Ohjelmistojen mallintaminen

Luento 9, 30.11.

Kertaus: oliosuunnittelun periaatteita

- **Single responsibility**
  - Jokaisella luokalla vain yksi selkeä vastuu
- **Favour composition over inheritance**
  - Älä liikakäytä perintää
- **Program to an interface, not to an Implementation**
  - Tee luokat mahdollisimman riippumattomiksi toisistaan
  - Tunne vain rajapinta
- ”ikiaikaisia periaatteita”
- Motivaationa ohjelman muokattavuuden ja uusiokäytettävyyden parantaminen
- Huonoa oliosuunnittelua on verrattu *velan* (engl. design debt tai technical debt) ottamiseen
  - Piittaamattomalla ja laiskalla ohjelmoinnilla/suunnittelulla saadaan ehkä nopeasti aikaan jotain, mutta hätäinen ratkaisu tullaan maksamaan korkoineen takaisin myöhemmin kun ohjelmaa on tarkoitus laajentaa tai muuttaa

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
- Lääke koodihajuun on *refaktorointi* eli muutos koodin rakenteeseen joka kuitenkin pitää koodin toiminnan ennallaan

Koodin refaktorointi

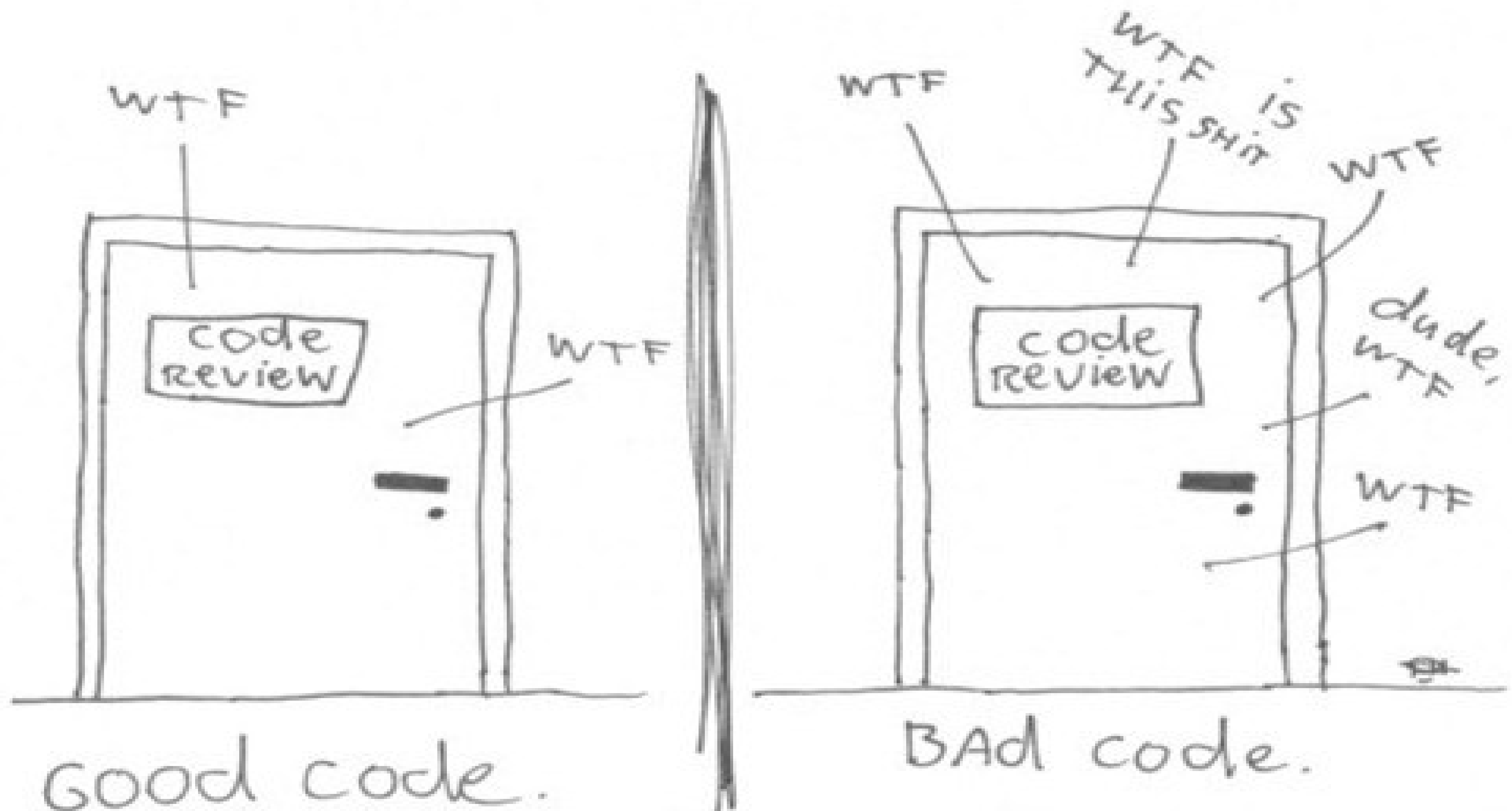
- Erilaisia koodin rakennetta parantavia refaktorointeja on lukuisia
  - ks esim. http://sourcemaking.com/refactoring
- Pari hyvin käyttökelpoista ja nykyaikaisessa kehitysympäristössä (esim NetBeans, Eclipse, IntelliJ) automatisoitua refaktorointia:
  - **Rename method** (rename variable, rename class)
    - Eli uudelleennietään huonosti nimetty asia
  - **Extract method**
    - Jaetaan liian pitkä metodi erottamalla siitä omia apumetodejaan
- Seuraavassa esimerkki Ohjan viikon 2 tehtävästä 1 Kokonaislukujoukko
  - Kohdan 1.5 lisäys suuruusjärjestyksessä olevaan taulukkoon
  - 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 ( kuuluuJoukkoon(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 kukin yhden pienen asian
- Apumetodit ovat lyhyitä, helppoja ymmärtää ja helppoja tehdä
- Koodi kannattaa kirjoittaa osin jo alusta asti suoraan ”puhtaaksi ja hajuttomaksi”
    - Helpompaa tehdä ja saada toimimaan oikein
- Syksyn 2011 kurssilla *Ohjelmistokehitys* palataan tarkemmin refaktorointiin ja puhtaan koodin kirjoittamiseen

The only valid measurement of code quality: WTFs/minute



(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

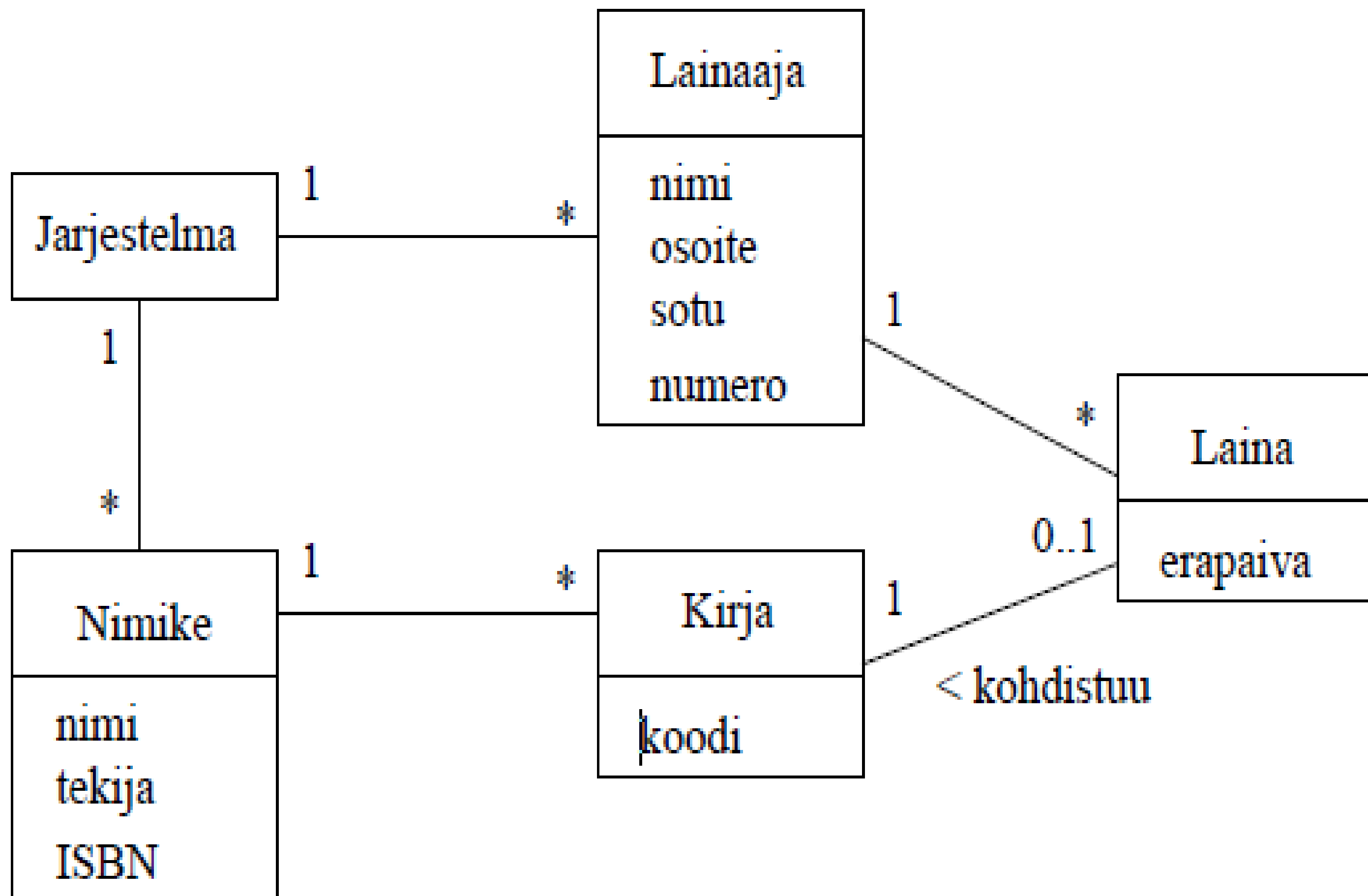
Menetelmä

- Aloitimme perjantaina tutustumisen *erääseen* oliomenetelmään
- **Menetelmä** siis kertoo, miten edetään ohjelmistoprosessin eri vaiheissa
    - Mitä tekniikoita, strategioita ja apuvälineitä (esim. UML-kaavioita) tulisi käyttää ohjelmiston kehittämisen tukena
    - Miten apuvälineitä kannattaa soveltaa
    - Nyt polttopisteessä *mitä ja miten UML-kaavioita käytetään* ohjelmistokehityksessä?
- Kaikissa oliomenetelmissä seuraavat vaiheet:
    - Vaatimusmäärittely ja olioanalyysi
    - Suunnittelu
        - Arkkitehtuurisuunnittelu
        - Oliosuunnittelu
- Menetelmässämme oliosuunnittelu perustuu ns. *vastuuperustaiseen oliosuunnitteluun (engl. responsibility driven object design)*
    - Esityksen pohjana Larmanin kirja
- Näillä kalvoilla asiat ylimalkaisemmin kuin monisteessa

Kirjastoesimerkki

- Esimerkkinä pienen kirjaston tietojärjestelmä
  - Ei moderneja ominaisuuksia, mm. verkkokäyttöä
  - ”klassikkoesimerkki”, käytössä monissa kirjoissa. Tässä lähteenä Erikssonin ja Penkerin 90-luvun lopulla kirjoittama kirja
- Ajan hengen mukaan projektissa edetään *iteratiivisesti*
- Ensimmäisessä iteraatiossa toteutetaan seuraavat käyttötapaukset:
  - Lainaa kirja
  - Palauta kirja
  - Lisää nimike
  - Lisää kirja
  - Lisää lainaaja
- Viimeksi tarkennettiin *käyttötapaukset* ja luotiin *kohdealueen luokkamalli*, joka kattaa ensimmäisen iteraation toiminnallisuuden
  - Kohdealueen luokkamalli seuraavalla sivulla muistutuksena

- Määrittelyvaiheessa ei luokille vielä merkitty metodeita
- Luokat saavat metodit vasta oliosuunnitteluvaiheessa
- Huomaa luokkien *Nimike* ja *Kirja* merkitys
  - Kirja on hyllyssä pidettävä lainattavissa oleva asia, Nimike taas on olio, johon liittyvät kirjan tiedot
  - Yhteen nimikkeeseen (esim. Bob Martin: Clean Code) liittyy useita kirjoja (esim. 5 kirjaa joista 3 hyllyssä ja 2 lainassa)

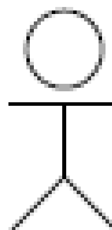


Kohti suunnittelua

- Toimiakseen halutulla tavalla, on järjestelmän tarjottava ne operaatiot, jotka käyttötapausten läpiviemiseen vaaditaan
- Yksittäisen käyttötapauksen vaatiman toiminnallisuuden toteuttamiseksi järjestelmä joutuu yleensä toteuttamaan muutaman erilaisen yksittäisen operaation
- Joissain tilanteissa on hyödyllistä dokumentoida tarkasti, mitä yksittäisiä operaatioita käyttötapauksen toiminnallisuuden toteuttamiseksi järjestelmältä vaaditaan
- Dokumentointiin sopivat *järjestelmätason sekvenssikaaviot*, eli sekvenssikaaviot, joissa koko järjestelmä ajatellaan yhtenä oliona
- Näin siis saadaan selkeästi esille ne yksittäiset toiminnot, joita järjestelmään kohdistetaan sen toiminnan aikana

- Eli ennen kun etenemme varsinaiseen suunnitteluun, täsmennetään ohjelmalta vaadittava toiminnallisuus esittämällä ensimmäisen iteraation käyttötapauksia vastaavat järjestelmätason sekvenssikaaviot

Lainaa kirja – käyttötapauksen kulku ja sekvenssikaavio

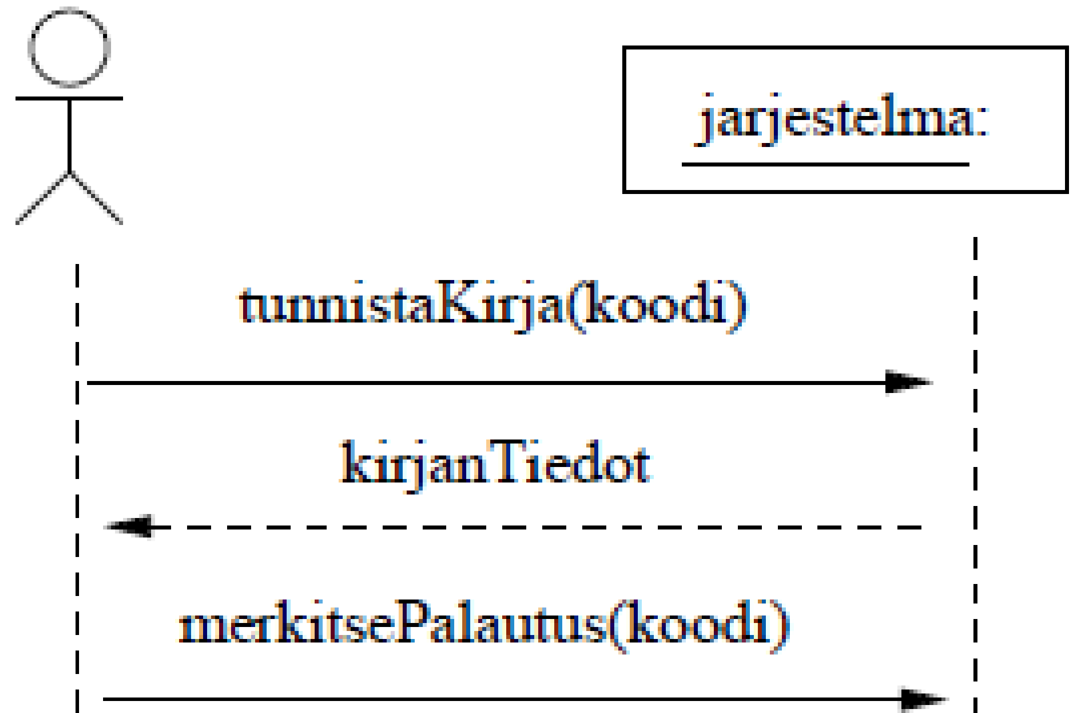


Käyttötapauksen kulku:

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

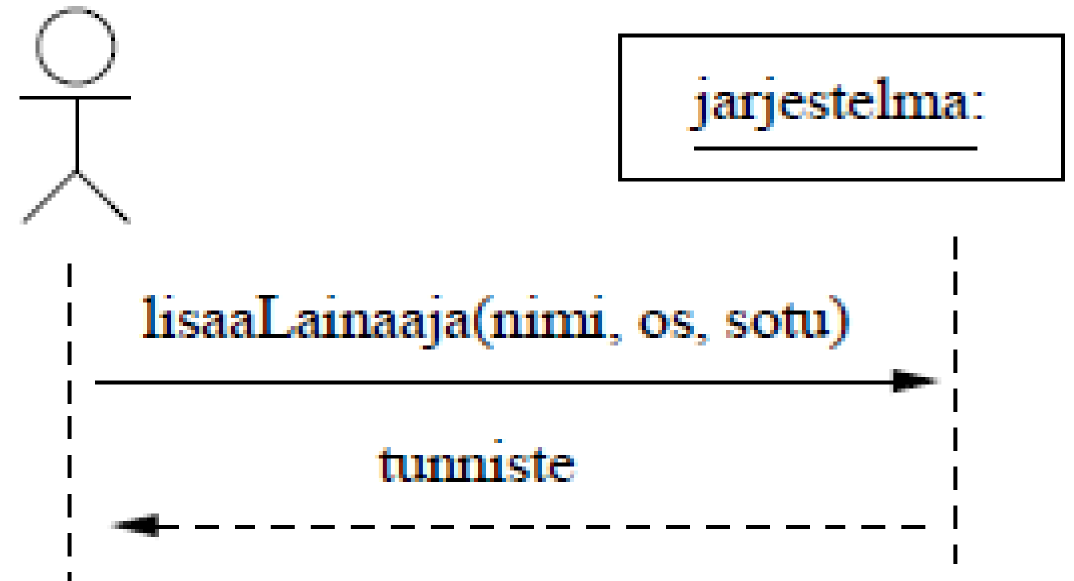
Palauta kirja

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



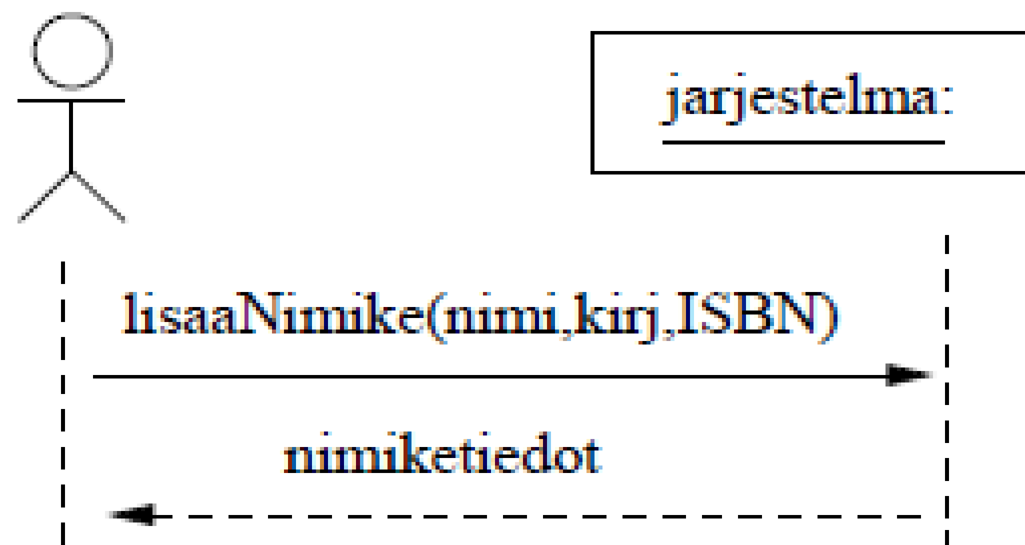
Lisää lainaaja

4. Kirjataan järjestelmään uuden lainaajan tiedot
5. Järjestelmä palauttaa uuden lainaajan tiedot, erityisesti lainaajanumeron, joka toimii lainaajan yksikäsitteisenä tunnisteena



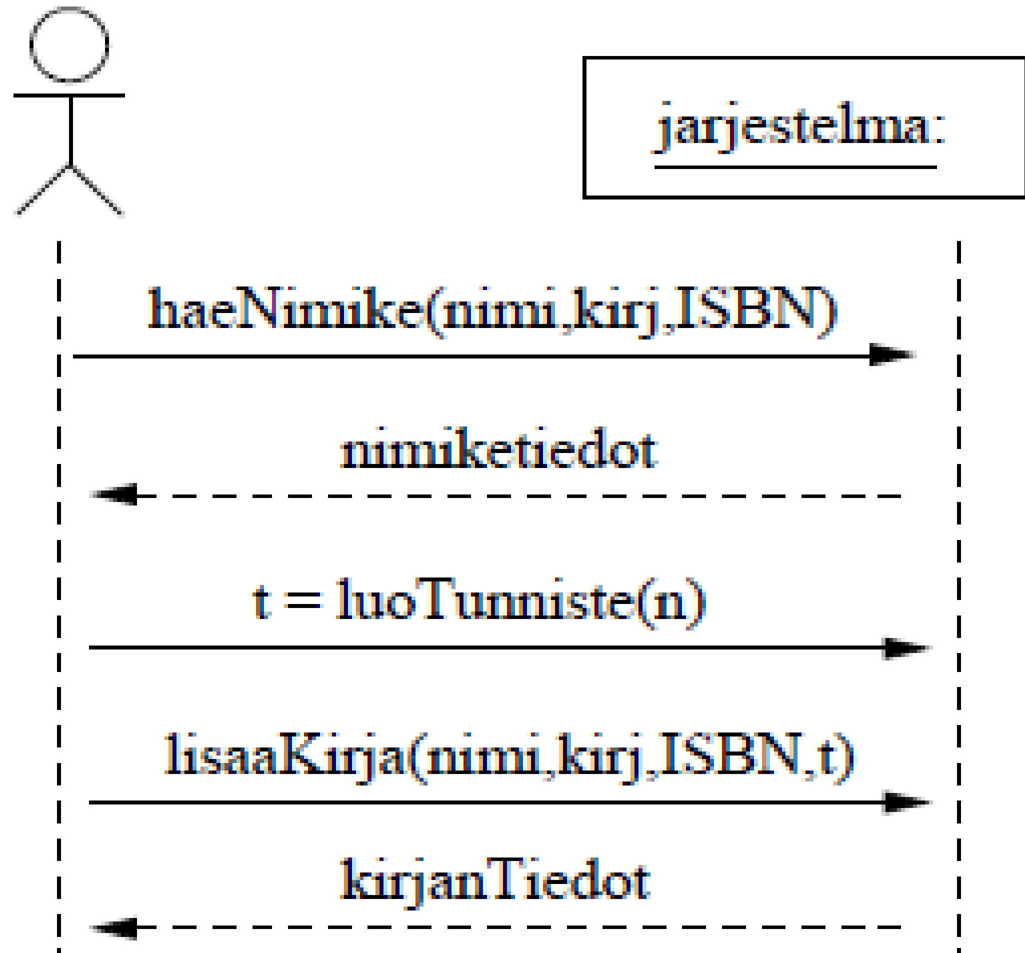
Lisää nimike

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



Lisää kirja

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



Järjestelmän tarjoamat operaatiot

- Järjestelmätason sekvenssikaaviosta näemme selkeästi, mitä kommunikaatiota käyttäjän ja järjestelmän välillä on
  - Erityisesti näemme operaatiot, jotka järjestelmän on suoritettava eri käyttötapausten aikana
  - Yksittäiset operaatiot on sekvenssikaaviossa nimetty, näin operaatioihin on helpompi viitata kuin käyttötapauksen yksittäisiin askeliin
  - Huomaamme, että samaa operaatiota saatetaan tarvita useampaa käyttötapausta suoritettaessa
- Pystyäkseen suorittamaan kaikki vaaditut käyttötapaukset, on järjestelmän toteutettava kaikki järjestelmätason sekvenssikaavioissa ilmenevät operaatiot

- Operaatioiden nimi ja parametrit on tietenkin sekvenssikaavion piirtäjän (joka on todennäköisesti ohjelmistosuunnittelija) päätettävä
  - Nimet ja parametrit saattavat muuttua vielä suunnittelun edetessä, mutta se ei estä tekemästä jo jotain tässä vaiheessa

- Lista järjestelmän operaatiosta
  - 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)
- Seuraavilla sivuilla vielä operaatiot tarkemmin
- Jokaisesta operaatiosta nimen ja parametrien lisäksi mainittu operaation aiheuttamat *toimenpiteet*
  - *Toimenpiteistä* kirjataan tässä tapauksessa ainoastaan *käyttäjälle näkyvä lopputulos*, se miten toimenpide toteutetaan on vasta suunnitteluvaiheen asia
  - Operaatioiden toimenpiteitä ovat *tulosteet* sekä *kohdealueen luokkamallin tasolla näkyvät seikat*, esim. luodaan laina, joka liittyy tiettyyn lainaajaan ja lainattuun kirjaan

Järjestelmän operaatiot ja niiden toimenpiteet

- tunnistaLainaaja(nro)
  - Tunnistetaan lainaaja ja tulostetaan lainaajan tiedot
- tunnistaKirja(koodi)
  - Tunnistetaan yksittäinen kirja ja tulostetaan sen tiedot
- kirjaaLaina(nro, koodi)
  - Luodaan Laina-olio, joka liitetään lainaajaan, jonka kirjastokortin numero *nro* ja kirjaan, jolla tunnisteena *koodi*
  - Tämä operaatio siis aiheuttaa toimenpiteen, joka heijastuu sovelluksen olioihin
- merkitsePalautus(koodi)
  - Kirjaa, jonka tunniste *koodi* vastaava Laina-olio tuhotaan
- lisaaLainaaja(nimi, os, sotu)
  - Luodaan järjestelmään Lainaaja-olio, jonka attribuutteina operaation parametreina olevat tiedot.
  - Operaatio palauttaa lainaajan lainaajanumeron, jonka avulla lainaaja voidaan tunnistaa yksikäsitteisesti

Lisää järjestelmän operaatiota

- lisaaNimike(nimi, kirj, ISBN)
  - Luodaan järjestelmään Nimike-olio, jolla attribuutteina parametrina annettuja tietoja vastaavat tiedot
  - Operaatio palauttaa lisätyn nimikkeen tiedot
  - Todellisuudessa nimikkeeseen liittyy paljon muitakin tietoja, esim. aiheluokitus, kustantaja, painovuosi, painos, ...
- tunnistaNimike(nimi, kirj, ISBN)
  - Tulostetaan tietoja vastaavaa nimikettä vastaavat tiedot
- luoTunniste(nimi, kirj, ISBN)
  - Pyydetään tietoja vastaavan nimikkeen uudelle kirjalle yksikäsitteinen tunnistenumero
- lisaaKirja(nimi, kirj, ISBN, tunniste)
  - Luodaan tietoja vastaavalle nimikkeelle uusi Kirja-olio, jolla attribuuttina parametrina oleva tunniste

Määrittelystä suunnittelun

- Järjestelmätason sekvenssikaaviot siis tuovat selkeästi esiin, mihin toimintoihin järjestelmän on kyettävä toteuttaakseen asiakkaan vaatimukset (jotka siis on kirjattu käyttötapauksina)
- Sekvenssikaavioista ilmikäyvien operaatioiden voi ajatella muodostavat *järjestelmän ulospäin näkyvän rajapinnan*
  - Kyseessä ei siis vielä varsinaisesti ole suunnittelutason asia
  - Nyt alkaa kuitenkin konkretisoitua, mitä järjestelmältä tarkalleen ottaen vaaditaan, eli mitä operaatiota järjestelmällä on ja mitä operaatioiden on tarkoitus saada aikaan
- Tässä vaiheessa siirrymme suunnitteluun
- Kuten muutama sivu sitten mainittiin, saattavat järjestelmän ulospäin näkyvät operaatiot vielä tarkentua nimien ja parametrien osalta
  - tämä ei haittaa sillä on täysin ketterien menetelmien hengen mukaista, että astiat tarkentuvat ja muuttuvat sitä mukaa järjestelmän suunnittelu etenee

Ohjelmiston suunnittelu

- *Suunnitteluvaiheessa tarkoituksena on löytää sellaiset oliot, jotka pystyvät yhteistoiminnallaan toteuttamaan edellisessä luvussa listatut järjestelmältä vaadittavat operaatiot*
- Suunnittelu jakautuu karkeasti ottaen kahteen vaiheeseen:
  - Arkkitehtuurisuunnittelu
  - Oliosuunnittelu
- Ensimmäinen vaihe on **arkkitehtuurisuunnittelu**, jonka aikana hahmotellaan järjestelmän rakenne karkeammalla tasolla
- Tämän jälkeen suoritetaan **oliosuunnittelu**, eli suunnitellaan oliot, jotka ottavat vastuulleen järjestelmältä vaaditun toiminnallisuuden toteuttamisen
  - Yksittäiset oliot eivät yleensä pysty toteuttamaan kovin paljoa järjestelmän toiminnallisuudesta
  - Erityisesti oliosuunnitteluvaiheessa tärkeäksi seikaksi nouseekin *olioiden välinen yhteistyö*, eli se vuorovaikutus, jolla oliot saavat aikaan halutun toiminnallisuuden

Määrittely- ja suunnittelutason luokkien yhteys

- Ennen kun lähdemme suunnitteluun, on syytä korostaa erästä seikkaa
- Sovelluksen kohdealueen (eli määrittelyvaiheen) luokkamallissa esiintyvät luokat edustavat vasta sovellusalueen yleisiä käsitteitä
  - Määrittelyvaiheessa luokille ei edes merkitä vielä mitään metodeita
- Kuten pian tulemme näkemään, monet sovelluksen kohdealueen luokkamallin luokat tulevat siirtymään myös suunnittelu- ja toteutustasolle
  - Osa luokista saattaa jäädä pois suunnitteluvaiheessa, osa muuttaa muotoaan ja tarkentuu
  - Suunnitteluvaiheessa saatetaan myös löytää uusia tarpeellisia kohdealueen käsitteitä
  - Suunnitteluvaiheessa ohjelmaan tulee lähes aina myös *teknisen tason luokkia*, eli luokkia, joilla ei ole suoraa vastinetta sovelluksen kohdealueen käsitteistössä
  - Teknisen tason luokkien tehtävänä on esim. toimia oliosäiliöinä ja sovelluksen ohjausolioina sekä toteuttaa käyttöliittymä ja huolehtia tietokantayhteyksistä

Ohjelmiston arkkitehtuuri

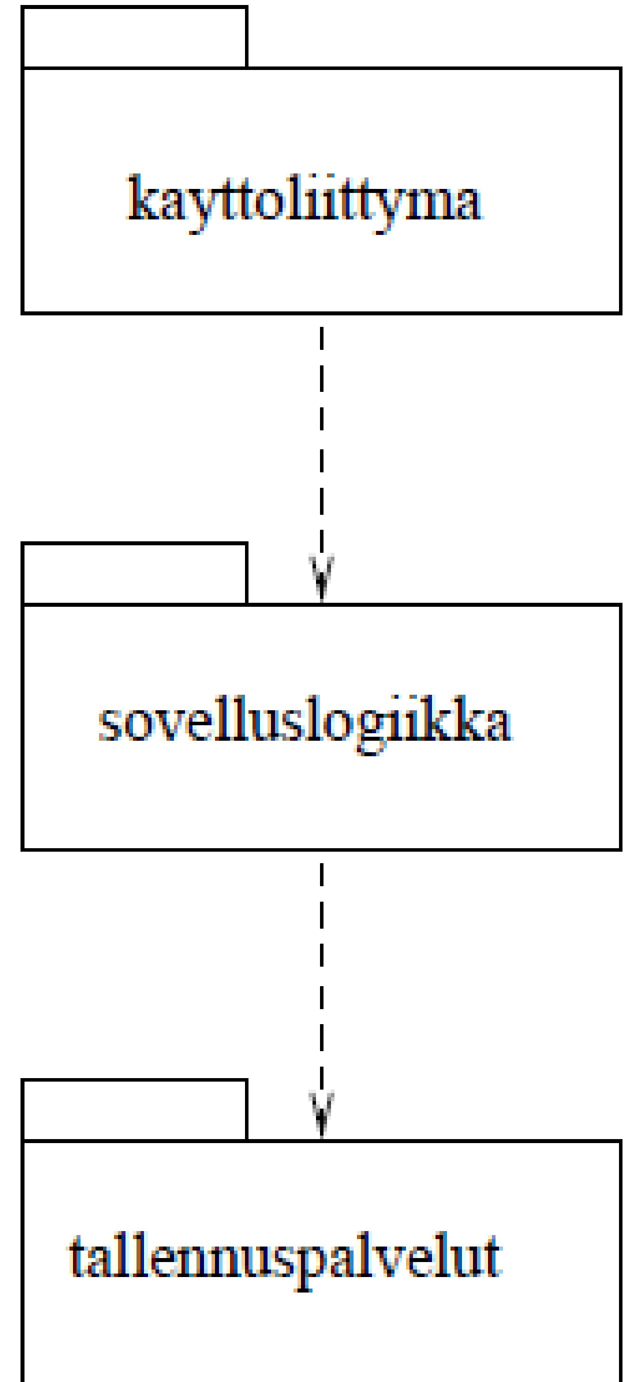
- Ohjelmiston *arkkitehtuurilla* (engl. software architecture) tarkoitetaan ohjelmiston korkean tason rakennetta
  - jakautumista erillisiin *komponentteihin*
  - komponenttien välisiä suhteita
- *Komponentilla* tarkoitetaan yleensä kokoelmaa *toisiinsa liittyviä olioita/luokkia*, jotka suorittavat jotain ohjelmassa tehtäväkokonaisuutta
  - esim. käyttöliittymän voitaisiin ajatella olevan yksi komponentti
- Ison komponentti voi muodostua useista *alikomponenteista*
  - kirjastojärjestelmän sovelluslogiikkakomponetti voisi sisältää komponentin, joka huolehtii sovelluksen alustamisesta ja yhden komponentin kutakin järjestelmän toimintokokonaisuutta varten
- Komponenttijako voi perustua myös tietosisältöihin
  - esim. kirjastojärjestelmässä voisi olla omat komponentit lainaajia, lainoja ja kirjakokoelmaa varten

Ohjelmiston arkkitehtuuri

- Jos ajatellaan pelkkää ohjelman jakautumista komponenteiksi, puhutaan oikeastaan loogisesta arkkitehtuurista
- Looginen arkkitehtuuri ei ota kantaa siihen miten eri komponentit sijoitellaan, eli toimiiko esim. käyttöliittymä samassa koneessa kuin sovelluksen käyttämä tietokanta
- Ohjelmistoarkkitehtuurit on laaja aihe jota käsitellään nyt vain pintapuolisesti
  - Aiheesta on olemassa noin 4. vuotena suoritettava erikoiskurssi *Ohjelmistoarkkitehtuurit*, pakollinen ohjelmistojärjestelmien linjalla
  - Asiaa käsitellään myös 2. vuoden kurssilla *Ohjelmistotuotanto*
- UML:ssa on muutama kaaviotyyppi jotka sopivat arkkitehtuurin kuvaamiseen
  - *Komponenttikaaviota* ja *sijoittelukaaviota* emme nyt käsittele
  - Komponenttikaavio on erittäin käyttökelpoinen kaaviotyyppi, mutta silti jätämme sen myöhemmille kursseille
  - Nyt käytämme *pakkauskaaviota* (engl. package diagram)

Pakkauskaavio

- Kuvassa karkea hahmotelma kirjastojärjestelmän arkkitehtuurista
- Näemme, että järjestelmä on jakautunut kolmeen komponenttiin
    - Käyttöliittymä
    - Sovelluslogiikka
    - Tallennuspalvelut
- Jokainen komponentti on kuvattu omana pakkauksena, eli isona laatikkona, jonka vasempaan ylälaitaan liittyy pieni laatikko
- Laatikoiden välillä on *riippuvuuksia*
    - Käyttöliittymä riippuu sovelluslogiikasta
    - Sovelluslogiikka riippuu tallennuspalveluista
- Järjestelmä perustuu **kerrosarkkitehtuuriin** (engl. layered architecture)

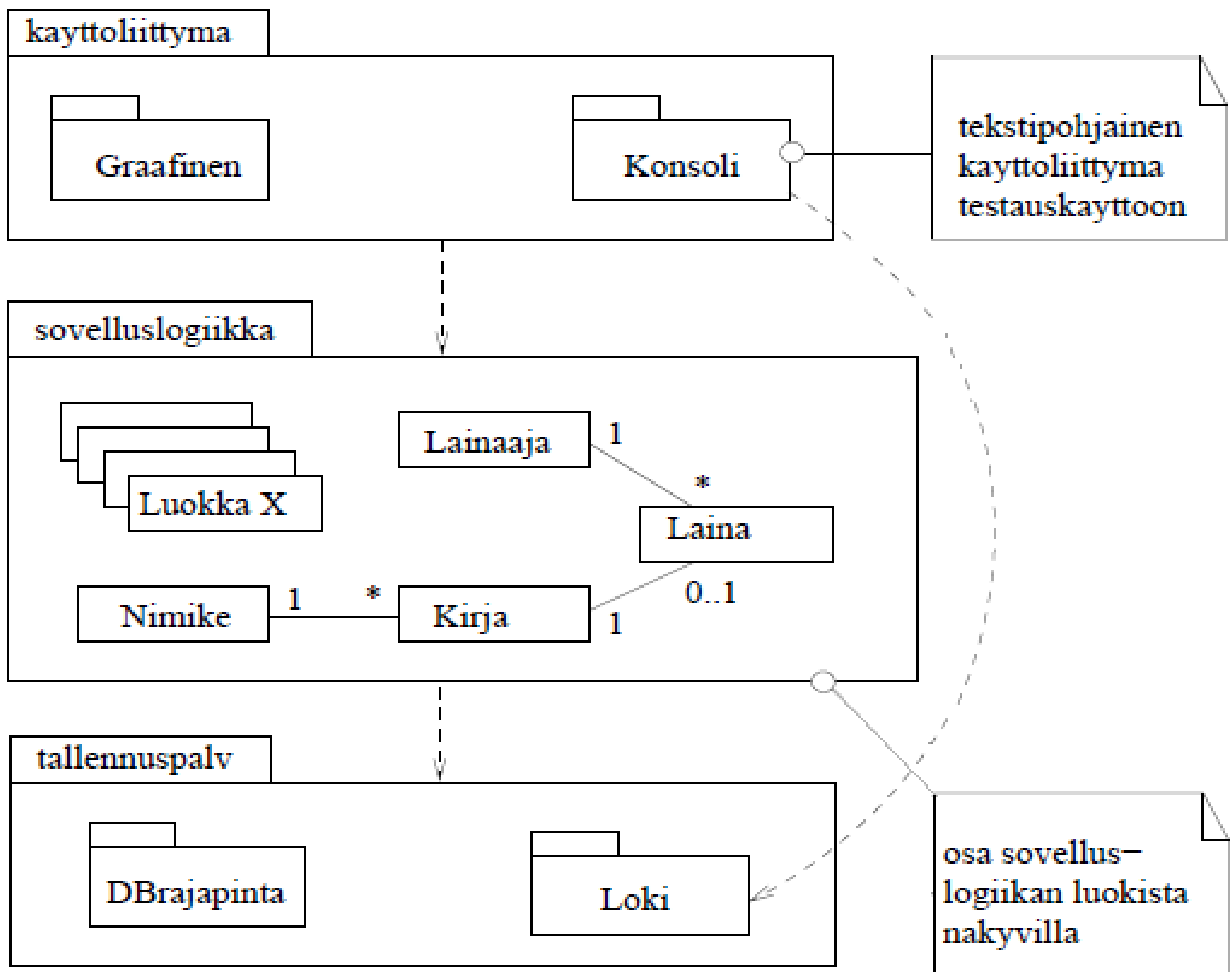


Kerrosarkkitehtuuri

- Kirjastojärjestelmän rakenne perustuu siis **kerrosarkkitehtuuriin**
  - Kerrosarkkitehtuuri yksi hyvin tunnettu *arkkitehtuurimalli* (engl. architecture pattern), eli periaate, jonka mukaan tietynlaisia ohjelmia kannattaa pyrkiä rakentamaan
  - Olemassa myös muita arkkitehtuurimalleja, joita ei nyt kuitenkaan käsitellä
- Kerros on kokoelma toisiinsa liittyviä olioita tai alikomponentteja, jotka muodostavat esim. toiminnallisuuden suhteen loogisen kokonaisuuden ohjelmistosta
- *Kerrosarkkitehtuurissa on pyrkimyksenä järjestellä komponentit siten, että ylempänä oleva kerros käyttää ainoastaan alempana olevien kerroksien tarjoamia palveluita*
  - Ylimpänä kerroksista on käyttöliittymäkerros
  - sen alapuolella sovelluslogiikka
  - alimpana tallennuspalveluiden kerros, eli esim. tietokanta, jonne sovelluksen olioita voidaan tarvittaessa tallentaa.
- Palaamme kerrosarkkitehtuurin hyötyihin pian, ensin hieman lisää pakkauskaavioista

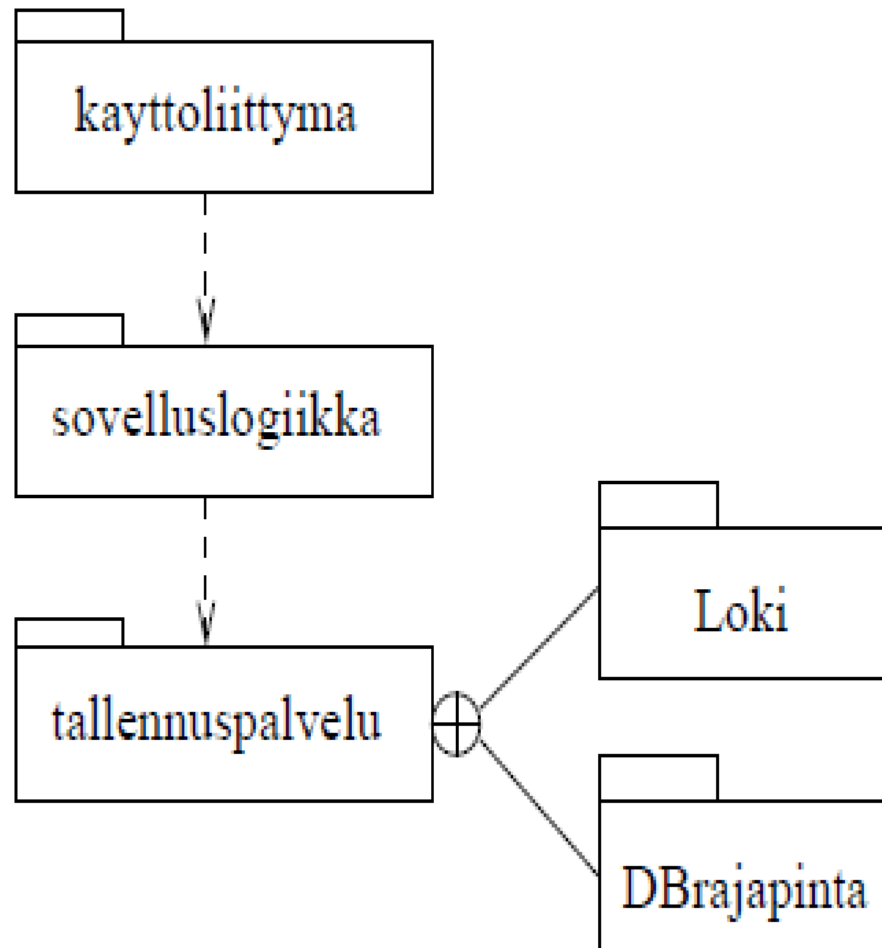
Pakkauskaavio

- Pakkauskaaviossa yksi komponentti kuvataan pakkaussymbolilla
  - Pakkauksen nimi joko keskellä symbolia tai ylänurkan laatikossa
- Pakkausten välillä olevat *riippuvuudet* ilmaistaan *katkoviivanuolena*, joka suuntautuu pakkaukseen, johon riippuvuus kohdistuu
- Kerrosarkkitehtuurissa siis pyrkimyksenä, että riippuvuuksia on ainoastaan alapuolella oleviin kerroksiin. *Kirjastojärjestelmän käyttöliittymäkerros riippuu sovelluslogiikkakerroksesta*
  - Riippuvuus tarkoittaa käytännössä sitä, että *käyttöliittymän oliot kutsuvat sovelluslogiikan olioiden metodeja*
  - Sovelluslogiikkakerros taas on riippuvainen tallennuspalvelukerroksesta
- Pakkauksen sisältö on mahdollista piirtää pakkaussymbolin sisään kuten seuraavalla sivulla olevassa tarkennetussa kirjastojärjestelmän arkkitehtuurikuvauksessa
  - Pakkauksen sisällä voi olla alipakkauksia tai esim. luokkia
- Riippuvuudet voivat olla myös alipakkausten välisiä

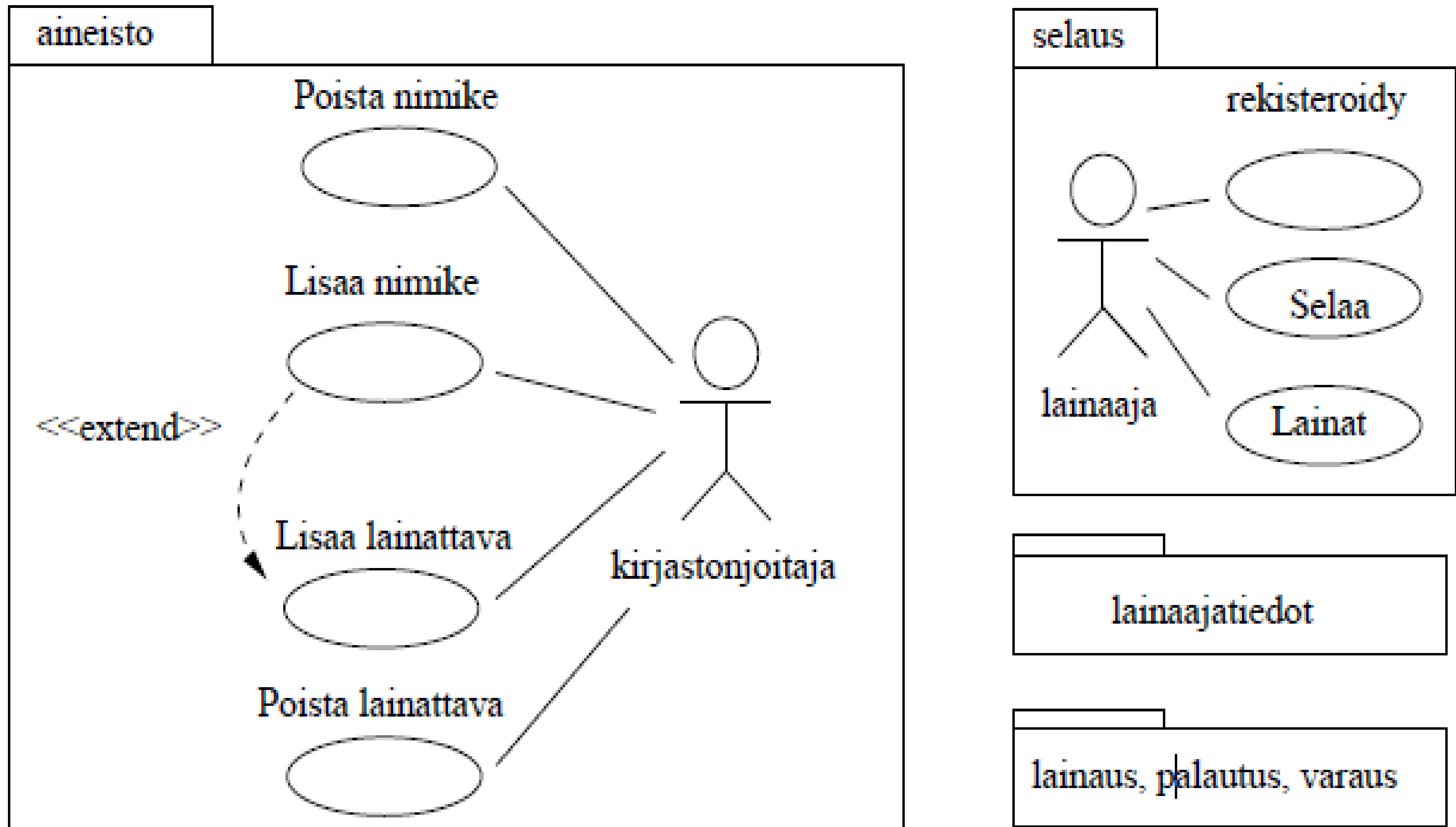

kayttoliittyma
Graafinen
Konsoli
tekstipohjainen
kayttoliittyma
testauskayttoon
sovelluslogiikka
Lainaaja
1
*
Laina
Luokka X
Nimike
1
*
Kirja
0..1
1
tallennuspalv
DBrajapinta
Loki
osa sovellus−
logiikan luokista
nakyvilla

Lisää pakkauskaavioista

- Jos pakkauksessa on paljon sisältöä, voi sisällön näyttäminen piirtämällä sisältyvät komponentit pakkauksen sisäpuolelle olla ongelmallista
- Mahdollista piirtää pakkaukseen sisältyvät komponentit ulkopuolelle
  - Merkitään sisältyminen ympyrällä, jossa plus



- UML:n pakkaussymbolilla voidaan ryhmitellä mitä tahansa komponentteja: pakkauksia, luokkia, olioita, käyttötapauksia, ...
- Voitaisiin esim. jakaa kirjastojärjestelmän käyttötapaukset ylläpidon helpottamiseksi tai dokumentoinnin selkeyttämiseksi neljään eri pakkaukseen



Kerrosarkkitehtuurin etuja

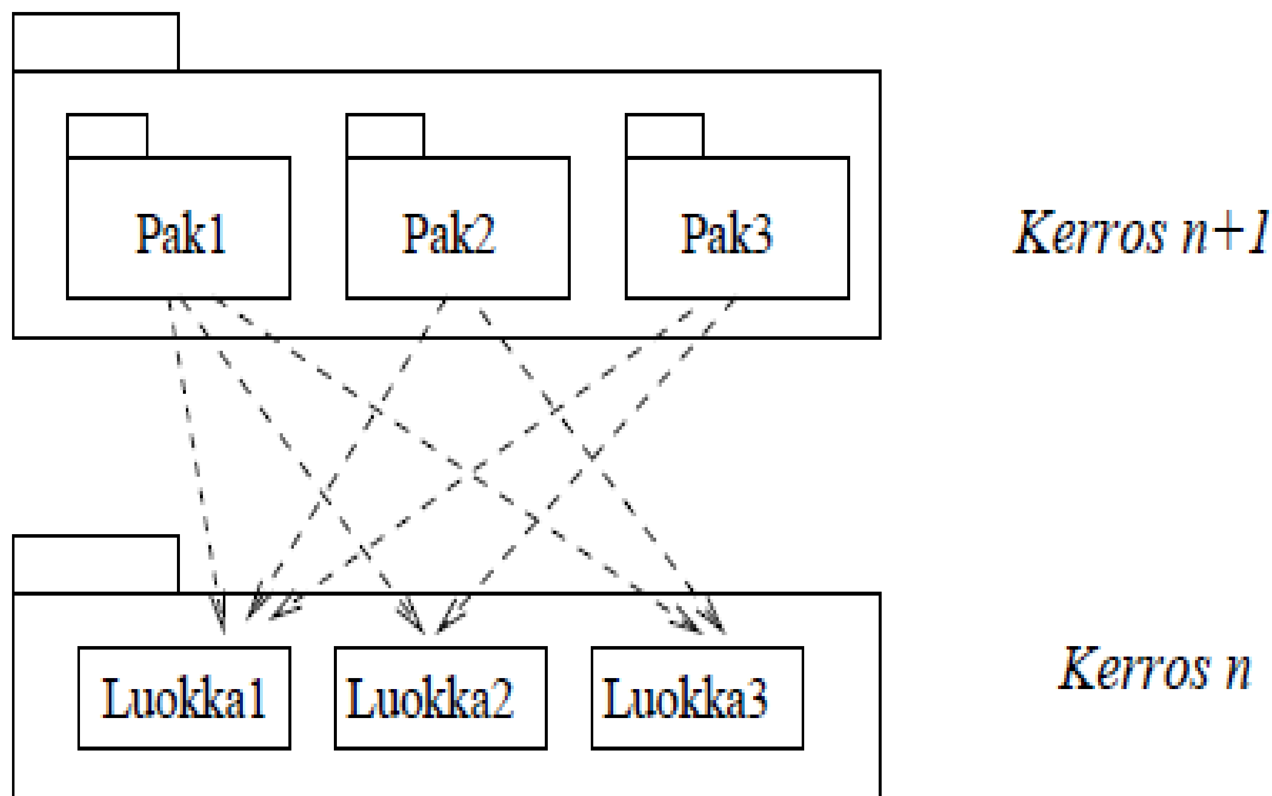
- Kerroksittaisuus *helpottaa ylläpitoa*
  - Kerroksen sisältöä voi muuttaa vapaasti jos sen palvelurajapinta eli muille kerroksille näkyvät osat säilyvät muuttumattomina
  - Sama pätee tietysti mihin tahansa komponenttiin
- Jos kerroksen palvelurajapintaan tehdään muutoksia, aiheuttavat muutokset y*lläpitotoimenpiteitä ainoastaan ylemmän kerroksen* riippuvuuksia omaavin kerroksiin
  - Esim. käyttöliittymän muutokset eivät vaikuta sovelluslogiikkaan tai tallennuspalveluihin
- Sovelluslogiikan riippumattomuus käyttöliittymästä helpottaa ohjelman siirtämistä uusille alustoille, esim. toimimaan www-selaimen kautta
- Alimpien kerroksien palveluja, kuten lokiin kirjoitusta tai tallennuspalvelu-kerrosta voidaan ehkä uusiokäyttää myös muissa sovelluksissa
- *Ylemmät kerrokset* voivat toimia *korkeammalla abstraktiotasolla*
  - Esim. hyvin tehty tallennuspalvelukerros kätkee tietokannan käsittelyn muilta kerroksilta: sovelluslogiikan tasolla voidaan ajatella kaikki olioina
  - Kaikkien ohjelmoijien ei tarvitse ymmärtää kaikkia detaljeja, osa voi keskittyä tietokantaan, osa käyttöliittymiin, osa sovelluslogiikkaan

Ei pelkkiä kerroksia...

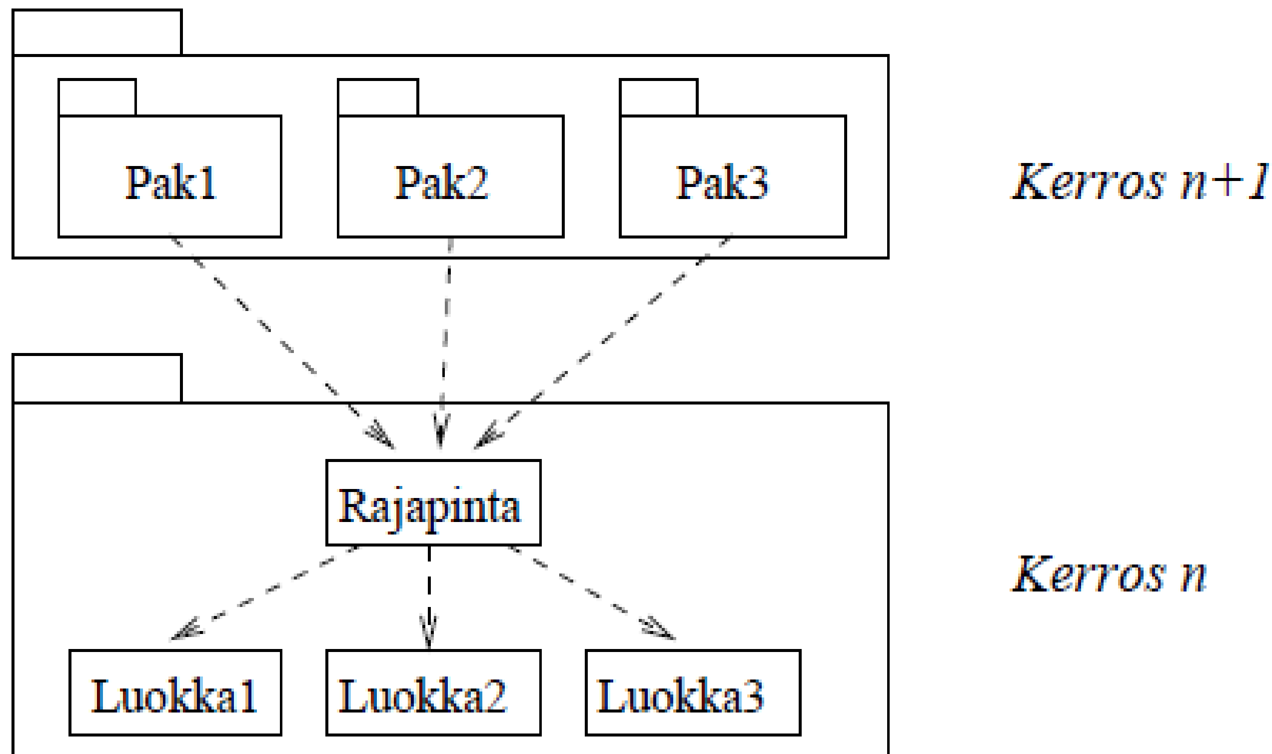
- Myös kerrosten sisällä ohjelman loogisesti toisiinsa liittyvät komponentit kannattaa ryhmitellä omiksi kokonaisuuksiksi joka voidaan UML:ssa kuvata pakkauksena
- Yksittäisistä komponenteista kannattaa tehdä mahdollisimman *yhtenäisiä* toiminnallisuudeltaan
  - eli sellaisia, joiden osat kytkeytyvät tiiviisti toisiinsa ja palvelevat ainoastaan yhtä selkeästi eroteltua tehtäväkokonaisuutta
- Samalla pyrkimyksenä on, että erilliset komponentit ovat mahdollisimman *löyhästi kytkettyjä* toisiinsa
  - komponenttien välisiä riippuvuuksia pyritään minimoimaan
- Ohjelman jakautuminen mahdollisimman riippumattomiin komponentteihin eristää koodiin ja suunnitelmaan tehtävien muutosten vaikutukset mahdollisimman pienelle alueelle, eli ainoastaan riippuvuuden omaaviin komponentteihin
- Tämä helpottaa ohjelman ylläpitoa ja tekee sen laajentamisen helpommaksi
- Selkeä jakautuminen komponentteihin myös helpottaa työn jakamista suunnittelu- ja ohjelmointivaiheessa

Kerroksellisuus ei riitä

- Pelkkä kerroksittaisuus ei tee ohjelman arkkitehtuurista automaattisesti hyvää.
- Alla tilanne, missä kerroksen *n+1* kolmella alipaketilla on kullakin paljon riippuvuuksia kerroksen *n* sisäisiin komponenttiin
- Esim. muutos kerroksen *n* luokkaan 1 aiheuttaa nyt muutoksen hyvin moneen ylemmän kerroksen pakkaukseen



- **Kerrosten välille** kannattaa määritellä selkeä **rajapinta**
- Yksi tapa toteuttaa rajapinta on luoda kerroksen sisälle erillinen *rajapintaolio,* jonka kautta ulkoiset yhteydet tapahtuvat
    - Tätä periaatetta sanotaan *fasaadiksi* (engl. facade pattern)
- Alla luotu rajapintaolio kerrokselle *n*. Kommunikointi kerroksen kanssa tapahtuu rajapintaolion kautta
    - ylemmän kerroksen riippuvuudet kohdistuvat rajapintaolioon
    - muutos esim. luokkaan 1 ei vaikuta kerroksen *n+1* komponentteihin
    - ainoat muutokset on tehtävä rajapintaolion sisäiseen toteutukseen



Käyttöliittymän ja sovelluslogiikan erottaminen

- Kerrosarkkitehtuurin ylimpänä kerroksena on yleensä käyttöliittymä
- Yleensä pidetään järkevänä, että **ohjelman sovelluslogiikka on täysin erotettu käyttöliittymästä**
  - Asia tietysti riippuu myös sovelluksen luonteesta ja oletetusta käyttöajasta
  - Sovelluslogiikan erottaminen lisää koodin määrää, joten jos kyseessä ”kertakäyttösovellus”, ei ylimääräinen vaiva ehkä kannata
- Käytännössä tämä tarkoittaa kahta asiaa:
  - **Sovelluksen palveluja toteuttavilla olioilla** (mitkä suunnitellaan seuraavassa luvussa) **ei ole suoraa yhteyttä käyttöliittymän olioihin**, joita ovat esim. Java-ohjelmissa Swing-komponentit, kuten menut, painikkeet ja tekstikentät
  - Eli sovelluslogiikan oliot eivät esim. suoraan kirjoita mitään ruudulle
  - **Käyttöliittymän toteuttavat oliot eivät sisällä ollenkaan ohjelman sovelluslogiikkaa**
  - Käyttöliittymäoliot ainoastaan piirtävät käyttöliittymäkomponentit ruudulle, välittävät käyttäjän komennot eteenpäin sovelluslogiikalle ja heijastavat sovellusolioiden tilaa käyttäjille

Käyttöliittymän ja sovelluslogiikan erottaminen

- Käytännössä erottelu tehdään liittämällä käyttöliittymän ja sovellusalueen olioiden väliin erillisiä komponentteja, jotka koordinoivat käyttäjän komentojen aiheuttamien toimenpiteiden suoritusta sovelluslogiikassa
- Erottelun pohjana on Ivar Jacosonin kehittämä idea oliotyyppien jaoittelusta kolmeen osaan, rajapintaolioihin (boundary), ohjausolioihin (control) ja sisältöolioihin (entity)
- Käyttöliittymän (eli rajapintaolioiden) ja sovelluslogiikan (eli sisältöolioiden) yhdistävät **ohjausoliot**
- *Käyttöliittymä ei siis ole suoraan yhteydessä sovelluslogiikkaan luokkiin,* vaan ainoastaan välittää käyttäjien komentoja ohjausolioille, jotka huolehtivat sovelluslogiikan olioiden hyödyntämisestä
  - Perjantaina esiteltävä Larmanin *ohjausperiaate* on käytännössä sama kuin tämä Jacobsonin esittelemä ajatus
- Huom: idea ohjausolioista on hiukan sukua ns. *MVC-periaatteelle,* tarkalleen ottaen kyse ei kuitenkaan ole täysin samasta asiasta

Käyttöliittymän ja sovelluslogiikan erottaminen

- Sovellusalueen dataan tulevat muutokset tulee kyetä näyttämään myös käyttöliittymään
- Jyrkässä kerrosarkkitehtuuriperiaatteessa palvelupyyntöjen pitäisi aina kulkea ylhäältä alaspäin, eli muuttuneiden tietojen välitys käyttöliittymälle aiheuttaa hankaluuksia kerroksellisuuden suhteen
- Sovellusalueen tietojen muutosten välittäminen käyttöliittymälle hoidetaan oikeaoppisesti käyttäen ns. *tarkkailijaperiaatetta* (engl. observer pattern)
  - tätä periaatetta saatetaan käsitellä hieman ensi viikolla
  - varsinaisesti asia kuuluu kurssille ohjelmistoarkkitehtuurit, vaikka se olisi syytä tietää jo ohjelmoinnin harjoitustyötä tehtäessä
- Tarkkailijaperiaatteessakaan sovelluslogiikka ei tunne suoraan mitään käyttöliittymään liittyvää
  - Käyttöliittymä tuntee ainoastaan rajapintoja jotka ovat rekisteröineet itsensä sovelluslogiikalle
  - Tarkkailijarajapinnan takana voi olla mitä vaan, esim. jokin käyttöliittymäkomponentti