melko hyvin oikeaoppista olioajattelua, eli:
  - Paljon yksinkertaisia olioita, joilla kullakin selkeä vastuu
  - Monimutkainen operaatio toteutuu näiden yksinkertaisten olioiden yhteistyönä
- Operaatiota pystyy yksinkertaistamaan (ja muuttamaan tehokkaammaksi) lisäämällä järjestelmään uuden luokan, joka tuntee kaikki Kirja-oliot
  - Oikeastaan myös Single responsibility -periaatteen nojalla olisi parempi tehdä näin, eihän ole oikeastaan luontevaa antaa Nimikkeet-olion vastuulle kirjan etsimistä

Operaation kirjaaLaina(nro,koodi) toteuttaminen

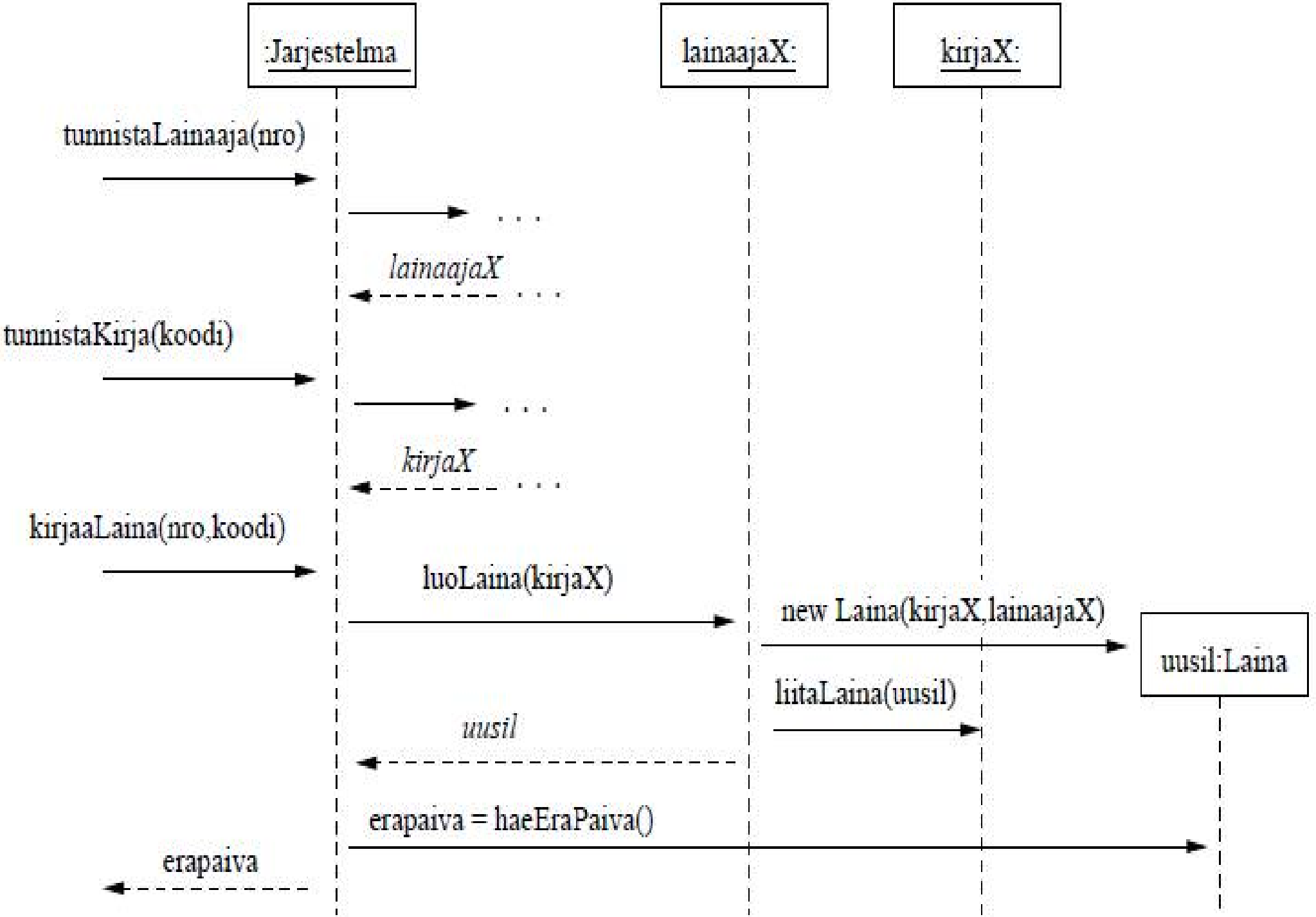
- Käyttötapauksen *Lainaa kirja* viimeinen suunniteltava operaatio on kirjaaLaina(nro,koodi).
- Operaation on siis *luotava Laina-olio*, josta tulee yhteys lainaajaan, jonka kirjastokortin numero on *nro* ja kirjaan, jolla tunnisteena *koodi*
- Muutama kalvo sitten mainitun **luontiperiaatteen** mukaan olion luo se, joka
  - sisältää tai säilyttää olion
  - pitää kirjaa oliosta
  - tuntee olion alustuksessa tarvittavan datan
- Periaatetta soveltaen Laina-olion voisi luoda
  - Lainaaja (säilyttää lainaa), tai
  - Kirja (pitää kirjaa lainasta), tai
  - Jarjestelma-olio, jolla on operaation kutsuhetkellä tiedossa ne oliot (Kirja- ja Lainaaja-oliot), joihin luotava Laina-olio on liitettävä
- Vaihtoehtojen välillä ei ole merkittäviä paremmuuseroja
- Päädytään siihen, että Lainaaja-olio luo Laina-olion

Operaation kirjaaLaina(nro,koodi) kulku

- Operaation toteutusta vastaava sekvenssikaavio seuraavalla sivulla
  - Pseudokoodia ei tälläkertaa esitetä
- Ennen uuden lainan kirjaamista on suoritettu operaatiot tunnistaLainaaja ja tunnistaKirja, joiden perusteella tiedetään mihin Kirja- ja Lainaajaolioon uusi Laina-olio liitetään

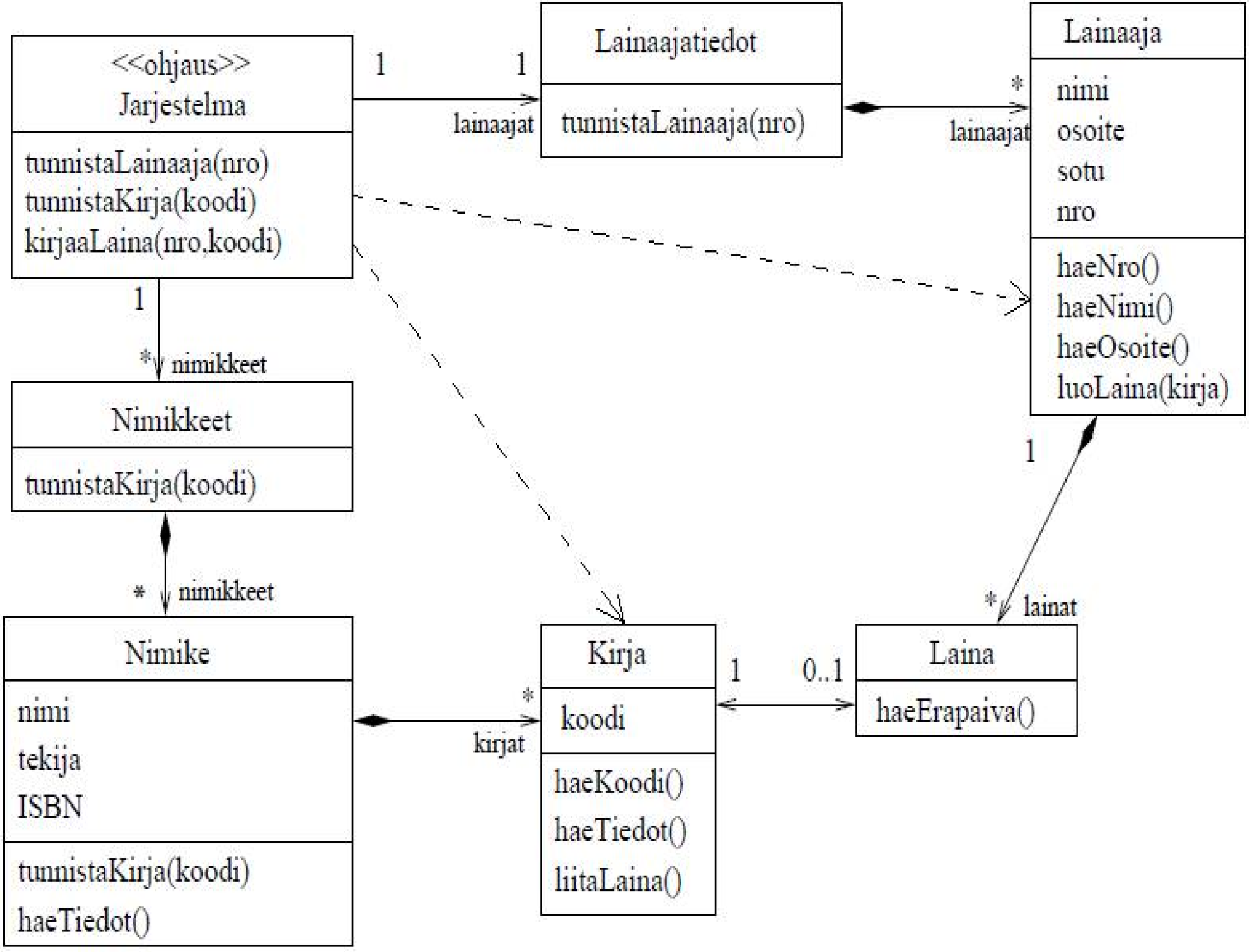
- Jarjestelma-olio delegoi uuden olion luonnin Lainaajalle
- Lainaaja luo uuden Laina-olion ja kertoo viitteen siihen Kirja-oliolle
- Operaatio palauttaa kutsujalle eräpäivän
  - On päädytty siihen, että Laina-olio määrittelee itselleen eräpäivän konstruktorin kutsun yhteydessä
- Jarjestelma-olio kysyy Lainalta eräpäivää ja palauttaa sen operaation kutsujalle
- Huomaa, että Laina-oliolle kerrotaan tieto sekä lainaajasta että kirjasta, johon se liittyy
  - Näin lainaan liittyvät yhteydet ovat kaksisuuntaisia

Operaation kirjaaLaina(nro,koodi) toteutus sekvenssikaaviona



- Käyttötapauksen suorittaminen siis edellyttää kolmen operaation, tunnistaLainaaja, tunnistaKirja ja luoLaina peräkkäisen suorittamisen
- Suorittaakseen viimeisen operaation, on Jarjestelma-olion talletettava viitteet kahden edellisen operaation aikaansaannoksena löydettyihin Kirja- ja Lainaaja-olioihin
- Tämä seikka voidaan tuoda luokkakaaviossa esiin piirtämällä Jarjestelma-luokasta riippuvuus luokkiin Kirja ja Koodi
    - Koska kyse on tilapäisistä, yhden käyttötapauksen suorituksen aikana olemassa olevasta suhteesta olioiden välillä, ei niitä kannata merkitä luokkakaavioon yhteyksinä
- Seuraavalla sivulla siis luokkakaavio käyttötapauksen Lainaa kirja -suunnittelu jälkeen

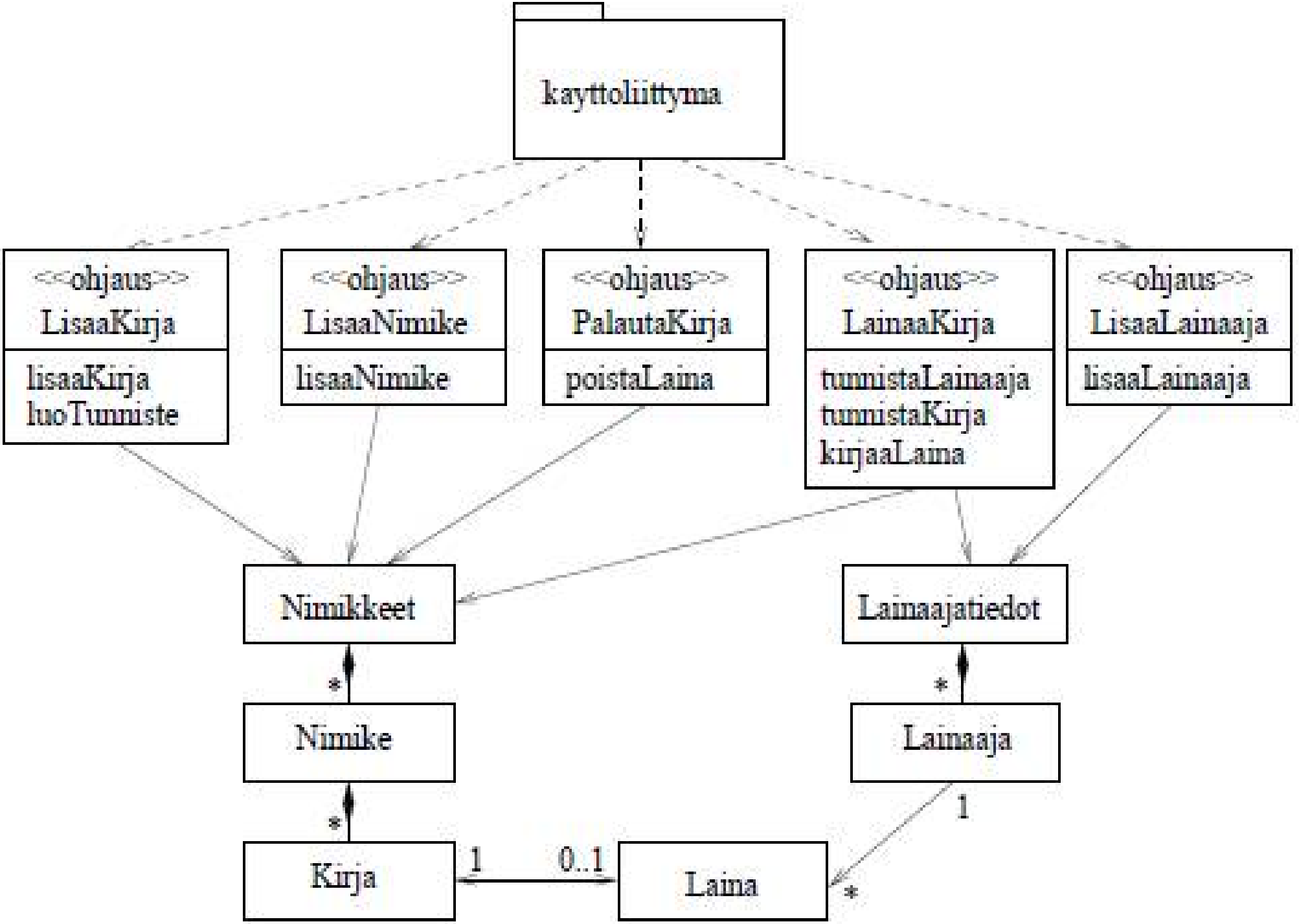
- **HUOM**: laina-olioiden luominen on nyt lainaajien vastuulla. Onko tämä järkevää?
    - Lainaajan vastuu on tietää henkilön tiedot (nimi, osoite, kirjastokorttinumero)
    - Kenties olisi parempi muodostaa uusi luokka jonka vastuulla on lainojen käsittely, eli näyttää siltä että **olemme rikkoneet single responsibility -periaatetta**


<<ohjaus>>
Jarjestelma
tunnistaLainaaja(nro)
tunnistaKirja(koodi)
kirjaaLaina(nro,koodi)
1
1
lainaajat
Lainaajatiedot
tunnistaLainaaja(nro)
*
lainaajat
Lainaaja
nimi
osoite
sotu
nro
haeNro()
haeNimi()
haeOsoite()
luoLaina(kirja)
1
*
nimikkeet
Nimikkeet
tunnistaKirja(koodi)
*
nimikkeet
Nimike
nimi
tekija
ISBN
tunnistaKirja(koodi)
haeTiedot()
*
kirjat
Kirja
koodi
haeKoodi()
haeTiedot()
liitaLaina()
1
0..1
Laina
haeErapaiva()
1
*
lainat

Yksi vai monta ohjausolioa?

- Kaikkien käyttötapausten ohjauksesta vastaavan Jarjestelma-luokkan metodit ovat toiminnallisuudeltaan varsin yksinkertaisia, ne ainoastaan delegoivat vastuita muille luokille
- Kun kaikki ensimmäisen iteraation käyttötapausten vaatimat operaatiot toteutetaan, tulee luokan rajapinnasta turhan epäyhtenäinen
- Järjestelmään lisätään uutta toiminnallisuutta myöhemmissä iteraatioissa, eli luokan rajapinta uhkaa tulevaisuudessa epäyhtenäistyä entisestään
- Aiemmin mainittiin, että vaihtoehtona yhdelle kaikesta huolehtivalle ohjausoliolle on *varata jokaiselle käyttötapaukselle oma ohjausolio*
- Näin kannattaa tehdä
  - järjestelmän uusi luokkarakenne seuraavalla sivulla
- Nyt järjestelmän toiminnallisuuden laajentaminen kattamaan uusia käyttötapauksia voidaan hoitaa suunnittelemalla uusia ohjausluokkia entisten jäädessä ennalleen
- Myös muutos tietyn käyttötapauksen toiminnallisuuteen paikallistuu ainoastaan yhteen luokkaan
- Käyttöliittymällä on nyt riippuvuuksia yhden luokan sijasta moneen luokkaan. Uuden luokkarakenteen edut ovat tätä haittaa suuremmat

Suunnitteluvaiheen luokkamalli 1. iteraation jälkeen



Oliosuunnittelu yhteenveto

- Yhden käyttötapauksen vaatimat operaatiot on nyt suunniteltu
- Muiden neljän ensimmäiseen iteraatioon valittujen käyttötapausten operaatioiden suunnittelu luentomonisteessa
- Suunnittelussa taustalla periaatteet
  - Ohjausperiaate
  - Eksperttiperiaate
  - Luontiperiaate
  - Single responsibility -periaate
  - Riippuvuuksien minimoinnin periaate
  - Ja ”nimetön periaate”, joka kehotti ottamaan mukaan tarvittaessa uusia luokkia
- Monisteessa ja kalvoissa asiat tehtiin aika perinpohjaisesti
- Tavoitteena, on että hyvät oliosuunnittelutavat tulevat enemmän tai vähemmän suunnittelijan selkärangasta
- Kalvoilla ja monisteessa suunnittelu eteni suoraviivaisesti
  - Todellisuudessa näin ei käynyt, ratkaisut luonnosteltiin ensin ja luonnoksia hiottiin kohti tässä esitettyä ratkaisua