Luentomoniste kurssille

# Ohjelmistojen mallintaminen

Matti Luukkainen ja Harri Laine
Tietojenkäsittelytieteen laitos
Helsingin Yliopisto

3. joulukuuta 2009

# Esipuhe

Käsissäsi on Ohjelmistojen mallintaminen -kurssin luentomonisteen ensimmäinen versio. Idea monisteen kirjoittamisesta syntyi toukokuussa 2009 laitoksen kahvihuoneessa. Kurssilla (ja sen hieman erinimisellä edeltäjällä) tilanne oli vuosikausia se, että vaikeaselkoisen ja kalliin kurssikirjan takia opiskelijat joutuivat tulemaan toimeen lähes pelkästään luentokalvoilla.

Markkinoilla on runsaasti hyviä kurssin aihepiiriä käsitteleviä kirjoja. Mikään tarjolla olevista kirjoista ei kuitenkaan sovi riittävän hyvin kurssille. Useat kirjat ovat hyvin laajoja ja tarkoitettu huomattavasti pidemmille opintojaksoille. Joissain kirjoissa taas oletettu kohdeyleisö on opinnoissaan huomattavasti keskimääräistä ensimmäisen syksyn opiskelijaa pidemmällä.

Kurssin varhaisella, ennen kevättä 2005 pidetyllä edeltäjällä, *Johdatus sovellussuunnitteluun* -kurssilla oli ainoinaan käytössä Harri Laineen kirjoittama luentomoniste. Alkuperäinen idea olikin päivittää vanha Johdatus sovellussuunnitteluun -kurssin moniste vastaamaan esim. uuden UML-standardin esitysmuotoa.

Aikaansaatu lopputulos sisältää osia vanhasta monisteesta, mutta suurin osa tekstistä on täysin uutta. Uutena tässä monisteessa on mm. kolme UML-kaaviotyyppiä: tilakaavio, aktivitettikaavio ja pakettikaavio, joita vanhassa monisteessa ei käsitelty ollenkaan. Uutta on myös laajahko UML:n soveltamista käsittelevä osa, eli luvut 5 ja 6.

Moniste on tehty LaTeX-ladontajärjestelmällä. Osa kuvista on piirretty Xfig-ohjelmalla ja osa taas Visual Paradigm -mallinnustyökalulla. Harri Laine vastasi luvuista 1.2 ja 2 ja 3.1-3.6 ja Matti Luukkainen lopuista sekä ulkoasun editoinnista.

Monistetta on tarkoitus kehittää jatkossa ja kaikki palaute otetaan kiitollisuudella vastaan. Jo tässä vaiheessa kannattaa pahoitella kirjoitusvrheistä, joita monisteesta varmasti löytyy siitä huolimatta, että useita henkilöitä on ollut oikolukemassa tekstiä.

# Sisältö

# 1 Johdanto ohjelmistotuotantoon

## 1.1 Ohjelmistotuotantoprosessi

Pieniä, kymmenien tai satojen koodirivien kokoisia yhden ihmisen tekemiä ohjelmia voi tehdä miten haluaa. Varsinkaan aloittelevat ohjelmoijat eivät useinkaan suunnittele ohjelmia etukäteen. Useamman hengen projekteissa tuotettujen suurempien ohjelmistojen tekemisessä on pakko käyttää enemmän systematiikkaa. Systemaattiset menetelmät ovat toki hyödyksi myös pienempien ohjelmien tekemisessä.

Ohjelmiston systemaattinen tekeminen, eli ohjelmiston tuotantoprosessi sisältää useita erilaisia vaiheita (ks. esim. [20]). Eri vaiheista tuotetaan dokumentteja, joista selviää esim. ohjelman haluttu toiminnallisuus tai ohjelman rakenne. Dokumentit toimivat ohjeena suunnittelijoille, ohjelmoijille sekä ohjelmiston ylläpitäjille.

### 1.1.1 Vaatimusanalyysi ja -määrittely

*Vaatimusanalyysin ja -määrittelyn* (engl. requirement analysis) aikana kartoitetaan ohjelman tulevien käyttäjien tai tilaajan kanssa se, mitä toiminnallisuutta ohjelmaan halutaan. Ohjelman toiminnalle siis asetetaan asiakkaan haluamat vaatimukset. Tämän lisäksi kartoitetaan ohjelman toimintaympäristön ja toteutusteknologian järjestelmälle asettamat rajoitteet.

Esimerkiksi yliopiston opetushallintojärjestelmälle asetettuja toiminnallisia vaatimuksia ovat

- opetushallinto voi syöttää kurssin tiedot järjestelmään
- opiskelija voi ilmoittautua valitsemalleen kurssille
- opettaja voi syöttää opiskelijan suoritustiedot
- opettaja voi tulostaa kurssin tulokset

Järjestelmälle asetettuja toimintaympäristön rajoitteita voisivat taas olla seuraavat:

- kurssien tiedot talletetaan jo olemassa olevaan tietokannan hallintajärjestelmään
- järjestelmää käytetään www-selaimen kautta
- toteutus tapahtuu Java-kielellä
- järjestelmän on kyettävä käsittelemään maksimissaan 100 ilmoittautumista minuutissa

Vaatimusten määrittelyn pitäisi tapahtua yhdessä ohjelmiston asiakkaan kanssa, jotta ohjelmiston toiminnallisuus saataisiin halutun kaltaiseksi.

Järjestelmän määrittelyssä pyritään vastaamaan kysymykseen *mitä* järjestelmän toiminnallisuudelta halutaan asiakkaan näkökulmasta. Vaatimusmäärittelyssä ei yleensä puututa siihen *miten* haluttu toiminnallisuus toteutetaan teknisellä tasolla. Toteutusratkaisuihin

ei siis oteta kantaa lukuunottamatta tiettyjä toteutukselle asetettuja reunaehtoja, kuten esim. tietyn tietokannan käyttö tai järjestelmän käytettävyys www-selaimella.

Vaatumismäärittelyvaiheessa on tärkeää ymmärtää ohjelman toimintaympäristö mahdollisimman hyvin. Ymmärryksen saavuttamiseksi toimintaympäristön käsitteitä ja niiden suhteita analysoidaan. Opetushallintojärjestelmän käsitteitä ovat esim. seuraavat:

- kurssi
- opettaja
- oppilas
- arvosana
- ...

Käsitteistä ja niiden suhteista voidaan luoda malli. Mallin tekeminen edesauttaa ohjelman kehittäjää ymmärtämään ongelma-aluetta syvällisemmin. Malli tulee myös olemaan hyödyksi ohjelmiston kehityksen myöhemmissä vaiheissa. Mallit laaditaan usein käyttämällä graafisia kaavioita. *Unified modeling language eli UML* [9] on standardoitu joukko erilaisia diagrammityyppejä, jotka sopivat ohjelmistokehityksen eri vaiheisiin.

Ohjelman vaatimukset kirjataan *määrittelydokumenttiin* (engl. requirement specification), joka toimii ohjeena ohjelmiston tuotantoprosessin myöhemmille vaiheelle. Koska määrittelydokumentti kertoo, mitä ohjelman toiminnallisuudelta vaaditaan, testataan valmista järjestelmää yleensä määrittelydokumentin asettamia vaatimuksia vastaan. Eli testauksessa tarkastetaan toimiiko järjestelmä siten kuin sen haluttiin toimivan.

### 1.1.2 Suunnittelu

Määrittelyn jälkeinen vaihe on ohjelmiston suunnittelu. Suunnittelu jakautuu yleensä muutamaan alivaiheeseen. Ensin suunnitellaan *arkkitehtuuri*, eli ohjelmiston jakautuminen korkean tason rakenneosasiksi. Esim. kurssihallintojärjestelmässä tietokanta ja varsinainen sovelluslogiikka sijaitsee erillisellä koneella. Asiakkaat taas ovat www-selaimia, jotka toimivat käyttäjien omalla koneella.

Ohjelman suuremmat toiminnalliset kokonaisuudet voidaan vielä jakaa alikomponentteihin, eli itsenäisemmin toimiviin osiin, jotka voidaan antaa esim. yksittäisten suunnittelijoiden ja ohjelmoijien vastuulle. Komponenttien välille suunnitellaan yleensä selkeät *rajapinnat* (engl. interface), jotka mahdollistavat erillään suunniteltujen ja toteutettujen komponenttien yhteenliitettävyyden.

Ohjelman komponentit toteutetaan yleensä joukkona erilaisia olioita, joista osa on itse suunniteltuja ja osa saadaan valmiina ohjelmointiympäristön tarjoamista kirjastoista (esim. Java:ssa Java API:n kautta on käytettävissä runsain määrin valmiita luokkia). Ennen toteutusta komponentin sisäinen rakenne yleensä suunnitellaan jollain tasolla, eli jos järjestelmä kehitetään oliomenetelmillä, tapahtuu *oliosuunnittelu* (engl. object design).

Tuloksena tästä on *suunnitteludokumentti* (engl. design document), joka toimii oppaana toteuttajille sekä ohjelman tuleville ylläpitäjille.

Myös suunnitteluvaiheessa käytetään UML:n erilaisia kaavioita suunnittelun apuna sekä dokumentoitaessa valittuja suunnitteluratkaisuja.

### 1.1.3 Toteutus, testaus ja ylläpito

Toteutusvaiheessa suunnitteludokumentin mukainen järjestelmä toteutetaan valitulla ohjelmointikielellä. Usein oliosuunnittelu ja toteutus kulkevat käsi kädessä. Toteutusvaiheessa tehdään myös usein havaintoja, jotka aiheuttavat muutoksia suunnitteluvaiheen päätöksiin.

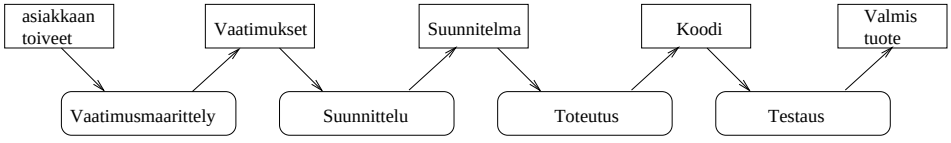
Toteutuksen yhteydessä ja sen jälkeen järjestelmää testataan. Testausta on monentasoista. *Yksikkötestauksessa* (engl. unit testing) tutkitaan yksittäisten metodien ja luokkien toimintaa. Yksikkötestauksen tekee usein testattavan komponentin ohjelmoija. Kun erikseen ohjelmoidut komponentit (eli luokat tai luokkakokoelmat) yhdistetään, suoritetaan *integrointitestaus* (engl. integration testing), jossa varmistetaan erillisten komponenttien yhteentoimivuus. Integrointitestauksessa testauksen kohteena ovat erityisesti suunnitteluvaiheessa erillisten komponenttien välille määritellyt rajapinnat. *Järjestelmätestauksessa* (engl. system testing) testataan järjestelmää kokonaisuutena ja verrataan, että se toimii niinkuin vaatimusdokumentissa määriteltiin.

Ohjelmistoprosessi ei pääty yleensä siihen, että tuote saadaan asiakkaalle. Ohjelmaa on myös ylläpidettävä. Ylläpidossa korjataan ohjelmasta löytyneitä virheitä ja usein myös lisäillään ohjelmaan uutta toiminnallisuutta tai suunnitellaan ja kirjoitetaan joitain osia kokonaan uudelleen.

### 1.1.4 Vesiputousmalli

Ohjelmistoja on perinteisesti tehty vaihe vaiheelta etenevän *vesiputousmallin* (engl. waterfall model) mukaan.

Vesiputousmallissa (ks kuva 1) suoritetaan ensin vaatimusmäärittely, jonka seurauksena kirjoitetaan vaatimusdokumentti, johon pyritään kokoamaan kaikki ohjelmalle osoitettavat vaatimukset mahdollisimman tarkasti dokumentoituna. Määrittelyvaiheen päätteeksi vaatimusdokumentti jäädytetään. Jäädytettyä vaatimusmäärittelyä käytetään usein ohjelman kehittämisen vievien resurssien arvioinnin perustana ja myös sopimus ohjelman hinnasta saatetaan tehdä vaatimusmäärittelyn pohjalta.




Kuva 1: Vesiputousmallin mukaisesti etenevä ohjelmistokehitys

Vaatimusmäärittelyä seuraa suunnitteluvaihe, joka myös dokumentoidaan tarkoin. Pääsääntöisesti suunnitteluvaiheen aikana ei enää tehdä muutoksia määrittelyyn. Joskus tä-

mäkin on tarpeen. Suunnittelu pyritään tekemään niin täydellisenä, että ohjelmointivaiheessa ei enää ole tarvetta muuttaa suunnitelmia.

Suunnittelun jälkeen toteutetaan ohjelman yksittäiset komponentit ja tehdään niille yksikkötestaus. Tämän jälkeen erilliset komponentit liitetään yhteen eli integroidaan ja suoritetaan integrointitestaus. Tyypillisesti tässä vaiheessa löydetään paljon virheitä, jotka johtuvat pahimmassa tapauksessa suunnitteluvirheistä ja aiheuttavat uutta suunnittelutyötä.

Integroinnin jälkeen ohjelmalle tehdään järjestelmätestaus, eli testataan, että ohjelmisto kokonaisuutena toimii niin kuin määrittelydokumentissa on määritelty.

Vesiputousmalli on monella tapaa ongelmallinen. Mallin toimivuus perustuu siihen oletukseen, että ohjelman vaatimukset pystytään määrittelemään täydellisesti ennen kuin suunnittelu ja ohjelmointi alkaa. Näin ei useinkaan ole. On lähes mahdotonta, että asiakkaat pystyisivät tyhjentävästi ilmaisemaan kaikki ohjelmalle asettamansa vaatimukset. Vähintäänkin riski sille, että ohjelma on käytettävyydeltään huono, on erittäin suuri. Usein käy myös niin, että vaikka ohjelman vaatimukset olisivat kunnossa vaatimusten laatimishetkellä, muuttuu toimintaympäristö (tapahtuu esim. yritysfuusio) ohjelman kehitysaikana niin ratkaisevasti, että valmistuessaan ohjelma on vanhentunut. Hyvin yleistä on myös se, että vasta käyttäessään valmista ohjelmaa asiakkaat alkavat ymmärtää, mitä he olisivat ohjelmalta halunneet.

### 1.1.5 Ketterä ohjelmistokehitys

Vesiputousmallin heikkoudet ovat johtaneet viime vuosina yleistyneiden *ketterien* (engl. agile) ohjelmiston kehitysmenetelmien kehittelyyn ja käyttöönottoon (ks. esim. [15]).

Ketterissä menetelmistä lähdetään oletuksesta, että vaatimuksia ei voi tyhjentävästi määritellä ohjelmistokehitysprosessin alussa. Koska näin ei voida tehdä, ei sitä edes yritetä vaan pyritään toimimaan niin, että ohjelmista saadaan toimivia jatkuvasti muuttuvista vaatimuksista huolimatta.

Ketterä ohjelmistokehitys etenee yleensä siten, että ensin kartoitetaan pääpiirteissään ohjelman vaatimuksia ja ehkä hahmotellaan järjestelmän arkkitehtuuri pääpiirteittäin. Tämän jälkeen suoritetaan useita *iteraatioita*[1], joiden kunkin aikana järjestelmään valitaan suunniteltavaksi ja toteutettavaksi osa järjestelmän vaatimuksista. Vaatimukset voivat tarkentua koko prosessin ajan.

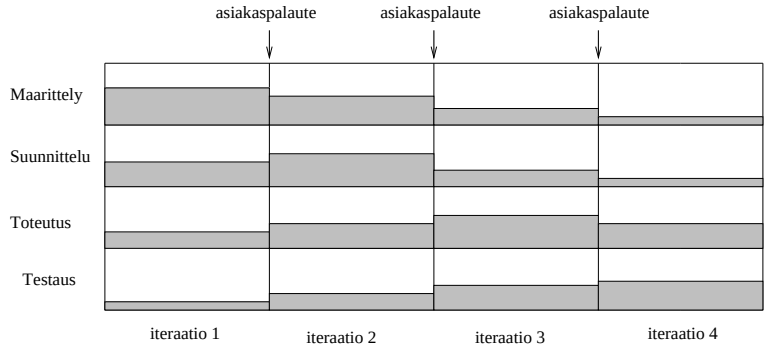
Yksittäinen iteraatio, joka voi olla kestoltaan muutamia viikkoja tai korkeintaan pari kuukautta, siis lisää järjestelmään pienen osan koko järjestelmän toivotusta toiminnallisuudesta. Tyypillisesti tärkeimmät ja toteutuksen kannalta haasteellisimmat ja riskialttiimmat toiminnallisuudet toteutetaan ensimmäisillä iteraatioilla. Yksi iteraatio sisältää toteutettavaksi valittujen vaatimusten tarkennuksen, suunnittelun, toteutuksen sekä testauksen.

Jokainen iteraatio tuottaa toimivan ja toteutettujen ominaisuuksien kannalta testatun järjestelmän. Asiakas pääsee testaamaan järjestelmää jokaisen iteraation jälkeen. Tällöin voi-

[1] Iteraatioista käytetään yleisesti myös nimityksiä sykli tai sprintti.

daan jo aikaisessa vaiheessa todeta, onko kehitystyö etenemässä oikeaan suuntaan ja vaatimuksia voidaan tarvittaessa tarkentaa ja lisätä.

Kuva 2 havainnollistaa ketterän ohjelmistokehityksen luonnetta. Jokainen iteraatio siis sisältää määrittelyä, suunnittelua, ohjelmointia ja testausta ja jokaisen iteraation jälkeen saadaan asiakkaalta palautetta siitä, onko kehitystyö etenemässä oikeaan suuntaan.




Kuva 2: Iteratiivisesti etenevä ketterä ohjelmistokehitys

Ketterässä ohjelmistokehityksessä dokumentointi ei ole yleensä niin keskeisessä osassa kuin perinteisissä menetelmissä. Vähäisemmän dokumentaation sijaan testauksella ja ns. jatkuvalla integroinnilla on hyvin suuri merkitys. Yleensä pyritään siihen, että järjestelmään lisättävät uudet komponentit testataan välittömästi ja pyritään heti integroimaan kokonaisuuteen, tästä työskentelytavasta käytetään nimitystä *jatuva integrointi* (engl. continuous integration). Näin uusia versioita järjestelmästä syntyy jopa päivittäin iteraatioiden toteutusvaiheiden aikana.[2] Uusien komponenttien toimiminen pyritään varmistamaan perinpohjaisella automaattisella testauksella. Joskus jopa "testataan ensin", eli jo ennen ennen uuden komponentin toteuttamista ohjelmoidaan komponentin toimintaa testaavat testitapaukset. Testitapausten valmistuttua toteutetaan komponentti ja siinä vaiheessa kun komponentti läpäisee testitapaukset, se integroidaan muuhun kokonaisuuteen.

Erilaisia ketteriä ohjelmistokehitysmenetelmiä on olemassa lukuisia, tunnetuimmat lienevät Extreme programming eli XP [2] ja Scrum [19]. Myös laajalti käytössä oleva Rational Unified Process eli RUP -prosessimalli [14] painottaa ketterien menetelmien tapaan iteratiivista ohjelmistokehitystä.

Vesiputousmalli ja ketterät menetelmät ovat kaksi ääripäätä ohjelmiston tuotantoprosessia ajatellessa. Usein käytännössä sovelletaan jotain näiden ääripäiden välimuotoa.

## 1.2 Ohjelmiston mallintaminen

Ohjelmiston kehittämisen yhteydessä ohjelmistosta on usein tarpeen laatia kuvauksia. Kuvaukset määrittelevät ohjelmistosta esimerkiksi mitä ohjelmisto tekee tai millainen ohjelmisto on rakenteeltaan ja missä ympäristössä ohjelmisto toimii. Kuvaukset toimivat kehitystyötä ohjaavina ohjelmiston piirustuksina samaan tapaan kuin esimerkiksi omakotitalon

[2] Vesiputousmallissa komponenttien integraatio taas tapahtuu yleensä vasta toteutusvaiheen lopussa.

piirustukset talonrakennusprojektissa. Taloa rakennettaessa siitä laaditaan aluksi monenlaisia piirustuksia, esimerkiksi arkkitehtuurisuunnitelma, useita rakennesuunnitelmia, sähkösuunnitelma, lvi-suunnitelma, jne. Näiden piirustusten muoto on tarkoin säädelty. Säädökset määrittelevät mitä piirustuksia tarvitaan. Piirustuksissa käytettävä esitystapa on standardoitu ja standardeja on noudatettava. Talojen kuvaamiseen käytettävät dokumenttityypit ja niiden esitystavat ovat syntyneet pitkän kehityksen ja viranomaismääräysten tuloksena.

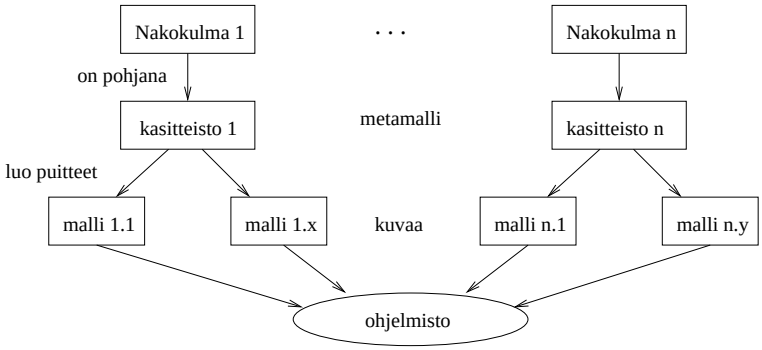
Ohjelmistojen kohdalla tilanne on erilainen. Ei ole yleisesti hyväksyttyä vakiintunutta kokoelmaa dokumenttityyppejä eikä niissä käytettäviä esitystapoja. Viranomaiset eivät säätele dokumenttien muotoa ja määrää. Kuitenkin ohjelmistokehityksessä tarvittavien kuvausten määrä on jopa laajempi kuin rakennusten kohdalla. Rakennuksista ei yleensä, ainakaan ennen rakentamista, kuvata siitä, kuinka niissä asutaan, millaisia asukkaat ovat ja miten he taloa hyödyntävät. Ohjelmistojen kuvauksiin tuollaisetkin asiat usein kuuluvat. Talot ovat myös, päinvastoin kuin ohjelmat, luonteeltaan passiivisia.

Rakennuksista tiedetään yleisesti, että niissä on ainakin seinät, katto ja lattia. Talo jakautuu huoneisiin, ja huoneesta toiseen pääsee jonkin yleensä seinässä olevan aukon kautta. Ohjelmistojen kohdalla tilanne on hankalampi. Ei ole itsestään selvää millaisista rakenneosista ohjelmisto koostuu ja miten osat liittyvät toisiinsa. Ohjelmisto on abstrakti kohde, ohjeisto, jonka tarkoituksena on saada tietokone suorittamaan tietojenkäsittelyoperaatioita, joiden tulos on ohjelmiston käyttäjälle jollain tavalla hyödyksi. Käsitteistö, jonka avulla ohjelmisto voidaan hahmottaa, on muuttunut laitteistokehityksen, ohjelmointikielten ja ohjelmointivälineiden, kehitysmenetelmien ja sovellusalueiden myötä.

Ohjelmistosta laadittavien kuvausten tarkoituksena on välittää tietoa ohjelmistosta sen kehittämiseen ja mahdollisesti myös käyttöön osallistuvien henkilöiden välillä. Kuvaamisen pohjana on aina jokin sovittu käsitteistö, joka kiinnittää sen, millaisia kuvattavia asioita pitäisi hahmottaa ja mitä näistä on oleellista esittää. Käsitteistöt ovat muodostuneet abstrahoinnin tuloksena. Kiinnitettyyn käsitteistöön perustuvaa kuvausta ohjelmistosta kutsutaan ohjelmiston malliksi ja mallin perustana olevaa käsitteistöä metamalliksi. Mallin laadinnassa käytettävän käsitteistön voi määritellä tapauskohtaisesti, mutta helpommin ymmärrettäviä kuvauksia saa perustamalla ne vakiintuneeseen tai standardoituun käsitteistöön. Ohjelmistoa voidaan tarkastella erilaisista näkökulmista (engl. view). Kuhunkin näkökulmaan liittyy oma käsitteistönsä.

Jos ohjelmisto ajatellaan abstraktina kohteena, niin myös jollakin ohjelmointikielellä kirjoitettu ohjelma on tietystä näkökulmasta laadittu kuvaus ohjelmistosta. Esimerkiksi Java-kielellä kirjoitettu ohjelma on Java-kielen määrittämään käsitteistöön perustuva malli ohjelmistosta. Java-kielen määrittely toimii tällöin metamallina, jonka mukaisesti ohjelmistoa kuvataan. Java-kielen tapauksessa on tarkoin määritelty paitsi käsitteet myös mallin ts. ohjelman lähdekoodin esitystapa.

Ohjelmistoihin liittyy paljon yksityiskohtia. Yhdestä näkökulmasta ne voivat olla kiinnostavia ja tarpeellista kuvata, jostain toisesta näkökulmasta ne ovat merkityksettömiä. Usein puhutaan kuvaamisen eri abstraktiotasoista "ylemmän tason"tarjotessa käsitteitä abstraktimpaan, vähemmän konkreettiseen tai yksityiskohtaiseen kuvaamiseen kuin alem-



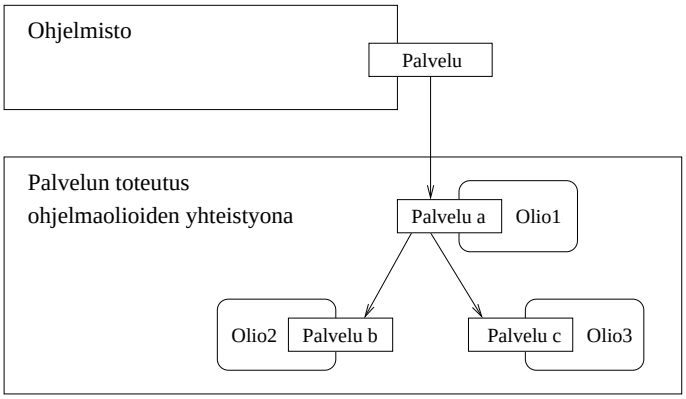

Kuva 3: Mallit ja metamallit ohjelmiston kuvaamisessa

man. Java-kielellä kirjoitettu kuvaus on yksityiskohtainen. Jos kuvauksessa esitetään vain ohjelman luokat ja näiden väliset riippuvuudet on yksityiskohtia vähemmän ja ohjelman toiminta kokonaisuudessaan jää kuvaamatta. Joissain tilanteissa tällainen kuvaus voi kuitenkin olla riittävä. Tällaisen kuvauksen pohjana oleva metamalli määrittelee suppeamman käsitteistön kuin Java-kielen määrittely. Koska kuvaus sisältää vähemmän yksityiskohtia, sitä voidaan pitää korkeammalla abstraktiotasolla olevana. Toisesta näkökulmasta tarkasteltuna ohjelman kuvaamiseen voi riittää ohjelman syöttö- ja tulostietojen määrittely ja selvitys millaisten sääntöjen mukaan ohjelma tuottaa syöttötiedoista tulostietoja. Abstraktiotaso näiden kuvausten yhteydessä on kuitenkin varsin epämääräinen käsite. Jossakin tilanteissa, esimerkiksi tietojen kuvaamisessa, on pyritty määrittelemään näkökulmat siten, että saataisiin aikaan selkeä tasojako (laiteläheinen, ohjelmoijan näkymä, ohjelmariippumaton näkymä). Toisissa tilanteissa näkökulmien välillä ei ole välttämättä eroa abstraktioiden yksityiskohtaisuudessa tai konkreettisuudessa, vaan pelkästään mielenkiinnon kohteiden valinnasta.

Jotta kuvauksia olisi helppo lukea tarvitaan käsitteiden lisäksi myös yhteinen kuvaustekniikka. Ohjelmistojen kuvaamiseksi on esitelty useita käsitteistojä ja vielä useampia kuvaustekniikoita. Paljous johtuu osittain siitä, että järjestelmiä on kovin monenlaisia ja kovin erilaisilla sovellusalueilla. Esimerkiksi tietoliikennejärjestelmää kuvattaessa tärkeät kuvattavat asiat ovat jossakin määrin erilaisia kuin vaikkapa kuvankäsittelyjärjestelmää tai asiakaspalvelujärjestelmää kuvattaessa. Järjestelmien abstraktisuus ja vakiintuneen teoriapohjan puute tuottavat uusia malleja.

Tällä kurssilla tarkastellaan ohjelmistojen *oliopohjaista* (engl. object oriented) kuvaamista. Oliopohjaisuus sinällään on metatason malli (malli mallista), jonka mukaan *minkä tahansa järjestelmän voidaan katsoa muodostuvan olioista, jotka yhteistyössä toimien ja toistensa palveluja hyväksikäyttäen tuottavat järjestelmän palvelut.* Järjestelmä itsekin on tällöin olio. Tällaista metamallia voidaan luontevasti soveltaa useisiin tilanteisiin. Liiketoiminnassa on tyypillisesti käytetty organisaatioyksikköjakoa. Organisaatioyksiköllä on tietyt vastuut ja se tuottaa palveluja muille organisaatioyksiköille. Se on siis ilmiselvä olio. Yritys pyörii sen organisaatioyksiköiden yhteistyön ansiosta. Olio-ohjelmoinnissa tietojärjestelmän tarjoamia tietojenkäsittelypalveluita tuotetaan tietokoneen sisuksissa toimivien tietojenkäsittelyolioiden yhteistyön tuloksena (ks. kuva 4). Oliot toimivat osina isommissa ja monimut-

kaisemmissa olioissa, jotka tuottavat omat laajemmat ja monimutkaisemmat palvelunsa alkeellisempien olioiden yksinkertaisempia palveluita hyödyntäen.




Kuva 4: Tietojärjestelmän palvelun tuottaminen

Koska oliopohjaisuus on metamalli, sitä voidaan soveltaa ohjelmistojenkin kuvaamiseen eri tavoin. Eri menetelmät esittelevät erilaisia soveltamistapoja eli malleja järjestelmien kuvaamiseen oliopohjaisesti. Menetelmät esittelevät myös kuvaustekniikoita, joita tulisi käyttää. Kuvaustekniikat ovat usein ainakin osittain graafisia, koska graafisen kuvauksen avulla on tekstikuvausta helpommin saatavissa kokonaiskuva kuvattavasta kohteesta.

## 1.3 UML

UML (Unified Modeling Language) (katso esim. [4, 9]) -kuvaustekniikka kehitettiin 90-luvulla yhdistämällä kolmen tunnetuimman ns. ensimmäisen sukupolven oliomenetelmän käyttämät kuvaustekniikat: Boochin oliotekniikka [3], Rumbaughn ja kumppaneiden OMT [18] ja Jacobsonin OOSE [11]. UML-tekniikan peruskehittämisestä on huolehtinut Rational Software -yritys, jonka osakkaina em. tekniikkojen kehittäjät olivat. Nykyään Rational Software on IBM:n tytäryhtiö. Oliotekniikoiden käyttöä edistämään perustettu Object Management Group -yhdistys (OMG) valitsi UML:n omaksi kuvausstandardikseen loppuvuodesta 1997. OMG:ssä ovat jäseninä useat merkittävät ohjelmistoyritykset, joten valinta oli varsin merkittävä askel oliopohjaisten kuvausten yhtenäistämiseksi. Nykyään käytännössä kaikki tietokoneavusteisten suunnitteluvälineiden (Computer Aided Software Engineering eli CASE-välineiden) tuottajat ovat muuttaneet välineitään UML-tekniikkaan perustuviksi. UML:n kehityksestä vastaa nykyään OMG.

UML määrittelee joukon kaaviotyyppejä käytettäväksi ohjelmiston kuvaamiseen. Tiettyä kaaviotyyppiä voi käyttää useassa eri tilanteessa. Esimerkiksi luokkakaaviota (engl. class diagram) voi käyttää ohjelman rakenteen kuvaamiseen, järjestelmän osajärjestelmäjaon kuvaamiseen, järjestelmän tietosisällön kuvaamiseen, jne. UML ei määrittele missä tilanteissa tiettyä tekniikkaa pitäisi käyttää. Tämä on kehittämismenetelmien tehtävä. UML on laajennettava kuvauskieli, eli kielen käyttäjällä on mahdollisuus lisätä siihen omia piirteitä.

UML-standardia on täydennetty vuosien kuluessa ja nykyinen käytössä oleva versio on 2.2. Kuten kaikki komiteoiden aikaansaannokset, on UML todella laaja standardi. Versiossa 2.2

on määritelty 13 erilaista kaaviotyyppiä. Seuraavassa on lueteltu nykyisen UML-standardin mukaiset kaaviotyypit. Tällä kurssilla käsiteltävien kaavioiden yhteydessä on mainittu myös joitain kaaviotyypin käyttökohteita.

- käyttötapauskaavio (engl. use case diagram) kuvaa ohjelmiston palvelut
- luokkakaavio (engl. class diagram) kuvaa ohjelman tai sen jonkin komponentin luokkarakenteen ja luokkien väliset suhteet
- oliokaavio (engl. object diagram) kuvaa ohjelman luokkien instanssien konfiguraatiota tietyllä suoritushetkellä
- sekvenssikaavio (engl. sequence diagram) kuvaa olioiden suoritusaikaista yhteistyötä
- kommunikaatiokaavio (engl. communication diagram) on toinen tapa kuvata olioiden suoritusaikaista yhteistyötä
- pakkauskaavio (engl. packet diagram) ryhmittelee ohjelman rakenteen suurempiin kokonaisuuksiin
- tilakaavio (engl. state diagram) kuvaa yksittäisen olion käyttäytymisen olion elinkaaren aikana
- aktiviteettikaavio (engl activity diagram) kuvaa kontrollin kulkua ohjelman suorituksessa
- komponenttikaavio (engl. component diagram)
- sijoittelukaavio (engl. deployment diagram)
- ajoituskaavio (engl. timing diagram)
- koostekaavio (engl. composite structure diagram)
- kokoava vuorovaikutuskaavio (engl. interaction overview diagram)

Kaavioiden syntaksi (eli oikeaoppinen piirtotekniikka) ja osin myös semantiikka (eli mitä kaavio merkitsee) on määritelty standardissa hyvin tarkasti. Standardin eri versioiden välillä on pieniä eroja ja esim. vanhemmissa kirjoissa ja dokumenteissa sekä www-lähteissä saattaa olla kaavioita, jotka eivät täysin noudata UML 2.0 -standardia.

UML:n käyttö voi tapahtua joko tarkoin syntaksia noudattaen tai luonnosmaisesti. Kun pyritään noudattamaan syntaksia tarkasti, piirretään kaaviot usein käyttäen tietokoneavusteisia suunnitteluvälineitä (CASE-välineitä)[3]. Tarkoista malleista voidaan sopivia työkaluja hyödyntäen jopa generoida koodirunko toteutukseen. Luonnosmaisemmassa käytössä taas ei pyritä noudattamaan standardinmukaista syntaksia aivan orjallisesti, vaan pääpaino voi olla vaikkapa järjestelmän ydintoiminnallisuuden havainnollistamisessa nopeasti valkotaululle tai paperille luonnosteltavina kuvina.

Kurssilla UML:n käytössä fokus on luonnosmaisuuden ja oikeaoppisen syntaksinkäytön järkevä balanssi. Standardin laajuuden vuoksi tutustumme vain tärkeimpiin kaaviotyyppeihin ja niistäkin vain keskeisimpiin piirteisiin. Ohjelmistokehityksen kannalta kaikkien UML-kaavioiden syntaksin osaamista tärkeämpi asia onkin oppia soveltamaan eri tekniikoita siten, että mallinnukseen käytettävä aika ja vaiva ovat järkevässä suhteessa siitä saatavaan hyötyyn. Eli voidaan todeta, että UML tarjaa ainoastaan mallinnusnotaation,

[3]CASE on lyhenne sanoista Computer Aided Software Engineering.

eli yhteiset merkintätavat kaikille mallintajille. Haaste onkin siinä, miten UML:ää käytetään järkevästi. Usein mallinnuksessa lopputuloksena syntyviä UML-malleja tärkeämpää onkin itse mallinnusprosessi, sillä se pakottaa mallintajan ajattelemaan systemaattisesti ja miettimään mallinnettavan ongelman kaikkia puolia.

Yleisesti voidaan todeta, että mallinnuksessa ideana on tehdä abstraktio jostain tarkastelun alla olevasta mielenkiintoisesta kohteesta. Abstraktiolla tarkoitetaan yksinkertaistettua mallia, joka kuitenkin sisältää riittävän määrän kiinnostavia yksityiskohtia. Mallinnusta on siis hyvin monen tasoista, ylimalkaisista luonnoksista aina hyvinkin detaljoituun mallinnukseen asti. Kuten jo edellisessä luvussa todettiin, mallinnusta tehdään myös monesta eri näkökulmasta. UML:n 13 erilaista kaaviotyyppiä tarjoavat kukin mahdollisuuden hieman erilaiseen näkökulman esilletuomiseen. Yhdellä kaaviotyypillä ei siis pystytä todennäköisesti mallintamaan tiettyä kohdetta tyhjentävästi vaan on käytettävä useita eri näkökulmia tarjoavia kaavioita. Hyvin tyypillistä on esim. kuvata järjestelmän rakenneosien staattista luonnetta luokkakaaviolla. Näin ei kuitenkaan saada vielä minkäänlaista kuvaa siitä, miten järjestelmä toimii, eli tarvitaan lisäksi esim. sekvenssikaavioita kuvaamaan järjestelmän rakenneosasten yhteistoimintaa.

Mallinnuksen detaljirikkauden ja näkökulmien lisäksi mallinnuksen luonteeseen vaikuttaa myös se, missä ohjelmistotuotantoprosessin vaiheessa mallinnus tapahtuu. Vaatimusmäärittelyn aikana kuvaillaan *ongelman kohdealueen* (engl. problem domain) käsitteitä ja niiden suhteita esim. luokkakaavioilla. Siirryttäessä suunnitteluvaiheeseen, vaatimusmäärittelyn aikana tehdyt mallit tarkentuvat ja mallinnettavat käsitteet ovat tyypillisesti luokkia, jotka tullaan ohjelmoimaan toteutusvaiheessa. Kohdealueen käsitteitä (esim. kurssihallintajärjestelmässä opiskelija, opettaja, kurssi, ilmoittautuminen, ...) kuvaavien luokkien lisäksi suunnittelutasolla malleihin tuodaan mukaan puhtaasti teknisen tason luokkia, joiden tehtävä on esim. toimia oliosäiliöinä ja ohjausolioina sekä hoitaa käyttöliittymä ja tietokantayhteyksiä. Siirryttäessä vaatimusmäärittelystä suunnitteluun, malleista tulee ohjelmoijan kannalta konkreettisempia, eli niiden abstraktiotaso laskee.

Malli siis tarkentuu ja muuttuu toteutusläheisemmäksi siirryttäessä määrittelystä suunnitteluun. Jos noudatetaan ketterän ohjelmistokehityksen iteratiivista lähestymistapaa, tarkentuu malli myös siinä mielessä, että ensimmäisessä iteraatiossa malli sisältää ainoastaan ydintoiminnallisuuden. Tämän jälkeen malliin lisätään jokaisessa iteraatiossa lisää toiminnallisuutta.

Mainittakoon vielä, että usein opiskelijoita vaivaa harhakuvitelma, jonka mukaan esim. suunnitteludokumentiksi riittää pelkkä UML-kaavio. Näin ei asianlaita tietenkään ole. UML-kaaviot tuovat usein hyödyllistä lisävalaistusta asioihin, mutta tekstuaalinen selitys on aina avainasemassa. Syntyvän dokumentaation sijaan mallinnuksen ehkä tärkein hyöty onkin itse mallinnusprosessi, joka auttaa mallintajia ymmärtämään paremmin ongelma-aluetta sekä mahdollistaa esim. ennen toteutusta tapahtuvan eri suunnitteluratkaisujen keskinäisen vertailun.

Vaikka tällä kurssilla käsitellään pelkkää UML:ää, se ei suinkaan ole ainoa mahdollinen mallinnusnotaatio. Usein mallinnuksessa käytetään myös vapaamuotoisia tai esim. yrityskohtaisessa käytössä olevia "laatikoista ja viivoista"muodostuvia kuvauksia.

## 1.4 Monisteen rakenne

Tässä luvussa on ollut paljon asiaa ja onkin todennäköistä, että lukija on hämmentynyt kaikesta vastaan tulleesta käsitteistöstä. Onkin toivottavaa, että luet tämän luvun uudelleen myöhemmässä vaiheessa, jotta kurssin aikana käsiteltävät yksityiskohdat nivoutuisivat paremmin suurempaan kokonaisuuteen.

Monisteen loppuosan rakenne on seuraavanlainen.

Luvuissa 2, 3 ja 4 käsitellään tärkeimpiä UML-kaavioita, vaatimusmäärittelyyn soveltuvia *käyttötapauskaavioita*, käsitteistön tai ohjelmaluokkien suhteita kuvailevia *luokkakaavioita* sekä olioiden yhteistoiminnan kuvaamiseen tarkoitettuja *sekvenssikaavioita* ja *kommunikaatiokaavioita*. Nämä neljä ovat UML:n tärkeimmät ja eniten käytetyt kaaviotyypit.

Luvuissa 5 ja 6 esitellään yksi mahdollinen tapa, miten UML-kaavioita voidaan hyödyntää ohjelmiston kehittämisessä. Tarkastelemme konkreettista esimerkkiä, yksinkertaista kirjaston tietojärjestelmää, ja etenemme vaatimusmäärittelystä suunnitteluun ja toteutukseen asti. Luvun 4 aikana esitellään ohjelman korkeamman tason rakenteiden kuvailuun sopiva UML:n *pakkauskaavio*. Luvun aikana opitaan lisää myös luokka- ja sekvenssikaavioiden käytöstä.

Luvussa 7 esitellään kahden hieman vähemmän käytetyn, mutta joissain tilanteissa hyödyllisen UML-kaaviotyypin, *tilakaavioiden* ja *aktiviteettikaavioiden* esittely.

UML:n kaavioista komponenttikaaviot (engl. component diagram, sijoittelukaaviot (engl. deployment diagram), ajoituskaaviot (engl. timing diagram), koostekaaviot (engl. composite structure diagram) ja kokoavat vuorovaikutuskaaviot (engl. interaction overview diagram) on siis rajoitettu kurssin ulkopuolelle. Pois jäävistä kaaviotyypeistä komponenttikaaviot lienevät hyödyllisimmät. Niihin palattaneen kurssilla *Ohjelmistotuotanto*.

# 2 Käyttötapausmalli

Ohjelmisto tarjoaa käyttäjilleen palveluita, jotka perustuvat järjestelmän tietosisältöön. Ohjelmiston toiminta voidaan kuvata määrittelemällä millaisia palveluita ohjelmisto tarjoaa. *Käyttötapausmalli* (engl. use case model) kuvaa ohjelmiston toimintaa käyttäjän kannalta tarkasteltuna.

Käyttäjä voi olla henkilö, toinen järjestelmä, laite, yms. taho, joka on järjestelmän ulkopuolella, mutta kuitenkin tekemisissä sen kanssa. Tällainen taho voi toimia

- tiedon tuottajana järjestelmään tai
- tiedon hyödyntäjä

Käyttäjän asemasta käytetään joskus termiä *sidosryhmä* (engl. stake holder). Käyttäjä voi olla järjestelmän kanssa tekemisissä suoraan tai epäsuorasti.

Käyttäjien tunnistaminen on ensimmäinen tehtävä järjestelmän palveluja määriteltäessä. Käyttäjien löytämiseksi voidaan esittää kysymykset:

- kuka / mikä saa tulosteita järjestelmästä?
- kuka / mikä toimittaa tietoa järjestelmään?
- kuka käyttää järjestelmää?
- mihin muihin järjestelmiin kehitettävä järjestelmä on yhteydessä?

Näiden kysymysten perusteella tunnistetaan *roolit*, joissa eri tahot toimivat suhteessa järjestelmään. Nämä roolit määritellään käyttäjiksi.

Esimerkiksi tietojenkäsittelytieteen laitoksen ilmoittautumisjärjestelmän käyttäjärooleja ovat

- opiskelija
- opettaja
- opetushallinto
- suunnittelija
- laitoksen johtoryhmä
- tilahallintojärjestelmä
- henkilöstöhallintojärjestelmä

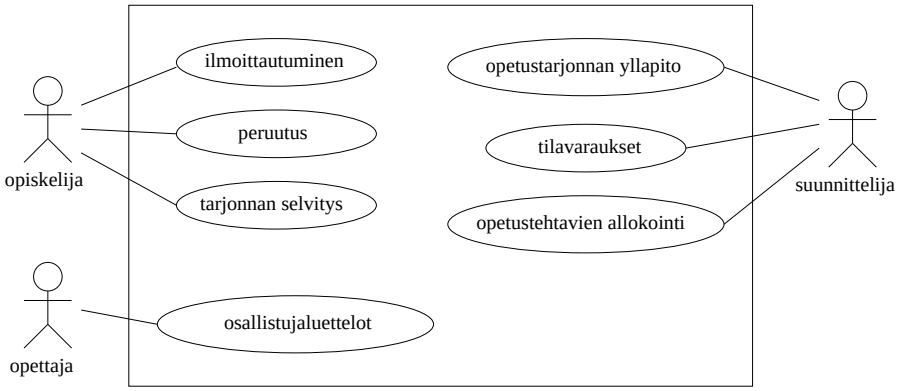
Yllä olevista käyttäjärooleista laitoksen johtoryhmä ei ole suorassa yhteydessä järjestelmään. Tilahallintojärjestelmä ja henkilöstöhallintajärjestelmä ovat erillisiä järjestelmiä, joiden palveluita laitoksen opetustietojärjestelmä hyödyntää.

*Käyttötapauksella* (engl. use case) mallinnetaan käyttäjän ohjelmiston avulla suorittamaa tehtävää eli *tapaa käyttää ohjelmistoa*. Luonteva tapa henkilökäyttäjien yhteydessä on kytkeä käyttötapaukset käyttäjän työtehtäviin. Tällöin ohjelmiston tarjoama palvelu tukee

käyttäjää hänen työtehtävässään. Käyttötapauksen laajuus on keskeinen tekijä mallin ymmärrettävyyden ja hallittavuuden kannalta. Käyttötapaukset eivät saisi määritellä liian laajoja eivätkä myöskään liian suppeita tehtäväkokonaisuuksia. Keskikokoisissa ohjelmistohankkeissa käyttötapauksia on tyypillisesti muutamia kymmeniä ja isommissa hankkeissa jopa satoja.

Pääperiaatteena käyttötapauksia määriteltäessä on, että yhden käyttötapauksen tulisi muodostaa looginen kokonaisuus, jolla on sekä selvä *lähtökohta* että *merkityksen omaava lopputulos*. Lähtökohtana eli herätteenä on tapahtuma tai tarve, joka käynnistää käyttötapauksen. Tapahtuma voi olla joko käyttäjän itsensä, jonkin ulkopuolisen tekijän tai järjestelmän aiheuttama. Se voi olla myös ajan kulumisesta aiheutuva.

Kuvassa 5 on hahmoteltu opetukseen ilmoittautumisista vastaavan järjestelmän käyttötapauksia. Opiskelijan käyttötapauksia ovat *ilmoittautuminen, peruutus* ja *tarjonnan selvitys*. Tarjonnan selvitys käyttötapauksen lähtökohtana on opiskelijan tarve saada tietoa tarjottavasta opetuksesta luennoista ja harjoituksista. Lopputuloksena opiskelija saa tarvitsemansa tiedot. Ilmoittautuminen käyttötapauksen lähtökohtana on opiskelijan tarve saada opiskelupaikka kurssilta. Lopputuloksena opiskelija kirjataan kurssin osallistujaksi. Peruutus käyttötapauksen lähtökohtana on tarve päästä pois kurssilta. Lopputuloksena on tilanne, jossa opiskelija ei ole enää kirjattu kurssin osallistujaksi.




Kuva 5: Ilmoittautumisjärjestelmän käyttötapauksia

On huomionarvoista että tietty taho voi toimia järjestelmän suhteen useammassa erilaisessa käyttäjäroolissa. Esim. laitoksella laskareita pitävä vielä itsekin opiskeleva sivutoiminen opettaja voi toimia ilmoittautumisjärjestelmää käyttäessään välillä opettajan roolissa ja välillä taas opiskelijan roolissa.

Termiä käyttötapaus käytetään yleisesti sekä tyyppitason käsitteenä että ilmentymätason käsitteenä. Tässä yhteydessä noudatetaan UML:n käytäntöä ja käyttötapaus nähdään tehtävätyyppinä. Tietyllä tyypillä on yleensä useita ilmentymiä eli esimerkkiskenarioita. Käyttötapauksen ilmoittautuminen esimerkkiskenariona voisi olla tapaus *Kalle Kenkkunen ilmoittautuu kurssin Ohjelmistojen mallintaminen harjoitusryhmään 2 syksylle 2009.*

Käyttötapauksella on *käyttäjä* (engl. actor), joka käyttötapauksessa toimii vuorovaikutteisesti järjestelmän (ohjelmiston) kanssa toteuttaakseen tavoitteensa. Käyttäjän toiminta on

syötteiden antamista ja palautteen saamista. Usein käyttäjä on ihminen, mutta käyttäjä voi olla myös ulkoinen järjestelmä. Käyttötapaukseen liittyy aina *tavoite eli asia, jonka käyttäjä haluaa saada aikaan käyttötapauksen avulla.* Kuvan 5 opiskelijan käyttötapausten tavoitteena on kussakin tapauksessa saada lähtökohtana mainittu tarve tyydytettyä. Yhdellä käyttötapauksella voi olla myös useita käyttäjiä.

Käyttötapausta kuvattaessa kuvataan käyttäjän ja järjestelmän välinen vuoropuhelu. Kun käyttötapausmallia käytetään määrittelyvaiheessa ja vaatimusanalyysin apuvälineenä kuvauksessa tulisi esittää vain vuoropuhelun *sisältö*, mitä käyttötapauksessa tapahtuu ja mistä asioista käyttäjä ja järjestelmä vaihtavat tietoa. Käyttöliittymää *ei tässä vaiheessa vielä kiinnitetä*, joskin joitain periaatteita saattaa olla tarpeen hahmotella, jotta käyttäjät ymmärtäisivät, mistä on kyse. Miten vuorovaikutus käytännössä tapahtuu, tulisi kiinnittää vasta suunnitteluvaiheessa. Käyttötapauksesta voidaan tällöin laatia yksityiskohtaisempi suunnittelutason kuvaus.

Käyttötapaus kuvataan esittämällä vuoropuhelun *peruskulku.* Mahdolliset käyttötapauksen kulkua muuttavat poikkeustilanteet voidaan määritellä erillisinä poikkeavina kulkuina tai käyttötapausta täydentävinä käyttötapauksina. Käyttötapaukseen voi liittyä vaatimuksia koskien suojausta, vasteaikoja, yms.

Käyttötapauksen kuvaamiseen ei ole mitään formaalia tekniikkaa, vaan tapaukset kuvataan luonnollisen kielen tekstinä. Vaikka mitään standardoitua tapaa kuvata käyttötapauksia ei ole olemassa, jokaisessa ohjelmistoprojektissa kannattaa kuitenkin sopia yhteinen tapa, miten käyttötapaukset kuvataan.

## 2.1 Esimerkkejä

Seuraavassa esimerkkejä ilmoittautumisjärjestelmään liittyen.

**Kurssille Ilmoittautuminen**

- *Käyttäjä:* opiskelija
- *Tavoite:* saada kurssipaikka
- *Laukaisija:* opiskelijan tarve
- *Käyttötapauksen kulku:* Opiskelija tutkii kurssitarjontaa ja valitsee ohjelmiston esittämästä tarjonnasta kurssin ja ryhmän, tunnistautuu ja aktivoi ilmoittautumistoiminnon. Opiskelija saa kuittauksen ilmoittautumisen onnistumisesta.
- *Poikkeuksellinen toiminta:* Opiskelija ei voi ilmoittautua täynnä olevaan ryhmään. Opiskelija ei voi ilmoittautua, jos hänelle on kirjattu osallistumiseste.
- *Lisähuomioita:* 4 ruuhkahuippua vuodessa, noin 400 ilmoittautumista ensimmäisen 10 minuutin aikana ilmoittautumisen alkamisesta. Muulloin tapahtumia on vähän
- *Esimerkkitapaus:* NN ilmoittautuu kurssin XML-metakieli / kevät 2010 harjoitusryhmään 3.

**Ilmoittautumisen peruminen**

- *Käyttäjä:* opiskelija
- *Tavoite:* perua ilmoittautuminen, välttää sanktiot
- *Käyttötapauksen kulku:* Opiskelija tunnistautuu ja valitsee järjestelmän näyttämistä omista ilmoittautumisistaan peruttavan kohteen sekä aktivoi peruutustoiminnon. Järjestelmä ilmoittaa operaation onnistumisesta.
- *Esimerkkitapaus:* NN peruu ilmoittautumisensa kurssin XML-metakieli / kevät 2010 harjoitusryhmään 3.

Edellisissä esimerkeissä käyttötapauksen kulku on kuvattu melko abstraktilla tasolla, kiinnittämättä käyttäjän ja järjestelmän välistä interaktiota tarkemmin. Näin toimitaan usein järjestelmän määrittelyn alkuvaiheissa, jotta haluttua toiminnallisuutta ei tule kiinnitettyä liian tarkasti turhain aikaisessa vaiheessa.

Edetessä järjestelmän vaatimuksien kartoittamisesta kohti suunnitteluvaihetta, halutaan käyttötapauksessa usein tuoda esiin tapahtumien kulku tarkemmalla tasolla. Tällöin tapahtumien kulku on tapana esittää numeroituna dialogina käyttäjän ja järjestelmän välillä. Joskus käyttötapauksille on myös tapana määritellä *esiehto* (engl. precondition) ja *jälkiehto* (engl. postcondition). Esiehto kuvaa asioiden tilan, jonka oletetaan olevan voimassa käyttötapauksen alussa. Jälkiehto taas kuvaa asioiden tilan, minkä oletetaan olevan voimassa käyttötapauksen onnistuneen läpikäymisen jälkeen.

Seuraavassa *Kurssille ilmoittautuminen* -käyttötapaus, jossa tapahtumien kulku ja järjestys on tuotu tarkemmin esiin. Tällä kertaa myös käyttötapauksen esi- ja jälkiehdot on ilmaistu.

**Kurssille Ilmoittautuminen, tarkennus**

- *Käyttäjä:* opiskelija
- *Tavoite:* saada kurssipaikka
- *Laukaisija:* opiskelijan tarve
- *Esiehto:* opiskelija on ilmoittautunut kuluvalla lukukaudella läsnäolevaksi
- *Jälkiehto:* opiskelija on lisätty haluamansa ryhmän ilmoittautujien listalle
- *Käyttötapauksen kulku:*
    1. Opiskelija aloittaa kurssi-ilmoittautumistoiminnon
    2. Järjestelmä näyttää kurssitarjonnan
    3. Opiskelija tutkii kurssitarjontaa
    4. Opiskeljia valitsee ohjelmiston esittämästä tarjonnasta kurssin ja ryhmän
    5. Järjestelmä pyytää opiskelijaa tunnistautumaan
    6. Opiskelija tunnistautuu ja aktivoi ilmoittautumistoiminnon
    7. Järjestelmä ilmoittaa opiskelijalle ilmoittautumisen onnistumisesta.
- *Poikkeuksellinen toiminta:* Opiskelija ei voi ilmoittautua täynnä olevaan ryhmään. Opiskelija ei voi ilmoittautua, jos hänelle on kirjattu osallistumiseste.

Näin dokumentoituna käyttötapaus kiinnittää jo useita asioita esim. käyttöliittymän suhteen. Kyseessä onkin aiempaa versiota konkreettisempi käyttötapaus.

## 2.2 Yleistykset, sisällytykset ja laajennokset

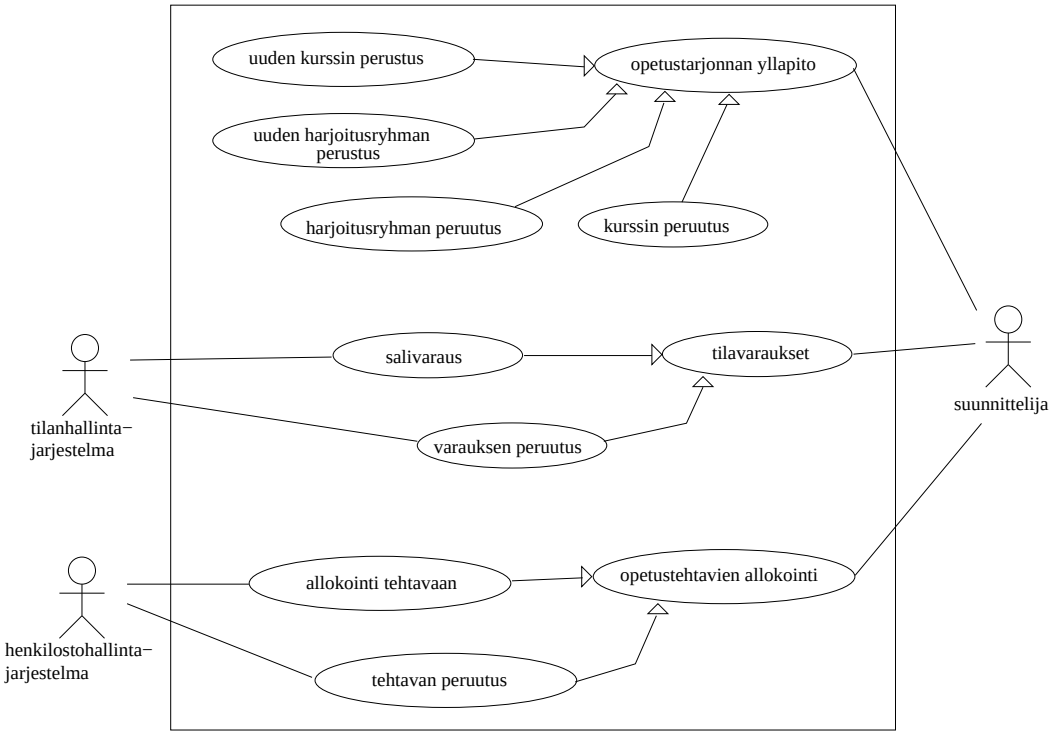
Laajoissa järjestelmissä voidaan lähteä liikkeelle käyttäjien työtehtäviin perustuvista käyttötapauksista. Käyttötapauksia analysoitaessa niistä saatetaan löytää yhteisiä osia, jotka voidaan erottaa omiksi käyttötapauksiksi. Myös erilaiset virhe- ja poikkeustilanteet sekä vaihtoehtoiset kulut käyttötapauksessa voidaan erottaa omiksi erillisiksi käyttötapauksiksi. Näin syntyy riippuvuuksia käyttötapausten välille. Käyttötapauksia voi löytyä myös suuria määriä. Tällöin voi yleiskuvan saamiseksi ohjelmiston toiminnosta olla paikallaan koota yhteen käyttötapauksia ja esittää ne yhtenä *yleistettynä* (eng. generalized) tapauksena.

Kuvassa 5 esitetyistä käyttötapauksista suunnittelijan käyttötapaukset ovat yleistettyjä käyttötapauksia. Esimerkiksi opetustarjonnan ylläpito jakautuu useaan erilliseen tehtävään: uuden kurssin perustaminen, uuden harjoitusryhmän perustaminen, kurssin poistaminen opetustarjonnasta ja harjoitusryhmän peruutus. Ilmoittautumisjärjestelmässä suunnittelijan käyttötapaukset muodostavat oman osajärjestelmänsä ja yksittäiset käyttötapaukset voitaisiin kuvata yksityiskohtaisemmin tämän yhteydessä. Osajärjestelmän käyttötapauskaavio on kuvassa 6. Kuvasta näemme, että erikoistava käyttötapaus, esim. *kurssin peruutus* on yhdistetty nuolella[4] yleistävään käyttötapaukseen *opetustarjonnan ylläpito*. Suunnittelijan osajärjestelmän mallissa on otettu mukaan myös muita järjestelmiä tilahallintojärjestelmä ja henkilöstöhallintojärjestelmä, joiden kanssa suunnittelijan käyttämä osajärjestelmä toimii yhteistyössä.

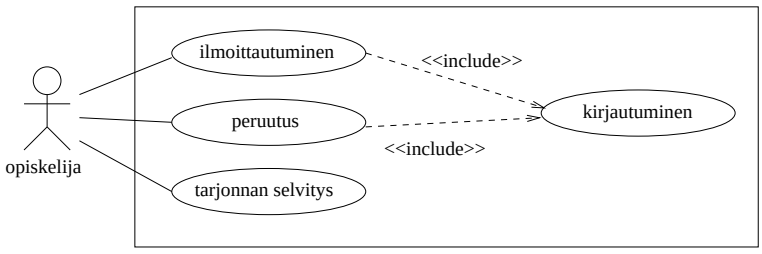
Useat työtehtävät muodostuvat sarjasta peräkkäisiä toimintoja. Käyttäjän tavoite liittyy tällöin usein koko toimintosarjaan eikä yksittäiseen toimintoon vaikka toiminto toisikin tavoitetta jonkin verran lähemmäs. Käyttötapauksena kannattaa tällöin tarkastella koko sarjaa eikä sen askelia. Kun käyttötapauksia analysoidaan, voidaan löytää usealle tapaukselle yhteisiä osia. Tällöin nämä voi kuvata omina apukäyttötapauksinaan, vaikka ne olisivat toimintosarjan osan kaltaisia. Esimerkiksi opiskelijan ilmoittautumiseen ja ilmoittautumisen perumiseen liittyy molempiin opiskelijan tunnistautuminen. Tätä varten voidaan määritellä käyttötapaukseksi *kirjautuminen*. Yhdenkään opiskelijan ensisijaisena tavoitteena ei liene kirjautua järjestelmään. Tästä syystä kirjautumista ei voi pitää pääkäyttötapauksena vaan täydentävänä, pääkäyttötapaukseen *sisällytettävänä* (engl. include) aputapauksena. Kuvassa 7 käyttötapaukset *ilmoittautuminen* ja *peruutus* sisällyttävät käyttötapauksen *kirjautuminen* toiminnallisuuden. Sisällytys on merkitty pääkäyttötapauksesta apukäyttötapaukseen kohdistuvalla katkoviivalla johon on liitetty *stereotyyppi* eli tarkenne *«include»*. Huomaa taas miten nuolen pää on piirretty. Se on nyt erilainen kun yleistyksen yhteydessä.

Kuten aiemmin todettiin, kuvataan käyttötapauksessa toiminnan peruskulku. Peruskulkunsa mukaisesti esimerkiksi ilmoittautuminen onnistuu aina. Käytännössä näin ei kuitenkaan välttämättä ole. Ilmoittautumiseen liittyy sääntöjä, jotka saattavat estää kurssille

[4]Huomioi että nuolenpään piirtotapa erikoistamissuhteessa on "avoin kolmio". Kuten kohta huomaamme, on nuolenpäiden piirtotapa määritelty UML:ssä tarkasti.

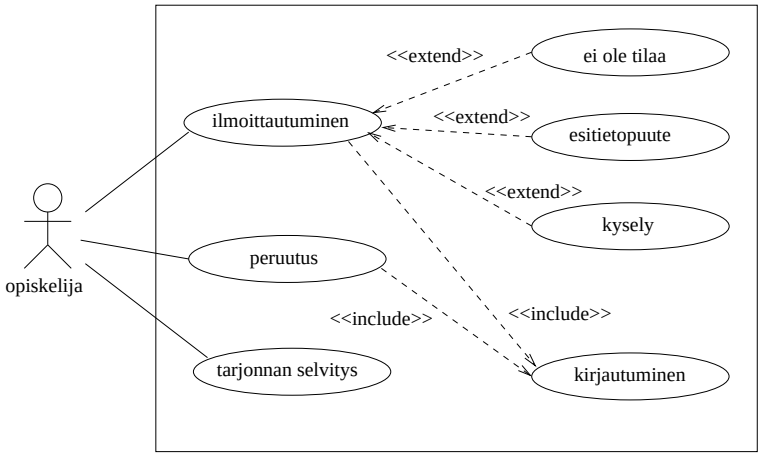



Kuva 6: Suunnittelijan osajärjestelmän käyttötapauskaavio




Kuva 7: Kirjautuminen apukäyttötapauksena ilmoittautumisen ja perumisen osana

pääsyn. Tällaiset poikkeustapaukset voidaan kuvata käyttötapauksen *laajennoksina* (engl. extend). Kuvassa 8 on ilmoittautumisen laajennoksina esitetty poikkeustapaukset *ei-ole-tilaa* ja *esitietopuute*, jotka kumpikin estävät ilmoittautumisen. Laajennoksena *kysely* kuvataan tilanne, jossa kurssille pääsemiseksi on vastattava kurssikohtaisiin hakukysymyksiin. Laajennos on merkitty laajentavasta käyttötapauksesta pääkäyttötapaukseen kohdistuvalla katkoviivalla johon on liitetty stereotyyppi *«extend»*.




Kuva 8: Käyttötapauksen poikkeuksia

Sisällytyksen ja laajennoksen käyttö hämmentää joskus kokeneitakin käyttötapausten määrittelijöitä. Ero näiden käytössä on siinä, että käytettäessä sisällytystä apukäyttötapaus liitetaan *aina* pääkäyttötapaukseen. Eli esimerkissämme kirjautuminen suoritetaan aina peruutuksen ja ilmoittautumisen yhteydessä. Laajennos taas liittyy pääkäyttötapaukseen vain *tarvittaessa*, eli esim. *ei-ole-tilaa* -käyttötapaus ilmenee ilmoittautumisen yhteydessä ainoastaan joskus. Huomaa kuvista, että nuoli sisällytyksen ja laajennoksen yhteydessä kulkee eri suuntiin. Sisällytyksessä nuoli kulkee apukäyttötapaukseen päin, kun taas laajennoksessa pääkäyttötapaukseen päin.

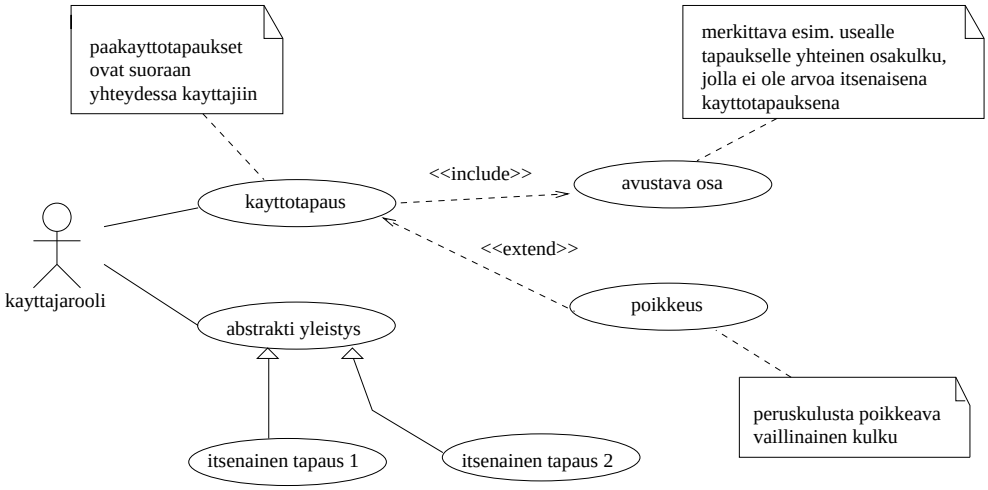
Useat käyttötapausasiantuntijat (mm. ehkä tunnetuimman käyttötapausoppaan kirjoittaja Alistair Cockburn [7]) ovat sitä mieltä, että sekavuuksien välttämiseksi käyttötapausten laajennusta ei välttämättä kannata käyttää, vaan parempi tapa ilmaista esim. poikkeukset on dokumentoida ne käyttötapauksen tekstuaalisen esityksen yhteydessä.

## 2.3 Yhteenveto käyttötapauskaavioiden merkinnöistä

UML määrittelee käyttötapausmallille graafisen esitysmuodon. Tätä muotoa on käytetty yllä esitetyissä kuvissa. Käyttötapausmallin symbolit on esitetty kootusti kuvassa 9. Kuvassa on käytetty kommentteja sisältäviä UML-elementtejä. Kommentit ovat käytettävissä kaikissa UML:n kaaviotyypeissä. Poikkeus on valinnainen täydentävä tai muuntava osa jotain laajempaa kokonaisuutta. Itsenäiset tapaukset ovat itsenäisiä peruskäyttötapauksia.

Käyttötapauskaavion informaatiosisältö on melko vähäinen. Lähinnä kaavio tarjoaa yleiskuvan järjestelmän käyttäjistä ja palveluista. Käyttötapauskaaviosta on enemmän hyö-

työ lähinnä silloin kun on esitettävä käyttötapausten välisiä riippuvuuksia (poikkeuksia ja osia). On kuitenkin tärkeä muistaa, että *käyttötapausmallin merkittävin osa on kunkin käyttötapauksen sisällön määrittelevä sanallinen kuvaus.*




Kuva 9: UML:n käyttötapausmallin graafisen esityksen symbolit

Käyttötapausmalli sopii hyvin vuorovaikutteisten ohjelmistojen kuvaamiseen. Tekstikuvauksen tarkkuustason valinta on usein hankalaa. Malli ei sovellu kovin hyvin järjestelmien välisen yhteistyön kuvaamiseen. Ulkopuolinen järjestelmä nähdään käyttäjänä ja yhteistyötä pitäisi kuvata käyttäjän kannalta vaikka luonnollisemmalta tuntuisi pitää kehitettävää järjestelmää tällaisessa tilanteessa käyttäjänä.

Käytännössä käyttötapauksia kannattaa ryhmitellä joko käyttämällä yleistettyjä käyttötapauksia tai UML-tekniikkaan sisältyviä pakkauksia, joita käsitellään luvussa 6. Ryhmittely voi tapahtuen perustuen käyttötapausten käyttäjiin (esim. opettajan ja opiskelijan käyttötapaukset erikseen) tai järjestelmän toiminnallisuuteen (esim. ilmoittautumiseen ja kurssien arvosteluun liittyvät käyttötapaukset erikseen).

# 3 Luokkakaaviot

*Luokkakaaviolla* (engl. class diagram) kuvataan

- ohjelmiston tai järjestelmän rakenne
- ohjelmiston tietosisältö
- palvelut, joita ohjelmisto ja sen osat kykenevät suorittamaan sekä
- ohjelmiston osien väliset yhteydet

*Oliokaaviolla* (engl. object diagram) havainnollistetaan luokkakaaviota, esimerkiksi olioiden välisiä yhteyksiä.

Oliopohjaisen lähestymistavan mukaisesti ohjelmisto voidaan hahmottaa oliona, joka tarjoaa palveluja käyttäjille (ks. luku 1.2). Järjestelmätasoisten olioiden palvelut toteutuvat järjestelmän sisällä toimivien pienempien olioiden yhteistyön tuloksena. Lähestymistavan lähtökohtana ovat sosiaaliset organisaatiot. Esimerkiksi yrityksen tarjoamat palvelut toteutuvat työntekijöiden palvelujen kautta.

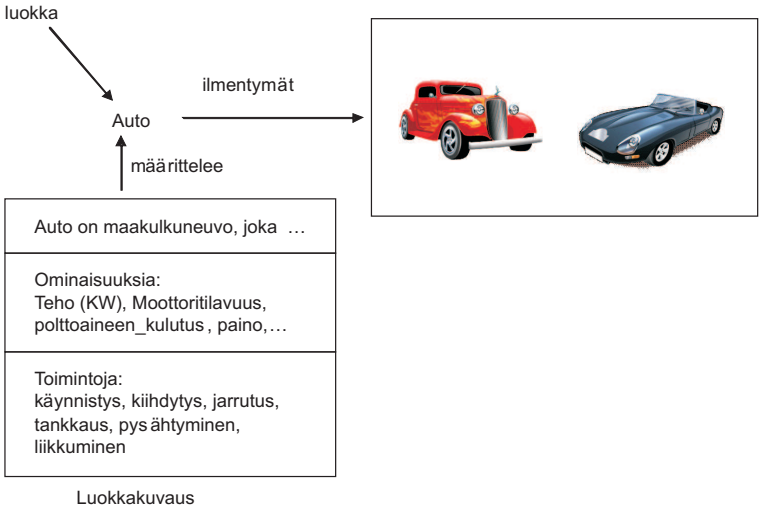
*Oliolla* (engl. object) tarkoitetaan tiedon ja sen käsittelyyn liittyvien palveluiden muodostamaa kokonaisuutta. Olio pitää sisällään tietoja, joita olion palvelut käsittelevät tai hyödyntävät. Oliota voisi verrata henkilöön. Henkilöllä on tietämyksensä, jota ei ulkoapäin tarkastelemalla saa selville. Henkilön tietämystä voi hyödyntää vain kysymällä tai hyödyntämällä palveluja, joita henkilö tietämykseensä perustuen tarjoaa.

Oliomallintaminen perustuu olioiden luokitteluun. Jokainen olio kuuluu johonkin luokkaan ja on toiminnaltaan ja sisällöltään luokkakuvauksessa määritellyn mallin mukainen. Olioluokka tai lyhyemmin pelkkä *luokka* (engl. object class) on siis samankaltaisten olioiden malli. Yksittäisiä olioita tarkastellaan tämän mallin *ilmentyminä* (engl. instance). Kuvassa 10 esimerkki luokasta, luokkakuvauksesta ja luokan ilmentymät reaalimaailmassa. Ohjelmaoliot elävät lyhyen elämänsä tietokoneen uumenissa. Ne voivat simuloida jotain konkreettista reaalimaailman oliota esimerkiksi palkanlaskijaa, opettajaa, opiskelijaa, kirjaston kirjaa, puhelunvälittäjää. Alun perin olio käsite esiteltiinkin simuloinnin yhteydessä. Ohjelmistoja tarkasteltaessa törmätään kuitenkin usein abstrakteihin olioihin, joille ei löydy reaalimaailman suoraa vastinetta.

Luokkakuvaus esitetään jollakin kuvaustekniikalla joko graafisesti tai jäsenneltynä tekstiesityksenä. Olio-ohjelmoinnissa luokkakuvaus esitetään ohjelmointikielellä (ks. kuva 11). Tämä esitys on tarkoitettu tietokoneelle. Lisäksi tarvitaan ihmisille tarkoitettu kuvaus. Kun järjestelmiä toteutettaessa edetään määrittelystä ja suunnittelusta toteutukseen, on luonnollista, että ihmisille tarkoitetut kuvaukset eli suunnitelmat tuotetaan ensin ja vasta sitten koneelle tarkoitettu esitys.

## 3.1 Luokkakuvaus

Luokkakuvauksessa nimetään luokka *kuvaavalla nimellä*. Tarvittaessa määritellään tekstikuvauksena luokan merkitys, eli selvitetään millaisia olioita luokkaan kuuluu. Luokkaku-




Kuva 10: Luokka, luokkakuvaus ja luokan ilmentymät reaalimaailmassa

```
public class elain {
   int elain_numero;
   String laji;
   Color vari;
   float paino;
   public elain(…){…}
   public setPaino(…){…}
   public float getPaino(){…}
   …
}
elain otus = new elain(….);
elain toinen = new elain(….);
```

ilmentymiä

Kuva 11: Luokkakuvaus ohjelmointikielellä

vauksessa määritellään myös luokan ilmentymiin sisältyvät tiedot ja palvelut (eli operaatiot), joita luokan ilmentymät kykenevät suorittamaan tai tarjoamaan.

### 3.1.1 Tietojen määrittely

Luokan ilmentymiin sisältyvät tiedot kuvataan antamalla

- *attribuutti* (engl. attribute), joka on tietoa kuvaava nimi
- *arvomäärittely* (engl. value specification), joka esittää millaisten arvojen avulla tieto esitetään ja
- sanallinen selvitys tiedon merkityksestä ja käytöstä.

Attribuutin arvona voi olla

- yksinkertainen arvo, esimerkiksi kokonaisluku tai merkkijono
- rakenteinen arvo, esimerkiksi osoite, joka jakautuu postiosoitteeseen, postinumeroon ja lähiosoitteeseen
- kokoelma yksinkertaisia tai rakenteisia arvoja, esimerkiksi järjestämätön joukko värejä tai järjestetty joukko koordinaattipareja
- kokoelmia voi olla rakenteeltaan erityyppisiä (taulukko, joukko, järjestetty joukko, lista, tiedosto ...)
- Mitä laajempia ja korkeammalla abstraktiotasolla oliot ovat, sitä laajempia ovat myös niiden tietosisällöt. Tällöin attribuutin arvona voisi tulla kyseeseen tiedosto, tietovirta tai jokin muu laaja tietokokoelma.

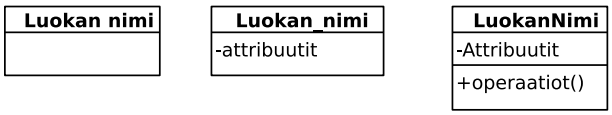
Attribuutin arvona voi olla myös *viittaus olioon* tai kokoelma viittauksia olioihin. Tällaista attribuuttia kutsutaan *olioarvoiseksi*. Olioarvoisen attribuutin olemassaolo tarkoittaa sitä, että on olemassa *yhteys* attribuutin sisältävän olion ja attribuutin arvona olevan olion välillä. Tällaiset yhteydet ovat usein hyvin oleellisia järjestelmän oliorakenteen ymmärtämisen kannalta. Niinpä niiden esittämiseen on olemassa myös oma erillinen tekniikkansa, jota tarkastellaan myöhemmin. Sama asia pitäisi aina kuvata vain kertaalleen. Kaavion tavoitteista riippuu mitä tekniikkaa käytetään. Jos yhteydet ovat merkittäviä ne esitetään eksplisiittisesti. Olioarvoisia attribuutteja on kuitenkin syytä käyttää, jos viitattu olio on perustietotyypin kaltainen. Tällaisia voisivat olla päiväys, kellonaika, henkilötunnus, väri, koordinaatti, jne.

Vaikka attribuuttinimen tulisi mahdollisimman hyvin kuvata sitä ilmiötä, joka attribuuttiin liittyvällä arvolla halutaan esittää, ei nimi yleensä riitä attribuutin määrittelyksi vaan tarvitaan erillinen selvitys attribuutin merkityksestä ja käytöstä. Selvityksessä voidaan käsitellä arvojen tulkintaa, arvojen tarkkuutta, arvojen mittaamista, yms. Arvojen käyttöön ja käyttäytymiseen saattaa myös liittyä erityispiirteitä, joilla on merkitystä kehitettävälle ohjelmistolle. Eräs tällainen piirre on *attribuutin käyttö olioiden ulkoiseen tunnistamiseen.* Olioilla on attribuuttien arvoista riippumaton identiteetti, mutta usein on tarve viitata

olioon myös jonkin siitä tiedetyn ja attribuuttien avulla esitetyn ominaisuuden perusteella, esimerkiksi henkilö-oliot voitaisiin ulkoisesti tunnistaa nimen ja syntymäajan avulla.

Attribuutti voi olla *kiinteäarvoinen* eli sellainen, jonka arvo säilyy samana koko olion eliniän, esimerkiksi henkilön syntymäaika on tällainen. Attribuutti voi myös olla *välttämätön* eli sellainen, jonka arvo ei voi olla tuntematon tai puuttuva. Esimerkiksi henkilön nimi voisi olla välttämätön henkilön attribuutti, mutta osoite ei.

UML-luokkakaaviotekniikassa luokka kuvataan suorakaiteena. Se voidaan esittää suppeassa muodossa, jolloin suorakaiteen sisään kirjoitetaan vain luokan nimi, tai laveassa muodossa, jossa suorakaiteen sisällä näkyvät myös luokan attribuutit sekä mahdollisesti myös palvelut (ks. kuva 12).




Kuva 12: Luokkasymboli erilaajuisena

Attribuutista voidaan UML-luokkakaaviossa esittää

- Nimi (välttämätön)
- Tietotyyppi

  Tietotyyppinä voi käyttää jonkin ohjelmointikielen mukaisia perustietotyyppejä (int, char, double, boolean, string, jne). Tietotyypit voivat olla myös sovellusalue- tai tapauskohtaisia (rahasumma, prosenttiosuus, nimi, jpg-kuva, mpeg-video, jne). Tyyppejä voi määritellä tarpeen mukaan.

- Moniarvoisuus (engl. multiplicity)

  Kokoelma-arvoisten attribuuttien kohdalla kokoelman koko voidaan ilmoittaa arvojen vähimmäis- ja enimmäismäärinä hakasulkeissa, esimerkiksi [0..5] tarkoittaa, että kokoelmassa on 0-5 arvoa. Tähtimerkkiä (*) voidaan käyttää tarkoittamaan *monta, mutta täsmällistä ylärajaa ei voida antaa*). Valinnainen attribuutti voidaan kuvata moniarvoisuusmääreellä [0..1].

- Näkyvyys (engl. visibility)

  Näkyvyys on eräissä olio-ohjelmointikielissä, esimerkiksi Java ja C++, tarjolla oleva tekniikka rajoittaa attribuutin tai palvelun käyttöä. Näkyvyydellä on merkitystä vain teknisissä ohjelmointikielitasoisissa kuvauksissa. UML määrittelee neljä näkyvyystasoa:

  - *julkinen* (public, UML:ssä +) attribuutti näkyy ja on käytettävissä vapaasti luokan ulkopuolelta.
  - *suojattu* (protected, UML:ssä #) attribuutti näkyy ja on käytettävissä vain luokan ja sen aliluokkien ilmentymille.

- *pakkauksen sisäinen* (package, UML:ssä ∼) attribuutti näkyy ja on käytettävissä kaikissa samaan pakkaukseen kuuluvissa luokissa, Pakkauksia käsitellään myöhemmin.
- *yksityinen* (private, UML:ssä -) attribuutti näkyy vain saman luokan ilmentymille. On syytä huomata, että yksityinen näkyvyys ei rajoita näkyvyyttä olion sisäiseksi vaan pelkästään luokan sisäiseksi.

Näkyvyyden määrittely tulee kyseeseen *vain ohjelmointiläheisessä luokkakuvauksessa*. Puhdasoppisen oliolähestymistavan mukaisesti attribuuttien näkyvyyden tulisi olla aina vähintään tasoa suojattu.

- Oletusarvo
- Muita määreitä

  UML:ssä attribuutin arvot ovat oletusarvoisesti muuttuvia. Ne voidaan kuitenkin määritellä kiinteäarvoiseksi määreellä *readonly*. Kokoelma-arvoiseen attribuuttiin voidaan liittää määre *ordered* kuvaamaan järjestettyä kokoelmaa. Kokoelmaan voidaan liittää myös määre *unique* kuvaamaan sitä, että kokoelman alkiot ovat keskenään erilaisia.

UML:n attribuuttimäärittelyn rakenne on

[<näkyvyys>] <nimi> [<moniarvoisuus>] [:<tietotyyppi>]
[= <oletusarvo>] [ { <muut määreet> } ][5]

Esimerkkejä:

**hetu : Henkilötunnus {readonly}**

Attribuutin hetu arvo on tyyppiä Henkilötunnus. Arvo on pysyvä.

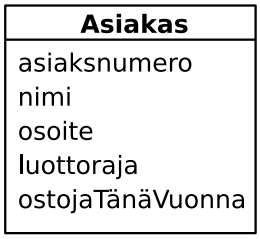
**suosikkiruoka [*] : String {ordered}**

Attribuutin suosikkiruoka arvona on järjestetty kokoelma merkkijonoja. ordered on UML:n perusmääre, joka voidaan liittää kokoelmiin ja yhteyksiin. Siitä ei näe millä perusteella arvot on järjestetty, mutta vapaamuotoisessa tekstikuvauksessa voidaan täsmentää, että ruoat ovat kokoelmassa suosituimmuusjärjestyksessä.

Kuvaesityksessä attribuuteista usein annetaan usein vain nimi. (ks. kuva 13) Tällöin tarvitaan kuvan täydennykseksi tekstimuotoinen määrittely. Tietokoneavusteisissa suunnitteluvälineissä (CASE-välineissä) on yleensä mahdollisuus valita miten paljon yksityiskohtia graafisessa luokkakuvauksessa näytetään.

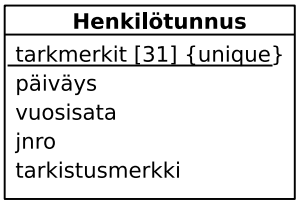
Eräissä ohjelmointikielissä (esim. Java, C++) on mahdollista määritellä luokan ilmentymiin sisältyvien tietojen lisäksi luokkakohtaisia tietoja. UML:ssä luokkakohtaiset tiedot

[5]Hakasulut ilmaisevat valinnaisen elementin eli [a] tarkoittaa, että elementti a voi kuulua esitykseen tai puuttua siitä. Kulmasulut (<>) rajaavat syntaksisen elementin. Attribuutille ainoa välttämätön kuvailutieto on täten nimi.




Kuva 13: Asiakas-luokkaa kuvaava kaaviosymboli

kuvataan alleviivaamalla attribuutin nimi. Kuvassa 14 *tarkmerkit* on luokkakohtainen kokoelma tarkistusmerkkejä.




Kuva 14: Luokka- ja ilmentymäkohtaiset attribuutit, luokkakohtainen alleviivattu

### 3.1.2 Palvelujen määrittely

Palvelun määrittely on UML:ssä muotoa

[<näkyvyys>] <nimi> [(<parametrit>)] [:<paluuarvon tyyppi>]
[{<muut määreet>}]

Määrittelyn osat ovat:

- Nimi

  Ainoastaan nimi on välttämätön. Korkean tason abstraktissa kuvauksessa riittää yleensä nimi ja palvelun tekstikuvaus.

- Parametrit

  Parametrit kuvaavat palvelupyyntöön liittyviä syöttö- ja tulostietoja, eli palvelun rajapintaa. Ohjelmointikielitasoisessa kuvauksessa ne vastaavat palvelupyynnön parametrilistaa eli signatuuria. Korkeamman abstraktiotason kuvauksessa parametrit välitetään palvelulle mahdollisesti muun mekanismin kautta, esimerkiksi sanomina tai tiedostoina.

  Parametrin määrittely on muotoa:

  [<suunta>] <nimi>: <tyyppi> [= <oletusarvo> ]

  Suunta ilmaisee onko kyseessä

  - *syöteparametri (in)*, joka välittää tietoa palvelulle. Jos syöteparametrina on olio, niin sen tila ei muutu.
  - *tulosparametri (out)*, joka välittää tietoa palvelulta sen käyttäjälle. Jos tulosparametrina on olio, sen tila voi muuttua.
  - *yhdistetty syöte- ja tulosparametri (inout)*, joka välittää tietoa kumpaankin suuntaan
  - On huomattava, että kaikki ohjelmointikielet eivät toteuta parametreja UML-määreiden mukaisina. Esimerkiksi Javassa ei ole mahdollista määritellä oliomuotoista parametria, jonka tila ei voisi muuttua.

- Näkyvyys

  Palvelujen näkyvyys voidaan määritellä samoin kuin attribuuttien näkyvyys. Näkyvyys on ohjelmointikielitason kuvauksen käsitteitä. Ylemmillä abstraktiotasoilla ollaan tyypillisesti kiinnostuneita vain julkisista palveluista.

  - *julkinen palvelu* (public, UML:ssä +) on muihin luokkiin kuuluvien olioidenkin pyydettävissä
  - *suojattu palvelu* (protected, UML:ssä #) on käytettävissä palveluissa, jotka on määritelty samassa luokkakuvauksessa kuin käytettävä palvelu tai tämän luokkakuvauksen perivissä kuvauksissa
  - *pakkauksen sisäinen* (package, UML:ssä ∼) on käytettävissä pakkauksen sisäisesti
  - *yksityinen* (private, UML:ssä -) sallii palvelun käytön vain samassa luokkakuvauksessa määritellyissä palveluissa, jossa palvelu on määritelty.

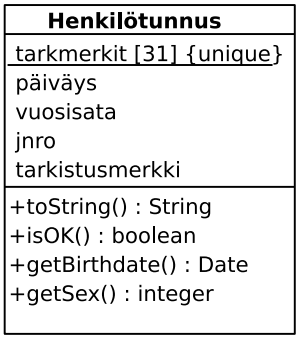
- Paluuarvon tyyppi

  Paluuarvon typpinä voi olla jokin perustietotyyppi. Paluuarvona voi olla myös olio, jolloin tyyppinä on olion luokka. UML:n palvelu perustuu palvelun funktiomaiseen tulkintaan, jossa palvelulla voi olla enintään yksi paluuarvo. Tämä ei korkeamman tason palveluita tarkasteltaessa ole erityisen onnistunut ratkaisu.

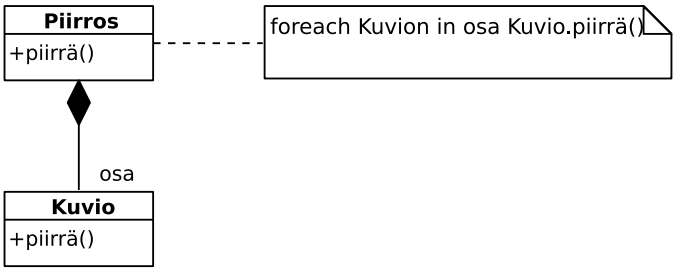
- Muut määreet

  Palveluun voidaan liittää muita määreitä. UML määrittely sisältää valmiina muutaman samanaikaisuuden hallintaan liittyvän määreen.

Kuvaesityksessä palvelusta esitetään usein vain nimi ja mahdollisesti näkyvyys (ks. kuva 15), muuten kuvasymbolit tulevat liian suuriksi. Joskus on hyödyllistä ottaa kaavioon mukaan UML-kommentti eli muistilappu, jossa voi kuvata jonkin keskeisen palvelun toimintaperiaatteen (ks. kuva 16). Palvelujen täsmälliseen kuvaukseen tarvitaan aina lisäksi tekstimuotoinen selvitys siitä, mitä palvelussa tehdään.




Kuva 15: Luokka- ja ilmentymäkohtaiset attribuutit




Kuva 16: Palvelun täsmennys muistilapulla

### 3.1.3 Olioiden kuvaaminen

Luokan ilmentymä, olio, kuvataan UML:ssä samanlaisella symbolilla kuin luokkakin. Erona on se, että ilmentymäkuvauksessa luokkanimen tilalla ylimmässä lokerossa on ilmentymäviite. Tämän rakenne on seuraava

<tunnus> | [<tunnus>] : <luokkanimi>[6]

Ilmentymäviite esitetään alleviivattuna. Ilmentymäviite voisi olla esimerkiksi Kalle:Henkilö, Kalle tai :Henkilö.

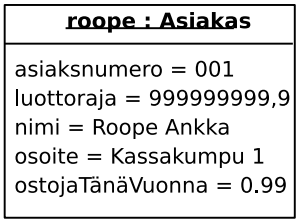
Attribuuttiosassa voidaan antaa attribuutin nimen lisäksi attribuutin arvo. Arvo annetaan vain niiden attribuuttien osalta, joihin lukijan halutaan kiinnittävän huomiota. Ilmentymäsymboleita käytetään esimerkeissä haluttaessa havainnollistaa luokkakaavion määrittelemää rakennetta (ks. kuva 17)

## 3.2 Olioiden väliset yhteydet

Olioiden (ilmentymien) välisiä rakenteellisia kytkentöjä kutsutaan *yhteyksiksi* (engl. association). Yhteys kahden olion välillä on olemassa esimerkiksi silloin, kun olio on toisen olion osa. Reaalimaailmassa tällaisia tilanteita olisivat esimerkiksi

- luku on kirjan osa

[6]Pystyyviiva ( | ) erottaa vaihtoehdot.




Kuva 17: Asiakasluokan ilmentymä

- hytti on laivan osa

Olioiden välillä voi olla myös muunlaisia yhteyksiä, esimerkiksi:

- henkilö työskentelee yrityksessä
- opiskelija suorittaa kurssia
- kirja kuuluu kurssin oppimateriaaliin

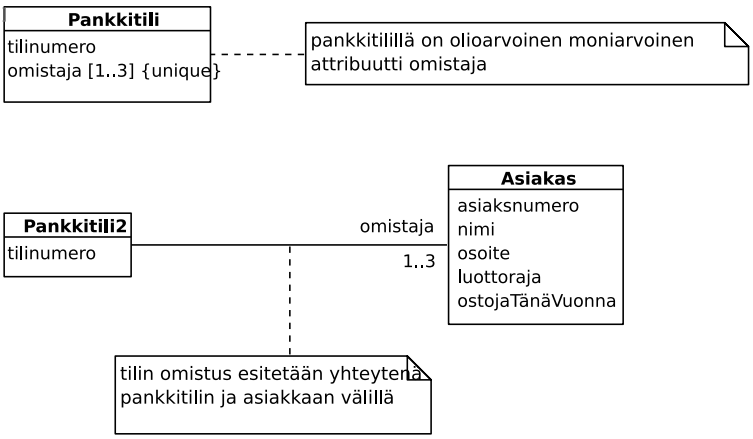
Yhteyden *Kirja kuuluu oppimateriaaliin* ilmentymänä voisi olla *Teos Koskimies&Mikkonen: Ohjelmistoarkkitehtuurit kuuluu kurssin Ohjelmistoarkkitehtuurit oppimateriaaliin*. Rakenteellisella kytkennällä tarkoitetaan tilannetta, jossa kytkentä olioiden välillä ei ole vain sattumanvaraista ja hetkellistä, kuten esimerkiksi *Pekka käyttää Java Programming kirjaa näytön päällä istuvan ampiaisen hätistelyyn*. Tässä Pekan ja kirjan välillä on selkeä kytkentä tuon hätistelyn ajan. Sen sijaan kytkentä ei ole vallitseva tilanne, asiantila, joka saataisiin selville henkilöiden työhuoneita tai työympäristöä analysoimalla. Rakenteellinen kytkentä sensijaan voisi olla se, että *henkilöllä on kirja lainassa*. Sen ilmentymä voisi yllämainitussa tilanteessa olla vaikkapa *Tiinalla on Java Programming -kirja lainassa*. Pekka sattui käyttämään hätistelyyn Tiinan lainaamaa kirjaa.

Olio-ohjelmassa rakenteellinen yhteys ilmenee siten, että oliolla on olioarvoinen attribuutti, joka sisältää viittauksen kytkettyyn olioon. Kytkentä voi olla toteutettuna myös jonkin epäsuoran viittauksen, esimerkiksi asiakasnumeron avulla.

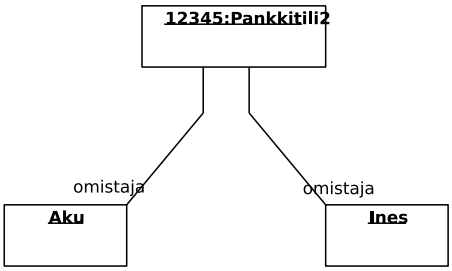
Yhteyksien esittämiseen on tarjolla myös eksplisiittinen tekniikka, joka tuo yhteydet paremmin esiin. Tässä tekniikassa yhteys piirretään näkyviin luokkakaavioon yhteyden omaavat luokat yhdistävänä viivana. Kuvassa 18 tilin omistus on kuvattu kummallakin tavalla, ylempänä olioarvoisen attribuutin avulla ja alempana eksplisiittisesti piirrettynä.

Kuvassa 18 asiakkaan *roolille* yhteydessä on annettu nimi omistaja. Lukupari 1..3 yhteysviivan päässä kuvaa kytkentärajoitteen. Mihin tahansa pankkitiliin voi omistaja-roolissa kytkeytyä enintään kolme asiakasta, ts. tilillä voi olla enintään 3 omistajaa. Edelleen jokaiseen pankkitiliin täytyy olla kytkettynä ainakin yksi asiakas.. Ilmentymätasolla tilanne voisi yhden pankkitilin kannalta tarkasteltuna olla kuvan 19 mukainen.

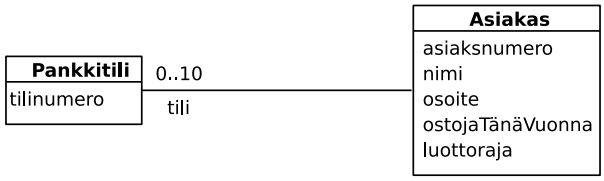
Yllä on tarkasteltu yhteyttä pankkitilistä asiakkaaseen. Yhteyttä voidaan tarkastella myös vastakkaiseen suuntaan asiakkaasta pankkitiliin. Tällöin voitaisiin päätyä kuvan 20 mukaisiin rajoitteisiin.




Kuva 18: Pankkitilin omistus olioarvoisen attribuutin ja yhteyden avulla kuvattuna

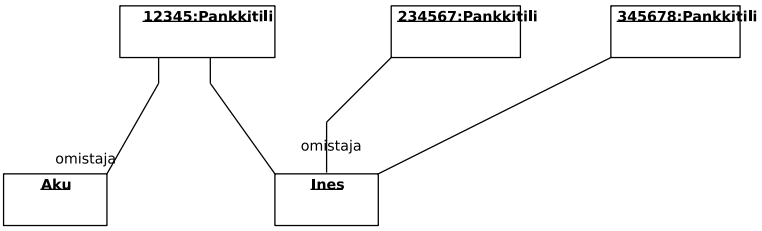



Kuva 19: Kahden omistajan yhteinen tili




Kuva 20: Tilin omistus tarkasteltuna asiakkaasta pankkitiliin

Kuvassa 20 on päädytty rajoitteeseen, jonka mukaan asiakkaalla voi olla enintään 10 pankkitiliä. Asiakkaalla ei tarvitse olla yhtään pankkitiliä. Kun ilmentymätason kuvaan otetaan mukaan kaikki asiakkaan tilit, voisi tilanne olla kuvan 21 mukainen.




Kuva 21: Ilmentymä tilinomistuksista. Ines omistaa yksinään 2 tiliä ja yhdessä Akun kanssa yhden tilin

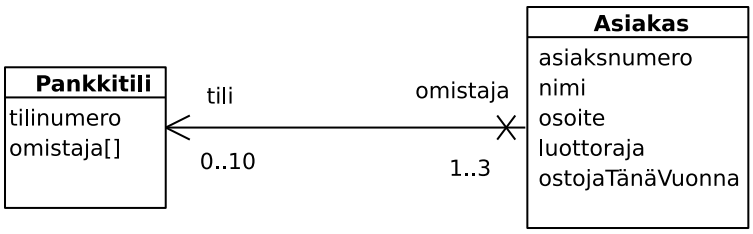
Java-ohjelmassa pankkitiliä ja asiakasta vastaavat luokat voisi olla määritelty seuraavasti

```
public class Pankkitili {
    TiliNumero tilinumero;
    int [] omistaja;  // arvona asiakasnumero
    ...
}
public class Asiakas {
    int asiakasnumero;
    Pankkitili [] tili;
    ...
}
```

Tässä ratkaisussa asiakkaasta on suorat olioviitteet asiakkaan pankkitileihin. Pankkitilistä puolestaan on epäsuora asiakasnumeron avulla hoidettu viite omistaja-asiakkaisiin. Tällöin asiakkaasta on *suora pääsy* (engl. direct access) pankkitiliin, mutta pankkitilistä ei ole suoraa pääsyä asiakkaaseen. UML:ssä tämä voidaan esittää *navigointimääreen* avulla. Navigointimääre esitetään liittämällä yhteysviivaan avoin nuolenkärki, joka osoittaa siihen luokkaan, jonne toisesta osapuolesta on suora pääsy (ks. kuva 22). Se yhteyden pää, jonka suuntaan ei kyetä ohjelmassa suoraan navigoimaan voidaan merkitä uusimmassa UML-standardissa viivan päälle piirretyllä rastilla (x). Vanhemmissa UML-standardeissa ei tähän päähän tehty mitään merkintöjä. Uudemmankin UML-standardin mukaisissa kaavioissa rastin merkitseminen on äärimmäisen harvinaista.

Suora pääsy on mahdollista määritellä kumpaankin suuntaan. Yleensä se toteutetaan ainakin toiseen suuntaan. Navigointimäärittelyt kuuluvat matalan tason ohjelmointiläheiseen kuvaukseen. Korkeamman abstraktiotason kuvauksissa se jätetään teknisen toteutuksen yksityiskohtina kuvaamatta.

Kuvassa 22 on yhteyden kuvauksessa käytetty roolinimiä tili ja omistaja. Rooli kuvaa yhteyden osapuolen asemaa suhteessa toiseen osapuoleen, esimerkiksi asiakkaan asema suhteessa pankkitiliin on olla omistaja. Kuvassa roolinimet ovat samat kuin yllä olevassa




Kuva 22: Omistus-yhteyden toteutustapa siten, että asiakkaasta on suora pääsy pankkitiliin, mutta pankkitilistä ei asiakkaaseen

Java-ohjelmassa yhteyden toteutukseen käytettyjen attribuuttien nimet. Tämä kuvastaakin varsin luontevaa tapaa edetä suunnitelmasta toteutukseen käyttämällä ohjelmassa suoraan niitä nimiä, jotka esiintyvät suunnitelmassa.

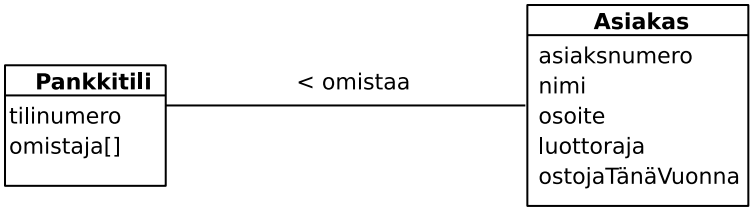
Olioiden väliset yhteydet esitetään luokkakaaviossa kytkemällä osapuolten luokat yhteen viivalla. Viivan päissä ilmoitetaan kytkentärajoitteet ja roolinimet. Kytkentärajoite esitetään alaraja..yläraja -parina. Käytännössä eniten merkitystä on alarajoilla 0 ja 1. Alaraja 0 tarkoittaa, että osapuoliluokan olioiden ei tarvitse olla mukana yhdessäkään yhteydessä eli yhteys on niille valinnainen (esim.. asiakkaalla ei tarvitse olla pankkitiliä). Alaraja 1 puolestaan merkitsee pakollista yhteyttä (esim. tilillä on oltava vähintään yksi omistaja). Ylärajoista eniten on merkitystä arvoilla 1 ja * (epämääräisen monta). Yläraja 1 määrittelee yhteyden funktionaaliseksi. Kullakin oliolla on enintään yksi kumppani (esim. kurssilla on vain yksi vastuuhenkilö). Usein tiedetään, että olio voi olla yhteydessä moneen olioon, mutta ei ole mahdollista antaa mitään kiinteää ylärajaa kumppaneiden määrälle. Tällöin käytetään ylärajaa * (monta), esimerkiksi tilanteessa *kurssilla on monta opiskelijaa*. Jos ala- ja yläraja ovat samat riittää antaa luku kertaalleen.

Yhteyksiä voi luokitella tyyppeihin osapuolten kytkentärajoitteiden ylärajojen suhteen perusteella. Näin saadaan yhteystyypit:

- yhden suhde yhteen
- yhden suhde moneen ja
- monen suhde moneen

UML:ssa roolinimien lisäksi myös koko yhteydelle voi antaa nimen. Yhteyden nimi merkitään keskelle yhteyttä. Yhteyden nimi voi olla suunnattu tai suuntaamaton. Esimerkiksi *omistus* on suuntaamaton nimi ja *omistaa* on suunnattu. Jotta suunnattua nimeä käytettäessä ei syntyisi tulkintaongelmia siitä, miten yhteysnimi pitäisi tulkita, voidaan nimeen liittää lukusuunta (kuva 23). Nimen antaminen yhteydelle ei ole välttämätöntä. Usein roolinimipari (esimerkiksi tili-omistaja) riittää identifioimaan, mistä yhteydestä on kyse. Joskus taas yhteyteen liittyvien olioiden roolit ovat niin itsestään selvät, että roolinimille ei ole tarvetta.

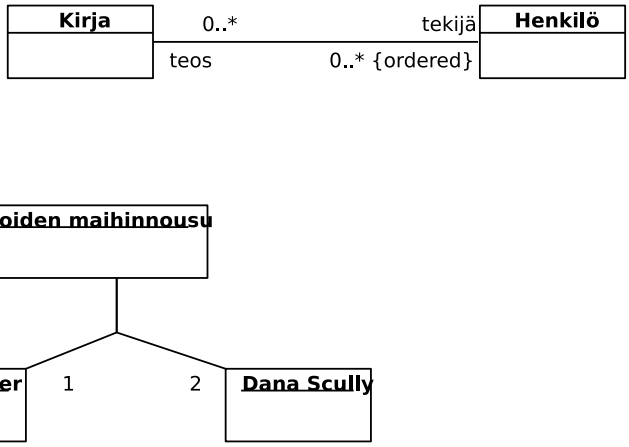
Yhteydet molemmat osapuolet voivat kuulua samaan luokkaan. Tällöin roolinimien käyttö on välttämätöntä. Kuvan 24 esimerkissä kuvataan työnjohtosuhdetta. Työntekijällä voi olla enintään yksi esimies ja useita alaisia.




Kuva 23: Yhteysnimen lukusuunta: Asiakas omistaa pankkitilin


Kuva 24: Yhteys, jonka osapuolet kuuluvat samaan luokkaan

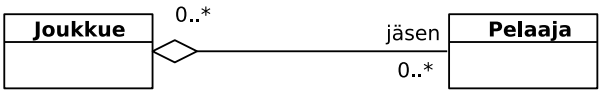
Yhteyteen voidaan liittää järjestysmääre kuvaamaan sitä, että samaan olioon yhteydessä olevien olioiden joukko on jollain perusteella järjestetty. Kuvassa 25 on esimerkki tällaisesta yhteysmäärittelystä ja sen mukaisesta ilmentymästä.




Kuva 25: Järjestetty kumppanijoukko

Luokkakaavioita laadittaessa joudutaan usein kuvaamaan kokonaisuuden ja siihen kuuluvan osan välisiä yhteyksiä. Tällaiselle yhteydelle on oma nimityksensä *kooste* (engl. aggregate) ja UML:ssä oma merkintätapansa, salmiakkisymboli kokonaisuuden puoleisessa päässä. Kuvan 25 esimerkissä joukkueeseen voi kuulua monta pelaaja ja pelaaja voi kuulua useaan joukkueeseen. Pelaaja on osa joukkuetta mutta voi myös vaihtaa joukkueesta toiseen.

Koosteen käyttö lisää mallin luettavuutta. Muuta merkitystä sillä ei ole. Se ei vaikuta millään tavalla navigointiin eikä osallistumisrajoitteisiin. Koosteen tapauksessa osan ja kokonaisuuden välinen kytkentä on löyhä. Osa voidaan irrottaa kokonaisuudesta ja liittää toiseen. Esimerkissä pelaaja voi vaihtaa joukkuetta ja kuulua moneen eri joukkueeseen. Joukkueen lopettaminen ei vaikuta mitenkään pelaajiin.

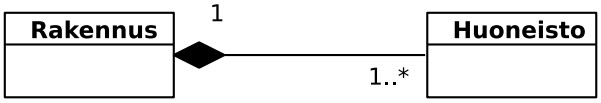



Kuva 26: Koosteyhteys

Koostetta merkittävämpi rakenne mallintamisen kannalta on *kompositio* (engl. composition). Kompositioon liittyy tiukkoja rajoituksia:

- kompositiossa osa on olemassaoloriippuva kokonaisuudesta. Kokonaisuuden tuhoaminen hävittää myös sen osat.
- osa voi kuulua vain yhteen samantyyppiseen kompositioon
- osa on koko elinaikansa kytkettynä samaan kokonaisuuteen.

Kompositio kuvataan UML:ssä mustalla salmiakkisymbolilla kokonaisuuden puolessa päässä yhteysviivaa (ks. kuva 27).

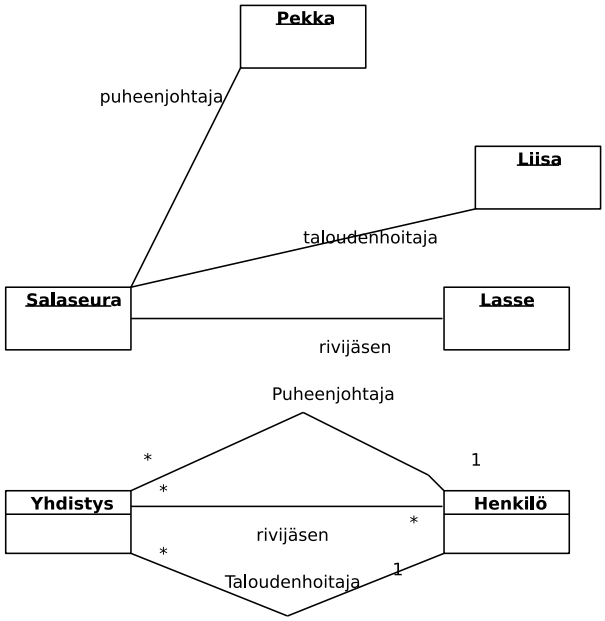



Kuva 27: Kompositio

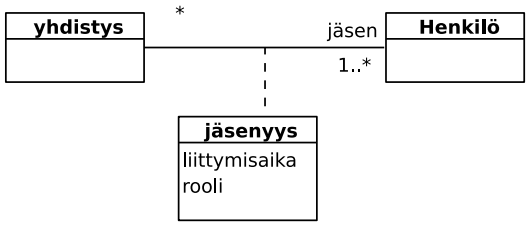
Kuvan 27 esimerkissä *Huoneisto* on koko olemassaolonsa ajan osa samaa rakennusta. Jos rakennus puretaan, häviää myös huoneisto. Olio-ohjelmassa komposition tehtävä on luoda ja tuhota osansa. Kompositiossa kokonaisuuteen liittyvien osien joukon ei tarvitse olla kiinteä (se voi toki olla sitä). Kompositioon voidaan lisätä osia ja siitä voidaan poistaa niitä. Kompositiota voidaan käyttää hyväksi myös sen osien ulkoisessa tunnistamisessa. Esimerkiksi huoneisto identifioidaan yleisesti identifioimalla rakennus, johon huoneisto kuuluu. Rakennuksen sisällä huoneiston identifiointiin riittää sisäinen huoneistonumero.

Joskus on tarpeen liittää yhteyteen täsmentävää tietoa siitä millainen yhteys on kyseessä. Tarkastellaan esimerkkinä jäsenyyttä yhdistyksessä (ks. kuva 28). Jäsen voi kuulua yhdistykseen erilaisissa rooleissa. Kuvassa ovat roolit puheenjohtaja, taloudenhoitaja ja rivijäsen. Kaikkia näitä vastaten olisi mahdollista määritellä eri yhteystyypit, kuten on tehty kuvassa 28.

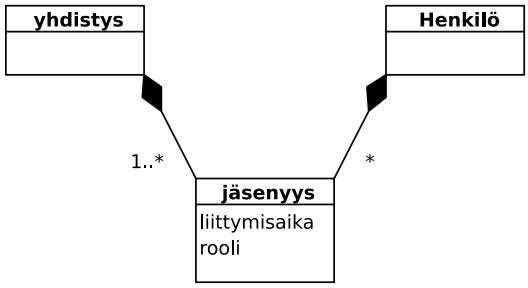
Jos vaihtoehtoja on runsaasti ja vaihtoehtokokoelma elää, tarvitaan jokin muu tapa mallintaa tilanne. UML:ssä tällaisen rakenteen kuvaamiseen voi käyttää *yhteysluokkaa* (engl. association class) ks. kuva 29. Yhteysluokka mahdollistaa attribuuttien liittämisen yhteyteen. Yhteysluokka tulee yleensä kyseeseen ns. monen suhde moneen yhteyksien (kummallakin osapuolella voi olla monta kumppania) tapauksessa. Yhteysluokan käyttö esimerkin tilanteessa johtaa väljempiin rajoitteisiin kuin erillisten yhteystyyppien käyttö. Tätä rakennetta käytettäessä ei ole mahdollista rajoittaa puheenjohtajien ja taloudenhoitajien määrää. Kolmas mahdollinen tapa olisi määritellä erillinen luokka jäsenyys ja kytkeä se yhdistykseen ja henkilöön (ks. kuva 30).




Kuva 28: Erilaisia jäsenyysrooleja, joista kukin mallinnettu omana yhteystyyppinään.



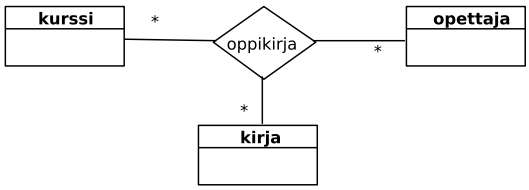

Kuva 29: Yhteyden luonnehdinta yhteysluokan avulla




Kuva 30: Yhdistävä luokka

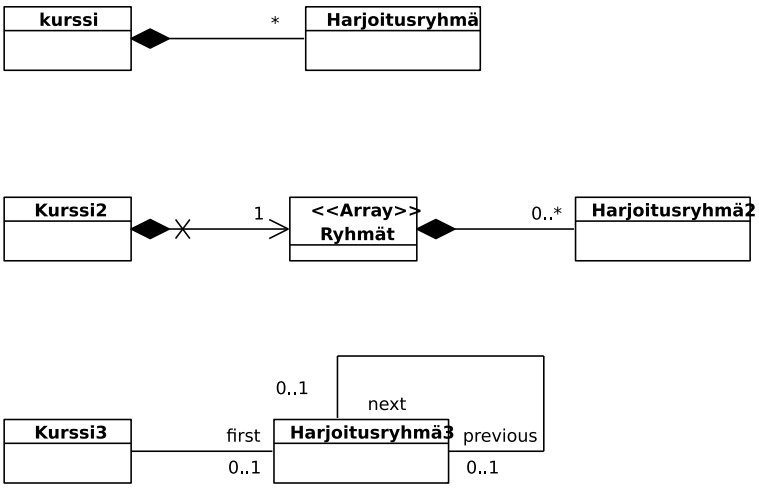
Kuvien 29 ja 30 kuvaamien vaihtoehtojen ohjelmatoteutuksessa ei välttämättä ole mitään eroa.

UML:ssä on tarjolla myös useamman kuin kahden osapuolen välinen yhteys. Tällaisesta on esimerkki kuvassa 31. Siinä eri opettajat käyttävät samalla kurssilla eri oppikirjaa. Tämä voidaan muuntaa kahdenvälisiin yhteyksiin perustuvaksi muuntamalla yhteys oppikirja luokaksi.




Kuva 31: Kolmenvälinen yhteys

Luokkakaavioita voidaan laatia ohjelmaläheisinä, jolloin niillä pyritään mahdollisimman tarkoin kuvaamaan ohjelman oliorakenne. Korkeammalla abstraktiotasolla laaditut abstraktit mallit jättävät joitain yksityiskohtia esittämättä. Eräs tyypillinen ero korkeamman tason kaavion ja ohjelmaläheisen kaavion välillä liittyy olioiden välisiin yhteyksiin. Ohjelmaläheisessä kuvauksessa esitetään, miten olioiden välinen yhteys teknisesti toteutetaan. Korkeamman abstraktiotason kuvauksessa riittää kuvata keskeisten yhteyksien olemassaolo. Tarkastellaan esimerkkinä tästä kurssin ja harjoitusryhmän välistä yhteyttä (ks. kuva 32).




Kuva 32: Abstrakti yhteys ja sen kaksi teknistä toteutusta

Kuvan ylälaidassa on abstrakti kuvaus tilanteesta, keskellä ja alhaalla on kaksi teknisen tason kuvausta rakenteen toteutuksesta, ylempi taulukkorakennetta ja alempi listarakennetta käyttäen. Taulukkorakenteen kuvauksesta pystytään vielä hahmottamaan alkuperäinen abstrakti rakenne, mutta listarakenteen kuvauksesta ei enää näy, että kurssiin liittyy useita harjoitusryhmiä. Sopivan abstraktiotason valinta onkin oleellista luokkakaavioiden käyttökelpoisuuden kannalta. Yllä olevassa kuvassa luokka *Ryhmät* on määritelty taulukoksi

käyttämällä ns. *stereotyyppi*-käsitettä. Tämä on UML:n mallikäsitteistön laajennustekniikka, jolla metamallin käsitteistöä voidaan laajentaa määrittelemällä erikoistuneita tietyssä ympäristössä käytettäviä mallinnuskäsitteitä. Karkeasti ottaen stereotyypin voidaan ajatella tarkoittavan kaavioon liitettyä määrämuotoista kommenttia. Stereotyyppi-käsitettä tarkastellaan laajemmin luvussa 3.4.

## 3.3 Luokkamallin laatiminen

Järjestelmän olioperustaista toteutusta voidaan ajatella simulointimallin muodostamisena jostakin reaalimaailman ilmiöstä. Tällöin ohjelmassa toimivat oliot vastaavat pitkälti reaalimaailman ilmiössä osallisena olevia olioita. Lisäksi tarvitaan olioita, joiden kautta voidaan tarkastella simulointimallin tilaa ja välittää siihen liittyvää ohjaustietoa (eli käyttöliittymä). Keskeistä simulointimallin muodostamisessa on löytää ne reaalimaailman kohteet, joita vastaavia luokkia ohjelmaan tulisi ottaa mukaan. Näiden luokkien etsimistä voidaan kutsua oliomallinnukseksi tai *käsiteanalyysiksi* (engl. conceptual modeling). Tehtävänä on löytää oleellinen rakenne simuloitavalle reaalimaailman ilmiölle.

Mallin laatiminen voisi tapahtua seuraavien vaiheiden mukaisesti:

1. Kartoita luokkaehdokkaita

   Laadi luettelo tarkasteltavan ilmiön kannalta keskeisistä kohteista tai ilmiöistä, jotka voisivat tulla kyseeseen luokkina tai olioina. Tällaisia voisivat olla toimintaan osallistujat, toiminnan kohteet, toimintaan liittyvät tapahtumat, materiaalit, tuotteet ja välituotteet, toiminnalle edellytyksiä luovat asiat.

   Kartoituksen pohjana voi käyttää vapaamuotoista tekstikuvausta tarkasteltavasta ilmiöstä, kutsutaan sitä jatkossa *kohdealueeksi* (engl. problem domain). Tästä kuvauksesta alleviivataan luokkaehdokkaita ja kerätään ne luetteloon. Luokkaehdokkaat esiintyvät kuvauksessa usein substantiiveina. Verbit ilmaisevat joskus yhteyksiä. Alustavaa karsintaa voi tehdä sen perusteella, onko asia lainkaan oleellinen mallinnettavan ilmiön kannalta.

2. Karsi ehdokkaita

   Luetteloon saadut ehdokkaat käydään läpi ja arvioidaan voisiko ehdokas tulla kyseeseen luokkana. Arvioinnissa tulisi tarkastella

   - Liittyykö ilmiöön tietosisältöä, joka on välttämätöntä järjestelmän kannalta
   - Onko asia riittävän tärkeä kohdealueen kannalta.

   Karsintaa ja ehdokkaiden kartoitusta joudutaan usein tekemään iteratiivisesti. Ensimmäinen karsintakierros ei välttämättä tuota lopullista tulosta.

3. Tunnista olioiden väliset yhteydet

   Yhteyksiä voi etsiä vapaamuotoisesta kuvauksesta (verbit, genetiivit, muut ilmaukset jotka kuvaavat kytkentää). Yhteyksienkin suhteen tulisi miettiä onko yhteys oleellinen tarkasteltavan ilmiön kannalta sekä onko se rakenteellinen.

4. Täsmennä luokkakuvauksia määrittelemällä attribuutit

   Attribuutteja saattaa löytyä vapaamuotoisesta kuvauksesta, mutta yleensä niiden löytäminen edellyttää lisäselvityksiä kohdealueesta, esimerkiksi toiminnan osapuolten haastatteluja. Attribuuttien kohdalla pitäisi myös selvittää mihin niitä tarvitaan.

5. Määrittele yhteyksiin liittyvät osallistumisrajoitteet

   Osallistumisrajoitteiden avulla ilmaistaan rakenteellisia sääntöjä. Ne eivät välttämättä tule esiin vapaamuotoisessa kuvauksessa vaan edellyttävät tarkempaa kohdealueen analysointia.

6. Liitä luokkiin palvelut ja varmista palvelujen ja tietosisällön yhteensopivuus

   Palveluja ei yleensä määritellä, kun tehdään kohdealueen luokkamallia. Palvelujen määrittäminen tapahtuu vasta ohjelman suunnitteluvaiheessa. Aiheeseen palataan luvussa 6.

### 3.3.1 Esimerkki

Tarkasteltavana ilmiönä on elokuvalipun varaaminen. Lippu oikeuttaa paikkaan tietyssä näytöksessä. Näytöksellä tarkoitetaan elokuvan esittämistä tietyssä teatterissa tiettyyn aikaan. Samaa elokuvaa voidaan esittää useissa teattereissa useina aikoina. Asiakas voi samassa varauksessa varata useita lippuja yhteen elokuvaan.

Edellistä kappaletta voidaan pitää ilmiön vapaamuotoisena kuvauksena. Otetaan se jatkokäsittelyyn ja alleviivataan luokkaehdokkaita.

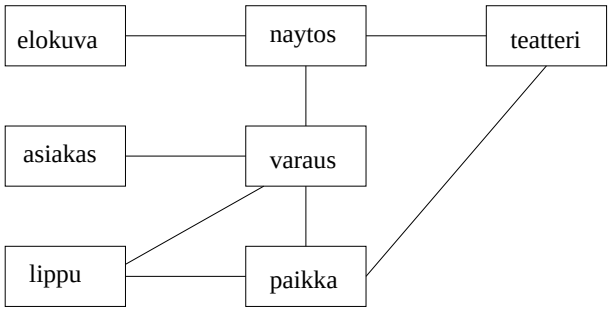
> Tarkasteltavana ilmiönä on elokuvalipun varaaminen. Lippu oikeuttaa paikkaan tietyssä näytöksessä. Näytöksellä tarkoitetaan elokuvan esittämistä tietyssä teatterissa tiettyyn aikaan. Samaa elokuvaa voidaan esittää useissa teattereissa useina aikoina. Asiakas voi samassa varauksessa varata useita lippuja yhteen elokuvaan.

Saatiin luettelo:

- *elokuvalippu*
- lippu
- paikka
- näytös
- elokuva
- teatteri
- asiakas

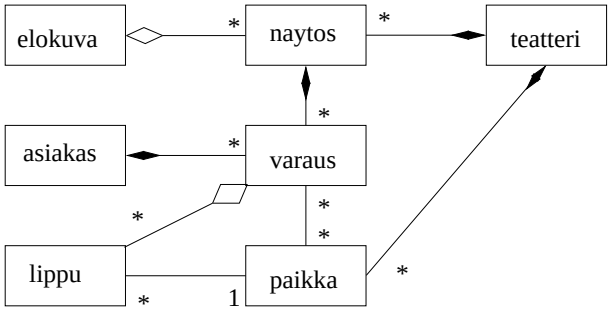
- varaus

Päätetään jättää muut ehdokkaat paitsi elokuvalippu, joka on selvästi termin lippu synonyymi. Kuvassa 33 on ensimmäinen luonnos luokkakaaviosta. Yhteyksiä on luonnosteltu, mutta niihin ei vielä liity rajoitteita eikä edes nimiä.




Kuva 33: Luokkakaavion ensimmäinen luonnos

Seuraavassa versiossa on tunnistettu kompositiot ja koosteet (ks. kuva 34) ja määritelty yhteyksien osallistumisrajoituksia. Koska varauksen ja paikan välinen yhteys on pääteltävissä varauksen ja lipun sekä lipun ja paikan välisistä yhteyksistä, se karsitaan redundanttina (toisteisena) pois.



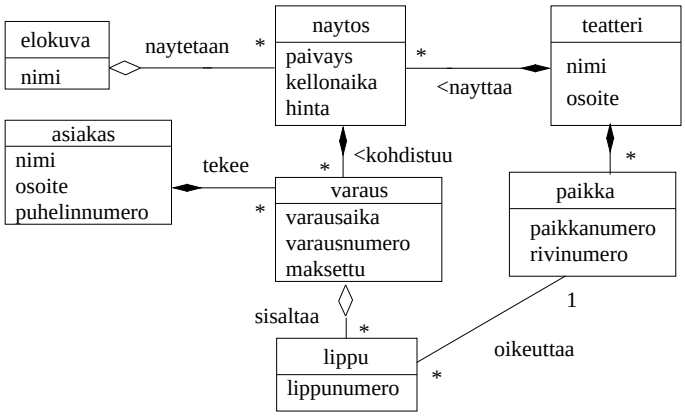

Kuva 34: Luokkakaavion seuraava versio

Seuraavaksi kaaviota täydennetään keskeisillä attribuuteilla (ks. kuva 35). Myös yhteyksiä on nimetty vaikka tässä tapauksessa nimet eivät tuo juurikaan lisäinformaatiota.

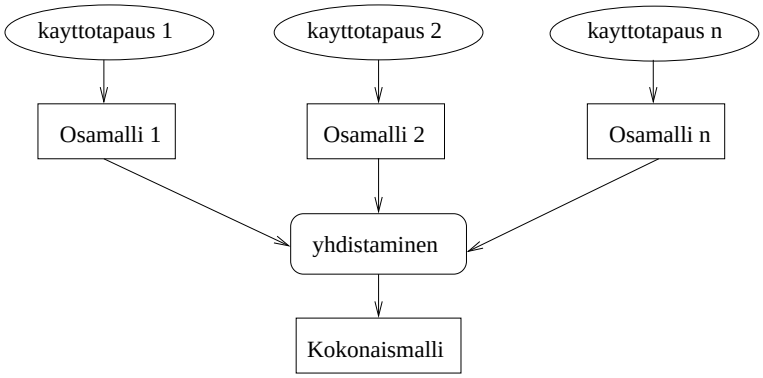
Yhteyksiä ja attribuutteja voidaan vielä joutua määrittelemään uudelleen kun luokkiin liitetään palvelut eli ohjelmiston suunnitteluvaiheessa.

### 3.3.2 Käyttötapauskohtainen ja iteratiivinen luokkamallin laatiminen

Käyttötapaukset voivat toimia myös perustana järjestelmän luokkamallin määrittelyssä. Tähän perustuu ns. *käyttötapauspohjainen* määrittely. Tässä lähestymistavassa laaditaan aluksi käyttötapauskohtaisia osamalleja, jotka sitten yhdistetään kokonaisvaltaiseksi malliksi (ks. kuva 36).




Kuva 35: Luokkakaavion seuraava versio




Kuva 36: Käyttötapauspohjainen tietosisällön määritys

Kukin käyttötapauskohtainen osamalli määrittelee sisältöluokat vain käyttötapauksen edellyttämässä laajuudessa. Syntyvät mallit ovat siten kokonaismallia suppeampia ja näin helpommin aikaansaatavissa. Lähtökohdaksi mallinnukseen voidaan ottaa käyttötapauksen kuvaus, mahdollinen käyttötapaukseen liitetty luonnos käyttöliittymästä, raporttimalli tai lomake, joka liittyy käyttötapaukseen. Jos lähtökohtana on tekstikuvaus, mallinnus voi lähteä liikkeelle siten, että aluksi alleviivataan tekstistä luokkaehdokkaat ja edetään sitten kuten edellä esitettiin.

Jos ohjelmiston kehittäminen tapahtuu ketterien menetelmien suosittamalla iteratiivisella lähestymistavalla (ks. luku 1.1.5), kannattaa myös ohjelman luokkamallia rakentaa iteratiivisesti. Eli jos ensimmäisessä iteraatiossa toteutetaan esim. ainoastaan käyttötapausten 1 ja 2 mukainen toiminnallisuus, esitetään iteraation luokkamallissa vain ne luokat, jotka ovat merkityksellisiä tarkastelun alla olevan toiminnallisuuden kannalta. Luokkamallia täydennetään myöhempien iteraatioden aikana tarpeellisilta osin.

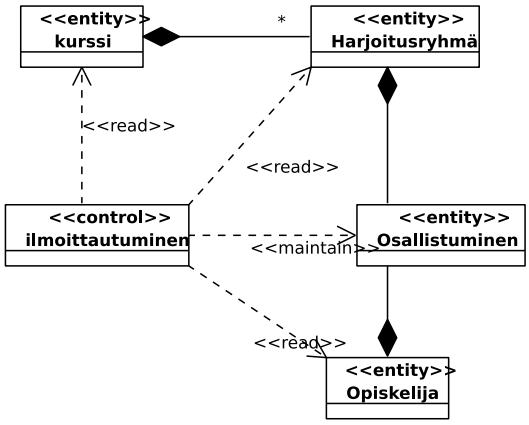
Tarkastelemme UML:n soveltamista iteratiiviseen ohjelmistokehitykseen tarkemmin luvuissa 5 ja 6.

## 3.4 Mallinnuskäsitteistön erikoistaminen

Ohjelmistojen kehittämismenetelmien ja ohjelmistotutkimuksen yhteydessä esitetään erilaisia käsitteistöjä, joiden avulla ohjelmistot olisivat ainakin käsitteistön kehittäjien mielestä aiempaa paremmin hahmotettavissa. Käsitteistöt voivat olla yleisiä tai sovellusaluekohtaisia. UML kuvauskielen pohjalla on abstrakti, joskin monin paikoin hyvin ohjelmointiläheinen käsitteistö. UML:n pohjana oleva käsitteistö ei olekaan tarkoitettu sinällään suoraan käytettäväksi vaan peruskäsitteistöksi, jonka päälle voi rakentaa uusia mallinnuskäsitteistöjä. UML kuvaustekniikka on kehitetty yleiskäyttöiseksi kehykseksi, jota voi laajentaa ja erikoistaa omien tarpeiden mukaiseksi. Erikoistaminen tapahtuu määrittelemällä uusia mallinnuskäsitteitä, kiinnittämällä mallinnuskäsitteisiin liittyviä ominaisuuksia tai määrittelemällä mallinnukseen liittyviä sääntöjä. Näitä laajennustekniikkoja käyttävät yleensä menetelmäkehittäjät. Seuraavassa näistä rakenteista tarkastellaan tärkeintä uusien käsitteiden määrittelytapaa.

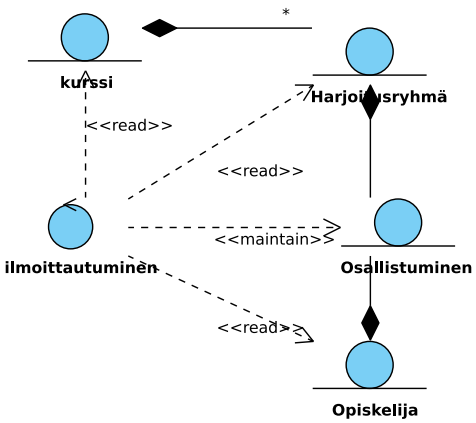
UML:n peruskäsite voidaan erikoistaa määrittelemällä peruskäsitteeseen pohjautuva erikoiskäsite. UML käyttää näistä nimitystä *stereotyyppi* (engl. stereotype). Esimerkiksi luokkakaavion käsite *luokka* voitaisiin erikoistaa käsitteiksi sisältöluokka eli tietokohde (entity), ohjausluokka (control) ja liittymäluokka (interface), kuten on tehty Jacobsonin esittämässä suunnittelumenetelmässä [11] (ks. myös luku 6.3.3). Tällöin luokkakaavion laatija käyttäisi luokkakaaviotekniikkaa, mutta erikoistetuin käsittein. Luokkien lisäksi voi stereotyyppejä määrittelemällä erikoistaa myös yhteyksiä, attribuutteja ja palveluita. UML-kuvauskieli tarjoaa oletusarvoisen tavan esittää erikoistettu käsite graafisesti liittämällä kohteen yhteyteen erikoistavan käsitteen nimi väkäsiin suljettuna. Kuvassa 37 esiintyy erikoistettuina luokkina tietokohde (entity) ja ohjausluokka (control) sekä erikoistettuja riippuvuuksia kuvaamassa tietokohteiden käyttöä.

UML-tekniikassa on mahdollista myös määritellä erikoistetuille käsitteille oma graafinen




Kuva 37: Erikoistettuihin käsitteisiin perustuva luokkakaavio

esitys. Kuvassa 38 on kuvan 37 kaavio esitetty Jacobsonin kuvaustekniikan symboleita käyttäen. UML-malliin perustuvat suunnitteluohjelmistot tukevat ainakin kuvan 37 esitysmuotoa. Osa työkaluista, kuten tämän monisteen joidenkin kaavioiden laadinnassa käytetty Visual Paradigm, tukevat myös erikoistettuja, kuvassa 38) käytettyjä symboleja.




Kuva 38: Erikoistetuin käsittein ja symbolein esitetty luokkakaavio

Erikoistettu käsite on luonteeltaan erikoistapaus UML:n tarjoamasta yleiskäsitteestä. Kaikki UML:n peruskäsitteistön yhteydessä määritellyt säännöt ja rajoitukset periytyvät erikoistetuille käsitteille, ellei niiden määrittelyn yhteydessä anneta uusia korvaavia sääntöjä. Periytymistä yleisesti tarkastellaan tämän luvun lopussa. Erikoistamismahdollisuus ei rajoitu pelkästään luokkakaavioihin, vaan sitä voidaan käyttää kaikissa UML:n tekniikoissa. Itse asiassa osa UML:n tekniikoista, esimerkiksi komponenttikaavio ja käyttötapauskaavio ovatkin pohjimmiltaan erikoistettuihin käsitteisiin perustuvia luokkakaavioita.

## 3.5 Riippuvuudet

Yllä olevassa esimerkissä (kuvat 37 ja 38) esiintyi luokkien välisiä *riippuvuuksia* (engl. dependency). Luokka riippuu toisesta, jos muutos toisen luokan määrittelyssä voi aiheut-

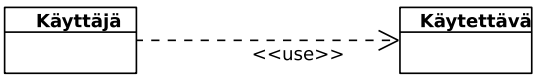
taa tarpeen muuttaa luokan määrittelyä. Esimerkissä esiintyi erikoistettu lukuriippuvuus (read). Ohjelma lukee tietokohteen tietoja. Jos tietokohteen määrittely muuttuu, voidaan joutua muuttamaan lukevaa ohjelmaa. UML:ssä on valmiiksi määriteltynä joitain riippuvuuksia sekä yleinen riippuvuus, joka soveltuu erikoistettavaksi. Valmiiksi määriteltyjä riippuvuustyyppejä ovat esimerkiksi

- *call* palvelujen välinen kutsuriippuvuus
- *create* luokan oliot luovat toisen luokan olioita
- *derive* atrribuutin arvo on laskettavissa toisen arvosta tai yhteys pääteltävissä
- *instantiate* luokan oliot luovat toisen luokan olioita
- *realize* luokka toteuttaa rajapinnan
- *refine* kohteet tarkentavat abstraktimpaa kuvausta
- *use* kohteiden välinen käyttösidos

Riippuvuudet ovat mallitason käsitteiden välisiä toisin kuin yhteydet, jotka ovat ilmentymien välisiä. Riippuvuus esitetään kaaviossa katkoviivalla, jonka päässä oleva nuolenkärki osoittaa siihen luokkaan, josta toinen luokka on riippuva.

Usein esiintyvä riippuvuus luokkien välillä on palvelun parametrin aiheuttama riippuvuus, jossa palvelun tarjoava luokka tulee riippuvaksi parametrin luokasta. Alla oleva ohjelmakoodi aiheuttaisi tällöin kuvan 39 mukaisen riippuvuuden. Riippuvuuden tyyppi on tällöin *use*.

```
class Käyttäjä {
     public omaPalvelu(Käytettävä k) {
        k.vierasPalvelu();
   }
}
```

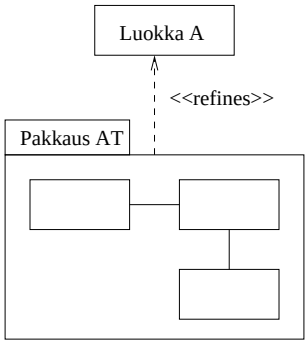



Kuva 39: Luokkien välinen riippuvuus Käyttäjä on riippuvainen Käytettävästä.

Tarkemmin luvussa 6.3.1 käsiteltävä *pakkaus* (engl. package) on tapa jäsentää ohjelmistoa esimerkiksi kokoamalla yhteenkuuluvia luokkia osajärjestelmäksi. Pakkauksia voi käyttää myös siten, että pakkauksessa määritellään alemman abstraktiotason tarkennus ylemmällä abstraktiotasolla kuvatulle käsitteelle. Tälläistä tilannetta voidaan havainnollistaa kuvan 40 tapaan, eli käyttäen luokan ja sen tarkentavan pakkauksen välillä tarkennusriippuvuutta.

## 3.6 Yleistyshierarkia ja periytyminen

Luokkakaavion laatiminen on kohdealueen jäsentämistä. Sitä tehtäessä luodaan tai kirjataan luokitusjärjestelmä kohdealueen ilmiöille. Luokitusjärjestelmälle voidaan asettaa erilaisia vaatimuksia. Joissakin menetelmissä edellytetään, että ilmiöt on luokiteltava siten,




Kuva 40: Pakkaus, jonka sisältö tarkentaa korkeammalla abstraktiotasolla esitettyä luokkaa

että kaikki luokat ovat erillisiä, ts. niillä ei ole yhteisiä ilmentymiä. Toiset menetelmät puolestaan sallivat ainakin jonkinasteisen luokkien päällekkäisyyden.

Luonnollisessa kielessä luokittelu on vapaata. Käsitteet ja niitä vastaavat luokat ovat limittäisiä tai päällekkäisiä. Kun tarkastellaan, miten luokan ilmentymien joukko suhtautuu toisen luokan ilmentymien joukkoon, voidaan erottaa tapaukset:

- ilmentymäjoukot ovat aina pistevieraat (esim. auto ja henkilö)
- ilmentymäjoukot voivat sisältää yhteisiä ilmentymiä (esim. nainen ja johtaja)
- ilmentymäjoukko sisältyy aina toisen luokan ilmentymäjoukkoon (esim. nainen ja henkilö).

Jos luokan A ilmentymäjoukko sisältää aina luokan B ilmentymäjoukon on luokkien välillä *yleistys-erikoistus* (engl. generalization-specialization) suhde. Käsite B on erikoistapaus käsitteestä A (nainen on henkilön erikoistapaus) tai päinvastoin käsite A on yleistys käsitteestä B (henkilö on yleiskäsite, johon nainen sisältyy). Kun luokkien suhdetta tarkastellaan yleistys-erikoistus-suhteeseen perustuen, kutsutaan yleisempää luokkaa *yläluokaksi* (engl. super class) ja erikoistuneempaa *alaluokaksi* (engl. subclass). Yläluokka on yleistyshierarkiassa (luokkahierarkiassa) ylemmällä tasolla, siis yleisempi, kuin alaluokka.

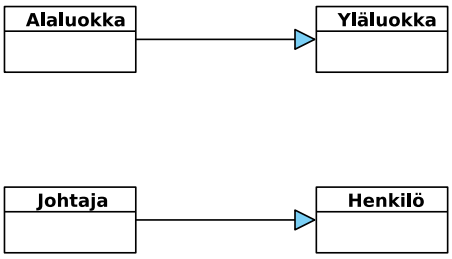
Esimerkiksi voitaisiin määritellä

- luokka nainen on luokan henkilö alaluokka
- luokka johtaja on luokan henkilö alaluokka
- luokat auto, laiva ja lentokone ovat luokan kulkuväline alaluokkia

Siitä, että luokka B on luokan A alaluokka seuraa, että jokainen B:n ilmentymä on myös A:n ilmentymä, eli jokainen johtaja on myös henkilö. Tästä taas seuraa, että kaikki yhteydet ja attribuutit, jotka ovat mahdollisia luokan A ilmentymille, ovat mahdollisia myös B:n ilmentymille. Jos siis henkilölle on määritelty attribuutti Nimi ja johtaja on henkilön alaluokka, niin myös johtajalla on automaattisesti Nimi-attribuutti. Jos henkilö on määritelty osapuoleksi työsuhde-yhteyteen, myös johtajan ilmentymä voi olla osapuolena työsuhde yhteydessä. Tätä ilmiötä kutsutaan *periytymiseksi* (engl. inheritance).

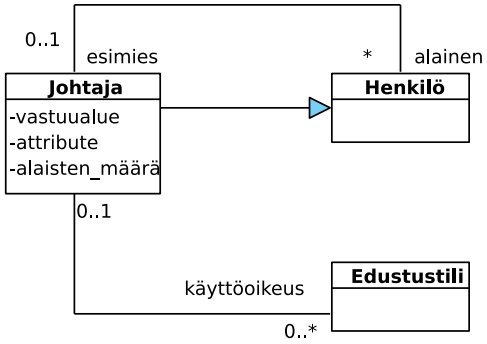
Periytymisessä yläluokkaan liitetyt attribuutit, palvelut ja yhteydet periytyvät alaluokalle, eli ovat voimassa myös alaluokan ilmentymille. On syytä pitää mielessä, että periytyminen on luokkien välistä määrittelyjen periytymistä - ilmentymät eivät peri mitään toisilta ilmentymiltä. Luokkakaavion laatimisen kannalta periytyminen tarkoittaa sitä, että piirrettä, joka on liitetty yläluokkaan, ei tarvitse erikseen liittää alaluokkaan, vaan se liittyy sinne automaattisesti.

Alaluokan ja yläluokan välinen riippuvuus kuvataan luokkakaaviossa alaluokasta yläluokkaan osoittavalla nuolella, jossa on iso kolmionmuotoinen kärki (ks. kuva 41).




Kuva 41: Yleistyshierarkian esittäminen

Alaluokkaan voidaan liittää yläluokalta perittyjen attribuuttien, palvelujen ja yhteyksien lisäksi omia attribuutteja, yhteyksiä ja palveluja. Esimerkiksi johtajalle voitaisiin lisätä attribuutit *vastuualue* ja *alaisten määrä*, joita ei henkilöillä yleisesti ole. Johtaja voitaisiin määritellä myös rooliin käyttäjä *edustustilin käyttöoikeus* -yhteyteen ja rooliin esimies *työnjohto*-yhteydessä (ks. kuva 42).




Kuva 42: Johtajan lisäattribuutit ja -yhteydet

Alaluokkaan johtaja voidaan liittää myös sellaisia palveluita, joita henkilöllä ei yleisesti ole. Tällaisia voisivat olla esimerkiksi *maksa edustustililtä*, *laadi luettelo alaisista*. Alaluokassa voidaan myös *syrjäyttää* (engl. override) yläluokassa määritelty palvelu. Syrjäyttämisellä tarkoitetaan palvelun sisällön tai toteutustavan uudelleenmäärittelyä, nimen säilyessä kuitenkin ennallaan. Esimerkiksi Henkilö-luokalle voisi olla määritelty palvelu *Viikkoraportti*, jonka sisältönä olisi

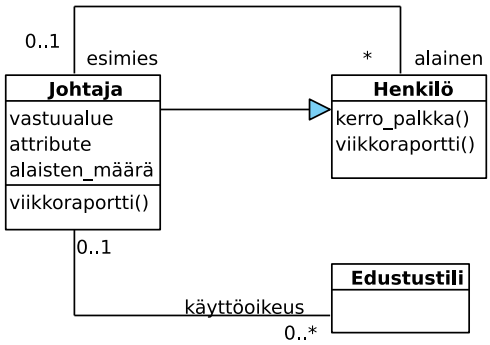
> *kerro ajankäyttö työtehtäviin.*

Tämä voitaisiin syrjäyttää Johtaja-luokassa siten, että johtajan *Viikkoraportti*-palvelun sisältö olisikin

> *kerro ajankäyttö työtehtäviin,*
> *laadi yhteenveto alaisten viikkoraporteista*
> *raportoi edustustilin käyttö.*

Olio-ohjelmoinnissa olio tietää oman luokkansa ja suorittaa palvelunsa oman luokkansa mukaisina. Niinpä olio-ohjelmassa miltä tahansa luokan henkilö tai sen alaluokan oliolta voidaan pyytää Viikkoraportti-palvelua. Jos olio ei ole johtaja, vaan tavallinen henkilö, hän kertoo oman ajankäyttönsä, mutta jos olio onkin johtaja, hän toimii johtajan Viikkoraportti-palvelun mukaisesti ja laatii lisäksi yhteenvedon alaisten viikkoraporteista ja raportoi edustustilin käyttönsä.

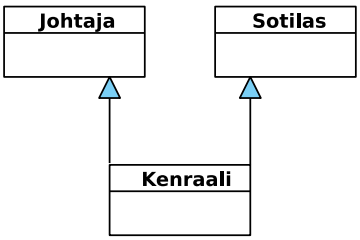
Mahdollisuus syrjäyttää palvelumäärityksiä on olio-ohjelmoinnin tärkeimpiä etuja perinteisiin ohjelmointikieliin verrattuna. Se lisää joustavuutta ohjelmakirjastojen käytössä, edistää ohjelmistokomponenttien uudelleenkäytettävyyttä ja muodostaa perustan erilaisten ohjelmistokehysten rakentamiselle (kehykset ovat eräänlaisia sovellusrunkoja, joissa toiminnan yksityiskohdat on jätetty esimerkiksi syrjäyttämisen avulla täsmennettäväksi). Syrjäyttämismahdollisuus onkin olio-ohjelmoinnin kannalta merkittävin syy yleistyshierarkian käyttöön. Sen hyväksikäyttömahdollisuudet tulevat kuitenkin usein esiin vasta laadittaessa teknisen tason suunnitelmaa ohjelmiston luokkarakenteesta. Luokkakaaviossa syrjäytetty palvelu kuvataan toistamalla palvelun nimi alaluokassa. Kuvassa 43 on henkilö-luokan palvelu *viikkoraportti* syrjäytetty luokassa johtaja. Sen sijaan palvelu *kerro_palkka* periytyy samansisältöisenä luokalta henkilö luokalle johtaja.




Kuva 43: Syrjäytetty palvelu viikkoraportti

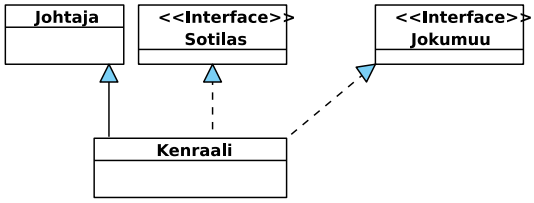
Kohdealuetta mallinnettaessa yleistyshierarkian tärkein käyttötilanne on kuvausvaivan säästäminen ja erilaisten sääntöjen täsmällinen ilmaiseminen. Jos usealla luokalla on samoja attribuutteja ja luokille voidaan määritellä yhteinen yläluokka, voidaan yhteiset attribuutit liittää yläluokkaan ja määritellä näin vain kertaalleen.

Tilannetta, jossa luokka on useamman kuin yhden luokan välitön alaluokka kutsutaan *moniperiytymiseksi* (engl. multiple inheritance) (ks. kuva 44).




Kuva 44: Moniperiytyminen

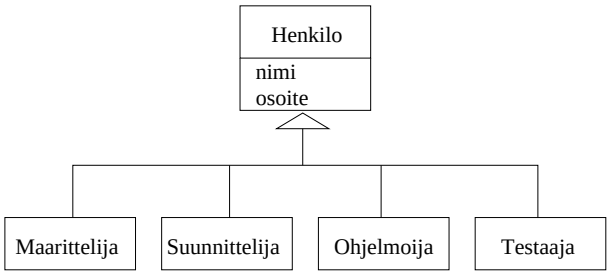
Moniperiytyminen voi olla kätevä tapa kohdealueen mallinnuksessa. Kaikki olio-ohjelmointikielet, esimerkiksi Java, eivät kuitenkaan tue moniperintää. Java ohjelmoinnissa hieman tätä vastaava rakenne on *rajapinnan periminen* (engl. interface realization). Rajapinnat ovat tavallaan abstrakteja luokkia, joilla ei ole määriteltyä tietosisältöä ja palveluista on annettu vain niiden määrittely. Tällöin rajapinnan perivässä (toteuttavassa) luokassa on syrjäytettävä kaikki rajapinnassa määritellyt palvelut (ks. kuva 45).




Kuva 45: Rajapintojen toteutus (palvelujen määrittelyt periytyvät)

## 3.7 Väärä- ja oikeaoppinen tapa soveltaa periytymistä

Tarkastellaan tilannetta, jossa mallinnetaan ohjelmistoyrityksen työntekijöitä. Ohjelmistoyrityksessä työskentelee henkilöitä eri työtehtävissä, *määrittelijöinä*, *suunnittelijoina*, *ohjelmoijina* ja *testaajina*. Nopeasti ajatellen kuvan 46 malli vaikuttaa hyvältä tavalta mallintaa tilanne.

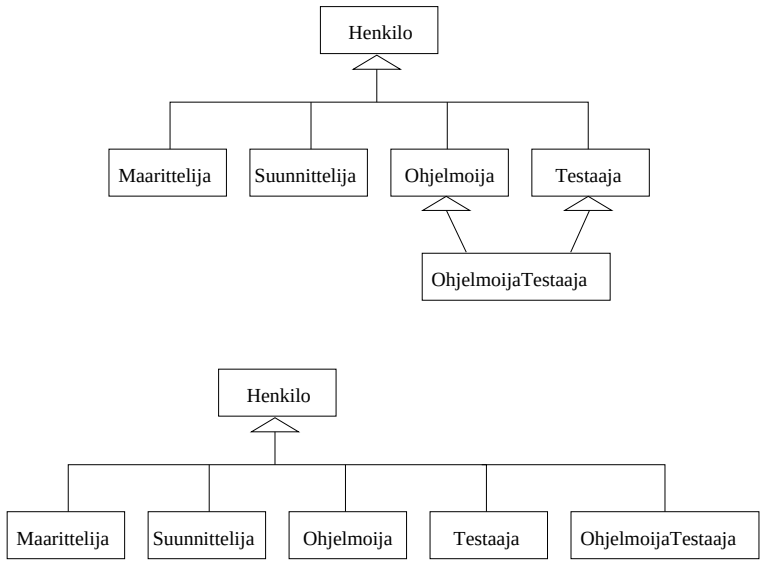



Kuva 46: Huono tapa mallintaa ohjelmistoyrityksen työntekijöitä

Malli vaikuttaa loogiselta, sillä kaikki työntekijäthän ovat Henkilöitä. Tämä on ainakin kuitenkin kahdella tapaa huono tapa tilanteen mallintamiseksi.

Ensimmäinen ongelma liittyy tilanteeseen, jossa on tarvetta mallintaa tilanne, jossa henkilö

toimii useissa työtehtävissä, esim. sekä ohjelmoijana että testaajana. Kaksi huonoa tapaa yrittää ratkaista tämä ongelma on esitetty kuvassa 47.




Kuva 47: Kaksi huonoa ratkaisuyritystä mallintaa kaksi työtehtävää omaava henkilö

Kuvan ylempi malli yrittää ratkaista ongelman moniperinnän avulla, eli uutta työtehtävää kuvaava luokkaa *OhjelmoijaTestaaja* perii luokat *Ohjelmoija* ja *Testaaja*. Kuten edellisessä luvussa todettiin, useat ohjelmointikielet, kuten Java eivät tue moniperintää, joten idea on huono.

Kuvan alemman mallin ratkaisu on uusi luokka *OhjelmoijaTestaaja* luokkahierarkiaan suoraan *Henkilö*-luokan alle. Nyt kierretään moniperintä, mutta ratkaisu on silti huono, sillä uusi luokka ei hyödynnä luokkien *Ohjelmoija* ja *Testaaja* määrittelyjä millään tavalla. Ongelmallista on myös se, jos yhdistelmätyötehtävien määrä kasvaa. Jokaisesta mahdollista kombinaatiosta tulisi uusi luokka henkilön alle.

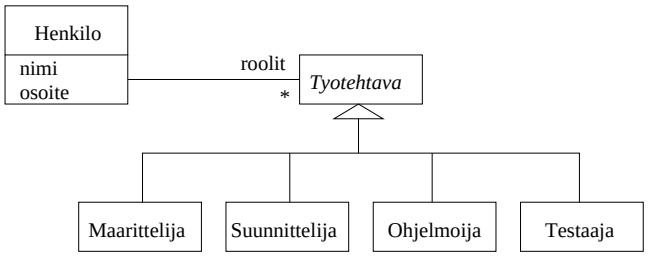
Toisen ongelman muodostaa tilanne, jossa henkilö vaihtaa työtehtävää. Useimmissa ohjelmointikielissä luokan tyyppi säilyy koko ohjelman suoritusajan samana, eli jos jostain on luotu *Ohjelmoija*, ei sama olio voi millään muuttua Testaaja-olioksi.

Näissä mallinnusyrityksissä on syyllistytty siihen, että henkilön *roolia* yrityksessä on yritetty mallintaa periytymishierarkian avulla. Monien kokeneiden oliomallintajien, esim. Peter Coadin mukaan tämä on huono, mutta valitettavan yleinen tapa soveltaa periytymistä (ks. esim. [6]).

Parempi tapa mallintaa tilannetta on pitää luokka *Henkilö* kokonaan erillisenä ja liittää työtehtävät, eli henkilön rooli, siihen erillisinä luokkina. Luokka *Henkilö* kuvaa siis henkilöä itseään ja sisältää ainoastaan henkilöön 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*.

Tilanne on esitetty luokkakaaviona kuvassa 48. Koska *Työtehtävä* on nyt ainoastaan kä-

site, jonka merkitys tarkentuu vasta sen perivissä luokissa, on se määritelty abstraktina luokkana, eli luokkana josta ei voi luoda ilmentymiä. UML:ssa abstraktien luokkien nimet kirjoitetaan kursiivilla.




Kuva 48: Parempi tapa mallintaa ohjelmistoyrityksen työntekijät

Oikea tapa siis on kuvata mahdolliset roolit periytymishierarkian avulla ja liittää sopivia rooleja tarpeen mukaan henkilöön. Näin ei ole ongelmaa sille, että henkilöllä on useita rooleja ja roolit vaihtuvat henkilö-olion olemassaolon aikana.

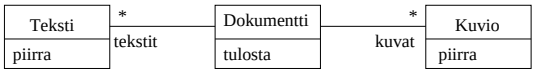
Peter Coadin ohje onkin, että olion roolit tulee kuvata yhteyksien, ei periytymisen avulla. Roolityypit voidaan sitten kuvata periytymishierarkialla [6].

## 3.8 Esimerkki monimutkaisemman rakenteen mallintamisesta

Dokumentti koostuu tekstielementeistä ja kuvioista. Kuvio voi olla joko piste, viiva, ympyrä tai joku näistä koostuva monimutkaisempi kuvio. Miten tilanne kannattaisi mallintaa?

Ensimmäinen versio mallista on kuvassa 49, eli dokumentti sisältää useita tekstejä ja kuvioita. Teksti ja kuviot on mallinnettu omina luokkinaan joihin dokumentista on yhteys. Luokan Dokumentti operaatio *tulosta* aiheuttaa jokaisen kuvan sekä tekstin piirtämisen näytölle:

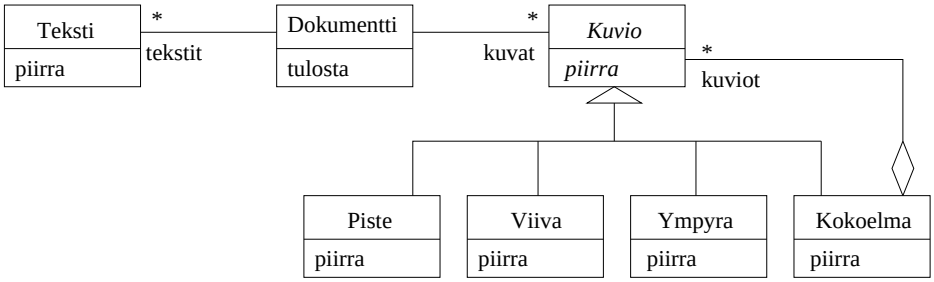
```
class Dokumentti{
   void tulosta(){
      for each k in kuvat  k.piirra();
      for each t in tekstit t.piirra();
   }
}
```




Kuva 49: Dokumentti koostuu tekstistä ja kuvioista

Tarkennetaan mallia luokan Kuvio osalta. Kuvio siis voi olla joko piste, viiva, ympyrä tai joku näistä koostuva monimutkaisempi kuvioiden kokoelma. Kuvio on siis yleiskäsite, jota konkreettiset käsitteet kuten piste, viiva ja ympyrä tarkentavat. Onkin luonnollista mallintaa kuvio periytymishierarkiana.

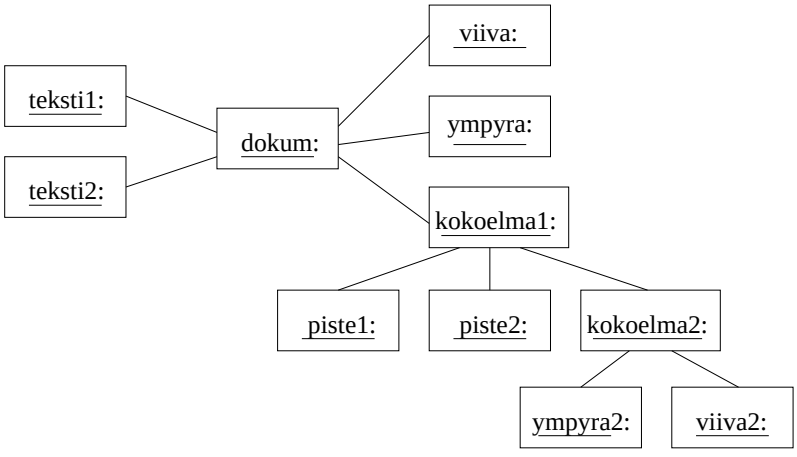
Kuvio voi olla myös kokoelma kuvioita. Termin kuvio määritelmä siis viittaa itseensä, eli on rekursiivinen. Tämä hiukan hankalalta vaikuttava tilanne kannattaa mallintaa kuvan 50 tapaan.




Kuva 50: Dokumentti koostuu teksteistä ja kuvioista. Kuvio voi olla yksinkertainen kuvio tai kokoelma kuvioita.

Kuviosta on nyt tehty abstrakti luokka, samoin kuvion operaatio *piirrä* on merkitty abstraktiksi, eli se on kirjoitettu kursiivilla. Operaation abstraktius tarkoittaa, että on operaatiosta on kiinnitetty ainoastaan nimi ja parametrit, mutta toteutusta sille ei ole annettu. Luokan erikoistavat aliluokat määrittelevät miten operaatio kussakin aliluokassa toteutetaan.

Jos kuvio on kokoelma, se *koostuu* useasta kuviosta, jotka voivat olla yksinkertaisia kuvioita (kuten viivoja) tai kokoelmia! Kuvan 51 oliokaavio ehkä helpottaa tilanteen ymmärtämistä. Kuvassa on dokumentti, joka koostuu kahdesta tekstistä sekä kolmesta kuviosta. Kuvioista kaksi ovat yksinkertaisia (*viiva* ja *ympyrä*) ja kolmas on kokoelma. Kokoelma taas koostuu kahdesta yksinkertaisesta kuviosta (*piste1* ja *piste2*) ja yhdestä kokokelmasta, joka taas koostuu kahdesta yksinkertaisesta kuviosta (*viiva2* ja *ympyrä2*).



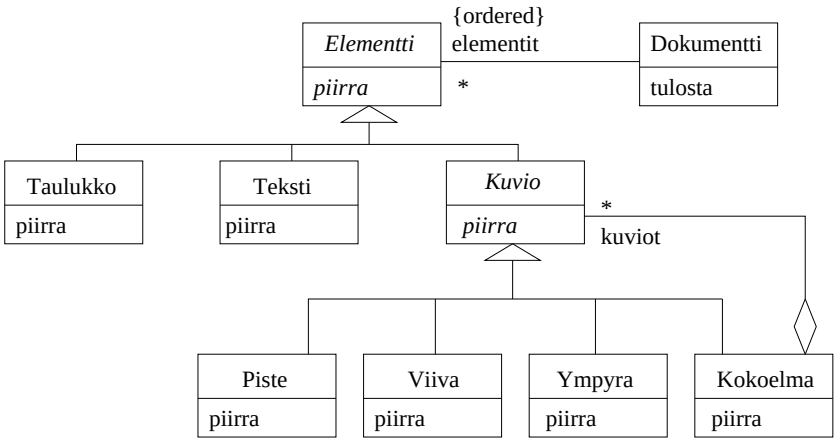

Kuva 51: Oliokaavio dokumentista, joka koostuu kahdesta tekstistä ja kolmesta kuviosta

Operaation *piirrä* toteutuksesta kannattaa mainita muutama sana. Aliluokissa *piste*, *viiva* ja *ympyra* operaatio toteutetaan todennäköisesti suoraan alla olevan ohjelmointiympäristön piirtokomennolla. Aliluokan *Kokoelma* piirto-operaatio sensijaan ainoastaan delegoi piirtovastuun kaikille sisältämilleen *Kuvio*-olioille. Eli kokoelmakuvio piirtää itsensä pyy-

tämällä kaikkien sisältämiensä kuvioiden pirtämään itsensä:

```
class Kokoelma extends Kuvio {
   void piirra(){
      for each k in kuviot  k.piirra();
   }
}
```

Mallia voitaisiin vielä yleistää hiukan. Dokumentissa voisi olla muunkinlaisia rakenneelementtejä kuin tekstiä ja kuvioita, esim. taulukoita. Voitaisiinkin ajatella, että dokumentti koostuu elementeistä. Elementti voi olla joko taulukko, teksti tai kuvio. Jotta dokumentin aikaansaamassa kokonaisuudessa olisi järkeä, ovat elementit dokumentissa tietyssä järjestyksessä. Kuvassa 52 on malli, missä nämä ajatukset on huomioitu. Elementtien järjestys merkitään yhteyteen liitetyllä määreellä *ordered*.




Kuva 52: Dokumentti koostuu erilaisista elementeistä. Elementit voivat olla taulukoita, tekstiä tai kuvioita ja elementeillä on järjestys.

Luokan Dokumentti operaatio *tulosta* näyttää nyt seuraavanlaiselta:

```
class Dokumentti {
   Elementti[] elementit;

   void tulosta(){
      for ( i=0 ; i < elementit.length; i++  )
        elementit[i].piirra();
   }
}
```

Dokumentti siis tuntee joukon elementtejä, jotka se tallettaa taulukkoon *elementit*. Taulukossa voi siis olla mitä elementtejä tahansa: tekstiä, taulukoita tai kuvioita. Dokumentin ei tarvitse edes tietää kunkin elementin oikeaa tyyppiä. Tulostusoperaatiossa kutsutaan

kullekin elementille operaatiota *piirrä*. Elementit piirretään taulukon määrittelemässä järjestyksessä. Kunkin elementin piirtyminen siis määrittyy elementin oikean tyypin mukaan.

Tässä luvussa esitetty tapa koosteisen tiedon esittämiseen on hyvin yleisesti tunnettu. Englanniksi tavasta käytetään nimitystä *composite pattern* [10].

# 4 Olioiden yhteistyön mallintaminen

Oliojärjestelmän toiminta perustuu olioiden yhteistyöhön. Yhteistyö ei tule esiin luokka- tai oliokaavioissa. Luokkakaaviossa esitetään ohjelman *staattinen rakenne*, eli luokkien attribuutit, operaatiot ja luokkien väliset suhteet. Vaikka luokkakaavioihin voidaan merkitä luokkien operaatiot, eli niiden tarjoamat palvelut, ei luokkakaaviosta kuitenkaan näy miten palvelun suoritus hyödyntää muita palveluita tai muita olioita. Yhteistyön selvittäminen on kiinteästi sidoksissa olioiden palveluiden määrittelyyn, sillä yhteistyö toteutuu palvelujen kautta. Samalla kun määrittelemme oliolle palveluja meidän tulisi ottaa kantaa siihen, millaista yhteistyötä olioiden välillä palvelun suorittamiseen liittyy. Palvelujen ja olioyhteistyön määrittely on ohjelmiston teknisen suunnittelun tehtäviä. Nämä tehdään siis myöhemmin kuin sovellusalueen kannalta keskeisten luokkien määrittely, joka tapahtuu jo ohjelmiston määrittelyvaiheessa.

UML tarjoaa kaksi tekniikkaa olioiden yhteistyön kuvaamiseen: sekvenssikaavion ja kommunikaatiokaavion. *Sekvenssikaaviossa* (engl. sequence diagram) keskitytään kuvaamaan operaatioiden suoritusjärjestystä ja kontrollin etenemistä oliolta toiselle. *Kommunikaatiokaavioissa* (engl. communication diagram) kuvataan erityisesti sitä, miten yhteistyö perustuu olioiden välisiin yhteyksiin. Suoritusjärjestys ja kontrollin eteneminen kuvataan myös, mutta ei yhtä helppolukuisesti kuin sekvenssikaavioissa.

Sekvenssikaaviot ja kommunikaatiokaaviot siis kummatkin ovat tarkoitettu ohjelman olioiden vuorovaikutuksen, eli ohjelman *dynaamisen toiminnan* mallintamiseen. On lähinnä makuasia kumpaa kaaviotyyppiä missäkin tilanteessa käyttää. Kannattaakin valita mallinnustilanteen kannalta luontevammalta tuntuva kaaviotyyppi. Sekvenssikaavio lienee kaavioityypeistä suositumpi.

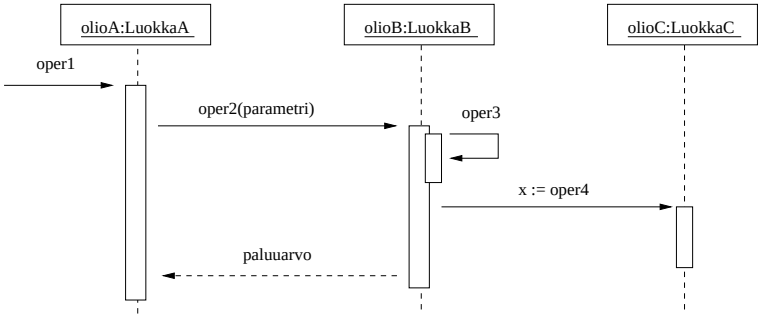
## 4.1 Sekvenssikaavio

Sekvenssikaavio kuvaa tietyn palvelun suorittamiseen liittyvän olioiden yhteistyön tai osan siitä. Kuvattava palvelu voi olla järjestelmän palvelu (käyttötapaus) tai johonkin luokkaan liittyvä palvelu. Sekvenssikaaviossa esitetään, mitä avustavia olioita palvelun suoritukseen osallistuu ja mitä näiden palveluita käytetään. Kaavio kuvaa myös missä järjestyksessä avustavien olioiden palveluja käytetään. Kuva 53 esittelee kaaviotekniikassa käytettäviä symboleja.

Sekvenssikaaviossa olio kuvataan suorakaiteena, jonka sisällä on olion tunnus. Tunnus kirjoitetaan samassa muodossa kuten oliokaavioissa, eli alleviivattuna muodossa *olioNimi:Luokka*, joista olion nimen tai luokan voi halutessaan jättää pois.

Suorakaiteesta alaspäin lähtevä katkoviiva kuvaa olion elinkaarta. Viivalla olevalla laatikolla eli *aktivaatiopalkilla* merkitään, että olio on aktiivisena eli olion jonkin operaation suoritus on menossa. Aika usein aktivaatiopalkki jätetään merkitsemättä, varsinkin jos sekvenssikaavioita piirretään paperille.

Olio pyytää avustavalta oliolta palvelua lähettämällä tälle viestin. Olio-ohjelmassa viestin lähettäminen tarkoittaa yleensä metodikutsua (esim. Javassa). Jatkossa käytetään olioille




Kuva 53: Esimerkki sekvenssikaaviosta

tapahtuvasta palvelupyynnöstä myös nimitystä operaatiokutsu. Kaaviossa viestin lähettäminen kuvataan lähettäjältä vastaanottajalle menevänä nuolena. Nuoleen liitetään tekstinä lähetettävä viesti. Yleensä tämä muodostuu pyydettävän palvelun nimestä ja pyynnön yhteydessä välitettävistä parametreista. Olioiden metodit voivat tuottaa paluuarvon. Paluuarvon palautus merkitään kaavioon tarpeen vaatiessa katkoviivallisena nuolena. Vaihtoehtoinen tapa merkitä paluuarvon vastaanotto on käyttää merkintää $vast := oper(param)$, joka tarkoittaa, että kutsuttavan metodin paluuarvo otetaan vastaan muuttujaan $vast$.

Kuva 53 alkaa tilanteesta, missä *olioA*:n palvelua *oper1* kutsutaan. Aika kuluu sekvenssikaaviossa alaspäin, eli mitä ylempänä viesti on, sitä aiemmin se on lähetty. *OlioA* kutsuu *olioB*:n palvelua *oper2*. Palvelukutsuun liittyy parametri. *OlioB* kutsuu ensin omaa palveluaan *oper3*, jonka jälkeen se kutsuu *olioC*:n palvelua *oper4* ottaen palvelun palauttaman vastauksen muuttujaan x. *OlioB* palauttaa lopussa paluuarvon *olioA*:lle. Huomaa, että *OlioB*:n paluuarvon palautus *OlioA*:lle on merkitty kuvaan katkoviivalla. Paluuarvon merkitseminen on siis vapaaehtoista.

Sekvenssikaaviossa aika siis etenee alaspäin. Sekvenssikaavion perusteella ei kuitenkaan voi tehdä johtopäätöksiä operaatiokutsujen kestoajoista. Kuvan 53 perusteella tiedetään esim. että *olioB*:n palvelun *oper*2 suoritus kestää varmuudella kauemmin kuin *olioC*:n palvelun *oper*4 suoritus. Vaikka kuvan perusteella näyttää, että *oper*2:n suoritusaika on noin kaksinkertainen *oper*4:n suoritusaikaan verrattuna, ei operaatioiden suoritusaikojen pituutta ei kuitenkaan voida päätellä suoritusten aktivaatiopalkkien pituudesta.[7]

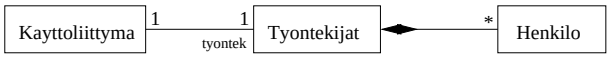
Käytimme kuvassa "täytettyä" eli suljettua nuolenpäätä palvelukutsujen yhteydessä. Sekvenssikaavioissa käytetään joskus myös avointa nuolenpäätä →. Suljetulla nuolenpäällä merkitty palvelukutsu tarkoittaa *synkronista* kutsua, eli kutsuja odottaa ja jatkaa vasta kun kutsu on suoritettu. Avoimella nuolenpäällä merkitty kutsu taas tarkoittaa *asynkronista* kutsua, eli kutsuja ei jää odottamaan operaation suoritusta vaan jatkaa välittömästi. Javalla ohjelmoinnissa kaikki metodikutsut ovat synkronisia operaatioita, eli kutsuja jatkaa vasta kun kutsuttu metodi on suoritettu.[8] Esim. tekstiviestin lähettäminen on asynkroninen toimenpide, eli lähettäjä jatkaa heti muihin toimiin kun tekstiviesti on lähetetty. Viesti menee perille lähettäjälle ennemmin tai myöhemmin. Erityisesti paperille piirrettävissä

[7] UML mahdollistaa operaatioiden kestoaikojen ym. aikainformaation merkitsemisen sekvenssikaavioihin. Tällä kurssilla aikainformaatiota ei kuitenkaan käsitellä.

[8] Jos ohjelma koostuu useista säikeistä, voivat jotkut metodikutsut olla asynkronisia.

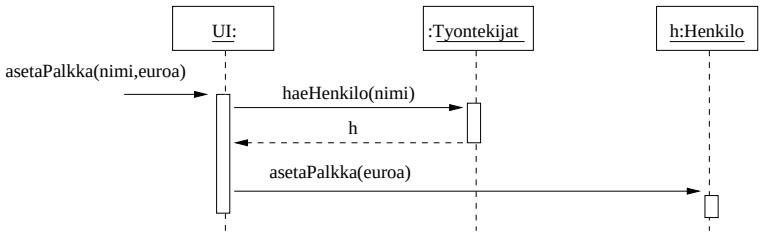
luonnosmaisissa sekvenssikaavioissa nuolenpäiden laatuun ei ole aina kiinnitetty huomiota, eli avoimesta nuolenpäästä huolimatta mallintaja saattaa joskus tarkoittaa synkronista palvelukutsua.

Tarkastellaan toisena esimerkkinä järjestelmää, joka pitää kirjaa yrityksen työntekijöistä. Osa järjestelmän luokkakaaviosta on kuvassa 54. Jokaisen työntekijän tiedot on koottu luokkaan *Henkilö*. Luokka *Työntekijät* sisältää kokoelman Henkilö-olioita ja järjestelmää käytetään *käyttöliittymä*-olion kautta.




Kuva 54: Osa työntekijöiden hallintajärjestelmän luokkakaaviosta

Kuvassa 55 on esitetty sekvenssikaaviona tilanne, jossa asetetaan parametrina annetulle henkilölle henkilölle palkka. Käyttöliittymäolio kutsuu ensin Työntekijät-luokan olion operaatiota, joka palauttaa viitteen nimellä haettuun Henkilö-olioon. Sekvenssikaaviossa paluuarvo *h* viittaa etsittyyn Henkilo-olioon. Saatuaan viitteen, kutsuu käyttöliittymä löydetyn Henkilö-olion operaatiota, joka asettaa uuden palkan.




Kuva 55: Palkkatiedon päivitys

Luokan *Käyttöliittymä* metodin *asetaPalkka* ohjelmakoodi voisi näyttää tilanteessa seuraavanlaiselta:

```
Class Kayttoliittyma{
   Tyontekijat tyontek;   // viite luokan Tyontekijat olioon

   void asetaPalkka(String nimi, int euroa){
      Henkilo h = tyontek.haeHenkilo();
      h.asetaPalkka(euroa);
   }
}
```

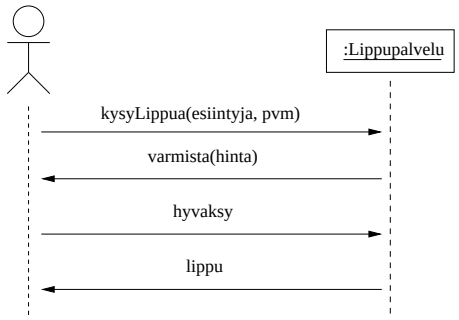
## 4.2 Järjestelmätason sekvenssikaaviot

Sekvenssikaavioita siis käytetään usein ohjelmiston teknisessä suunnittelussa hahmoteltaessa olioiden yhteistyötä. Sekvenssikaavioita voidaan käyttää myös vaatimusmäärittelyn yhteydessä kuvaamaan käyttötapauksiin liittyvää käyttäjän interaktiota järjestelmän kanssa. Tällöin vuorovaikutus siis tapahtuu käyttötapauksen käyttäjien ja järjestelmän kesken, eli koko järjestelmä nähdään yhtenä oliona tai "mustana laatikkona".

Tarkastellaan esimerkkinä yksinkertaista lippupalvelujärjestelmää ja sen käyttötapausta *Lipun varaus, tilanne missä lippuja löytyy.* Käyttötapauksen kulku:

1. Käyttäjä kertoo esiintyjän ja esityspäivän
2. Järjestelmä kertoo, minkä hintainen lippu on mahdollista ostaa
3. Käyttäjä hyväksyy lipun
4. Käyttäjälle annetaan tulostettu lippu

Käyttötapauksen kulku on esitetty kuvan 56 sekvenssikaaviossa. Huomaa, että suorituksen kontrollia ilmaisevia aktivaatiopalkkeja ei ole merkitty kuvaan.




Kuva 56: Järjestelmätason sekvenssikaavio

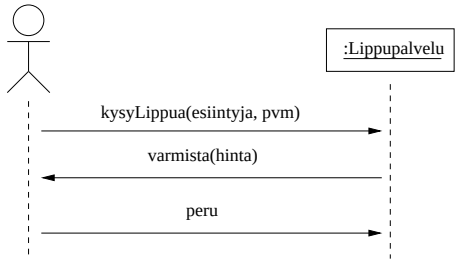
Normaalisti sekvenssikaavio kuvaa ainoastaan yhden suoraviivaisesti etenevän tapahtumaketjun. Jos halutaan kuvata vaihtoehtoisia tapahtumaketjuja, tarvitaan useita sekvenssikaavioita.

Lipunvarausjärjestelmän toisen käyttötapaksen *Lipun varaus, tilanne missä lippuja löytyy, mutta käyttäjä peruu ostoksen* kulku on seuraava:

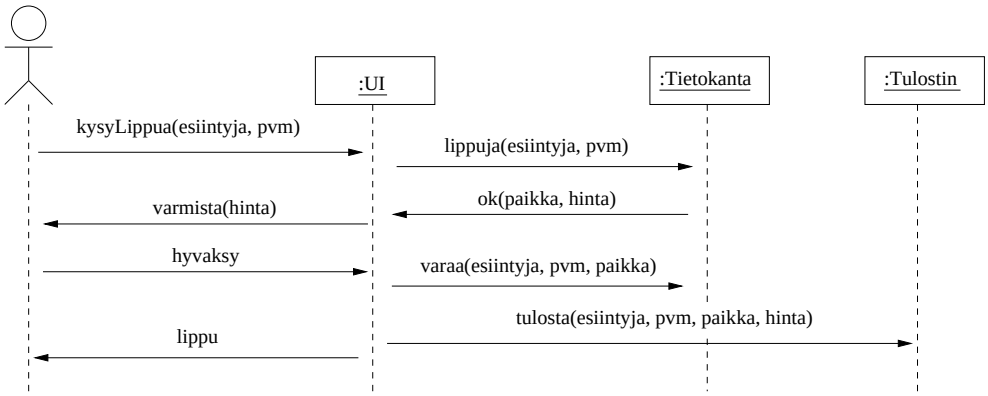
1. Käyttäjä kertoo esiintyjän ja esityspäivän
2. Järjestelmä kertoo, minkä hintainen lippu on mahdollista ostaa
3. Käyttäjä peruu ostoksen

Kuvassa 57 käyttötapausta vastaava sekvenssikaavio. Hetken päästä tutustumme menetelmään, joka mahdollistaa valinnaisuuden kuvaamisen sekvenssikaaviossa. Valinnaisuutta hyväksikäyttäen esimerkkimme molemmat käyttötapaukset olisi voitu kuvata yhden sekvenssikaavion avulla.

Vaatimusmäärittelyssä tehtävät järjestelmätason sekvenssikaaviot tarkentuvat ohjelman teknisessä suunnittelussa. Kuvassa 58 hahmotelma lippupalvelujärjestelmän suunnitteluvaiheen sekvenssikaaviosta. Edellisten kuvien järjestelmä on nyt tarkentunut kolmeksi olioiksi, eli käyttöliittymäksi, tietokannaksi ja tulostimeksi. Huomaa, miten parametreja välitetään eri olioiden kesken. Oliden tekniseen suunnitteluun palaamme tarkemmin monisteen luvussa 6.




Kuva 57: Järjestelmätason sekvenssikaavio, vaihtoehtoinen skenaario



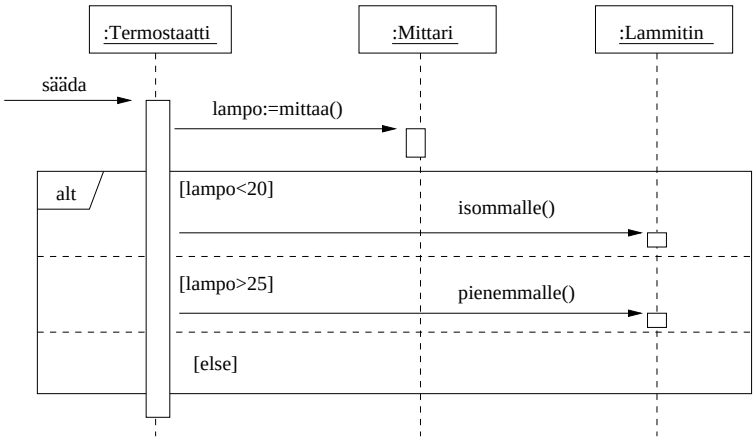

Kuva 58: Suunnittelutason sekvenssikaavio

## 4.3 Toisto ja valinnaisuus sekvenssikaavioissa

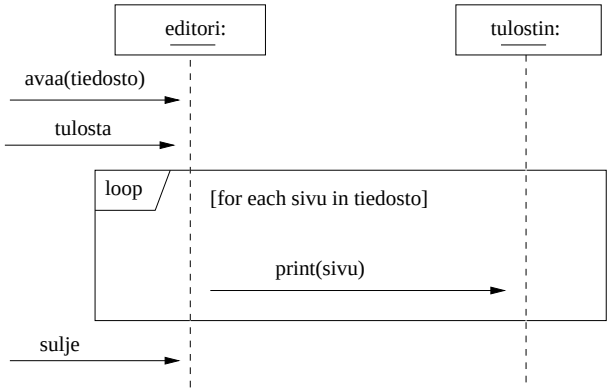
Sekvenssikaavioihin voidaan tarvittaessa sisällyttää valinnaisuutta. Kuvassa 59 termostaatti selvittää ensin lämpömittarilta lämpötilan. Jos lämpötila on liian alhainen laitetaan lämmitin isommalle, jos taas lämpötila on liian korkea, laitetaan lämmitin pienemmälle. Valinnaisuutta kuvaa erillinen lohko, jonka nurkassa on tunniste *alt* (lyhenne sanasta alternative). Lohko on jaettu vaihtoehtoisiin osiin, ja se osa suoritetaan, jota vastaava ehto on tosi. Ehto esitetään hakasulkeissa. Jos mikään muu ehdon haara ei ole tosi, suoritetaan else-osa. Eli kuvassa lämmittimelle ei tehdä mitään jos on lämpötila välillä 20-25.

Myös toiston kuvaaminen on mahdollista. Toisto esitetään lohkolla, jonka ylälaidassa on tunniste *loop*. Toistoehto kirjoitetaan hakasulkeisiin. Kuvassa 60 editori lähettää tiedostoa tulostimelle rivi riviltä. Tulostusehdoksi on kirjoitettu *for each sivu in tiedosto* ja toistettava viesti on *print(sivu)*, eli jokainen tiedoston sivu lähetetään tulostimelle yksitellen print-operaation parametrina.

Ehdoissa käytettävät merkintätavat ovat mallintajan vapaasti valittavissa. Pääperiaate on tehdä ehdoista mahdollisimman selkeät. Tässä esimerkissä ehdoksi olisi voitu kirjoittaa yhtä hyvin esim. *[tulostetaan tiedoston kaikki sivut]*.




Kuva 59: Valinta sekvenssikaaviossa




Kuva 60: Toisto sekvenssikaaviossa

## 4.4 Uudet ja tuhoutuvat oliot sekvenssikaaviossa

Kuva 61 tarkentaa edellistä esimerkkiä. Nyt myös editoitava tiedosto esitetään omana oliona. Editorin saadessa tiedoston avauskomennon, luo se tiedostoa edustavan olion. Tietosto-olio lataa tiedoston sisällön levyltä kutsumalla luokassa itsessään määriteltynä olevaa lataa-operaatiota. Kuvasta näemme, miten sekvenssikaavioon merkitään uuden olioiden luominen. Skenaarion aikana syntyviä olioita ei siis merkitä ylälaitaan, vaan uusi olio tulee kaavioon sille korkeudelle, missä olion synnyttävä viesti on. Olion luovan operaation voi nimetä kostruktorikutsun tapaan.

Tämän jälkeen toistorakenne kuvaa, että käyttäjä editoi tiedostoa niin kauan kuin haluaa. Muutokset päivittyvät muistissa olevaan tiedosto-olioon.

Editoinnin jälkeen tiedosto tulostetaan. Editori välittää tulostuskomennon tiedosto-oliolle, joka tulostaa itsensä pyytämällä tulostinta tulostamaan jokaisen sivun.

Kun käyttäjä pyytää tiedoston tallennusta, vie tiedosto-olio sisältönsä levylle siinä tapauksessa, että sisältö on muuttunut. Kuten kuvasta 61 nähdään, tallennuksen ehdollisuus esitetään lohkolla, jonka tunnisteena on *opt*, eli jos ehto toteutuu, suoritetaan lohkon sisältö, muuten hypätään sen yli.

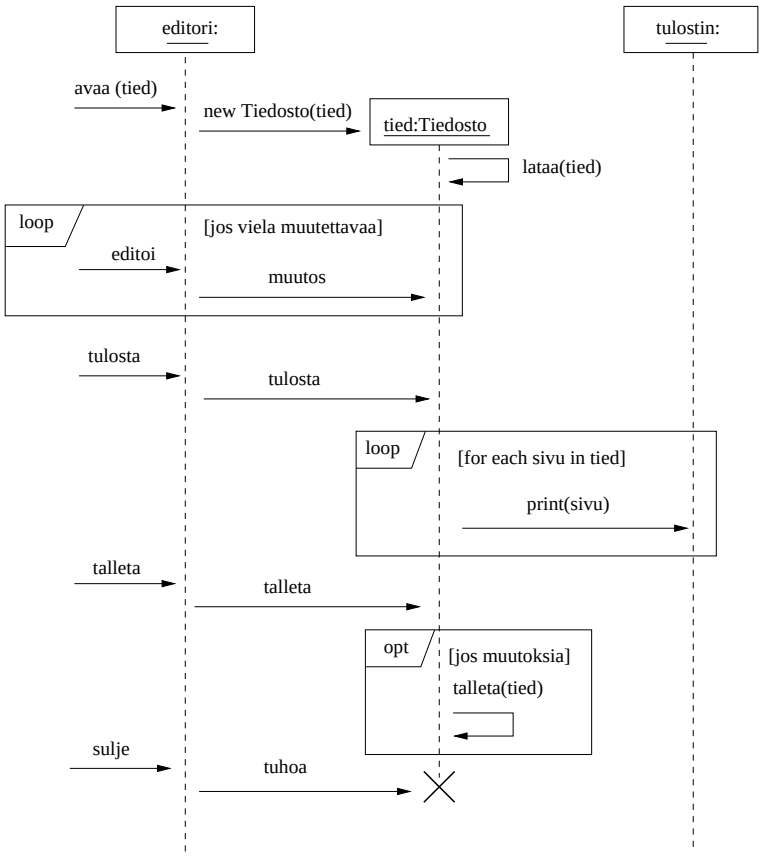
Lopussa käyttäjä sulkee tiedoston. Editori tuhoaa tiedosto-olion muistista. Olion tuhoutuminen merkitään olion elinviivan päättävänä rastina.

## 4.5 Takaisinmallinnus

Yksi UML:n hyödyllisimpiä käyttötarkoituksista on *takaisinmallinnus* (engl. reverse engineering) eli kaavioiden laatiminen valmiista ohjelmakoodista lähtien. Tämä on tarpeen esim. opeteltaessa ymmärtämään huonosti dokumentoitua ohjelmaa, jota on tarve ylläpitää.

Seuraavassa esimerkki Javalla ohjelmoidusta kellosta. Kello koostuu viisareista. Seuraavassa viisarin koodi:

```
class Viisari{
  int max;      // arvo, jonka saavutettuaan viisari pyörähtää nollaan
  int arvo;

  Viisari( int m ){ arvo = 0;  max = m; }

  void etene(){
    arvo++;
    if ( arvo==max ) arvo = 0;
  }

  int aika(){ return arvo; }
}
```

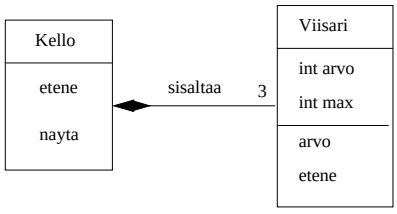



Kuva 61: Toistoa ja valinnaisuus

Kello luo konstruktorissaan kolme viisaria. Kellon metodi *etene* vie aikaa yhden sekunnin eteenpäin. Riippuen ajasta tämä saattaa vaatia useamman viisarin edistämisen. Eli normaalisti vain sekuntiviisari etenee. Jos sekuntiviisari pyörähtää nollaan etenemisen yhteydessä, myös minuuttiviisari etenee, jne .... Kutsuttaessa metodia *nayta* kello kysyy kunkin viisarin arvon ja tulostaa ajan ruudulle.

```
class Kello {
  Viisari tunti;
  Viisari minuutti;
  Viisari sekunti;

  Kello(){
    sekunti  = new Viisari(60);
    minuutti = new Viisari(60);
    tunti    = new Viisari(24);
  }

  void etene(){
    sekunti->etene();
    if ( sekunti->aika()==0 ) {
      minuutti->etene();
      if ( minuutti->aika()==0 )
        tunti->etene();
    }
  }

  void nayta(){
    System.out.print( tunti->aika() );   System.out.print(":");
    System.out.prin( minuutti->aika() ); System.out.print(":");
    System.out.print( sekunti->aika() );
  }
}
```

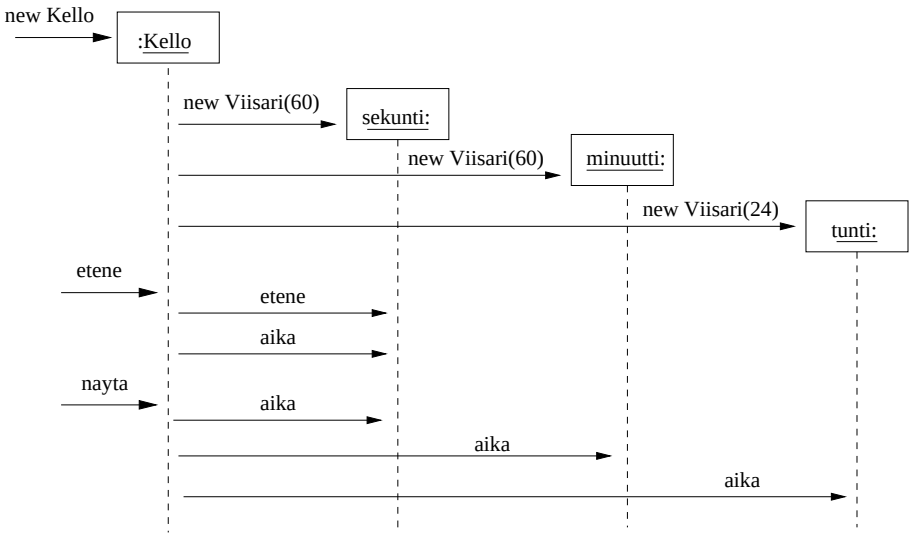
Kuvassa 62 Kellon luokkakaavio. Luokkakaavio ei kerro juuri mitään ohjelman toiminnan logiikasta ja sekvenssikaavioiden merkitys on luokkakaaviota tärkeämpi ohjelman toiminnan ymmärtämisen kannalta.




Kuva 62: Kellon luokkakaavio

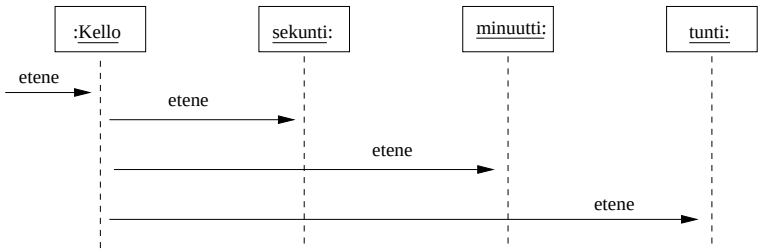
Kuvassa 63 sekvenssikaavio, jossa on näytetty kellon ja viisareiden luonti, eteneminen sekunnilla ja kellon arvon tulostus. Sekvenssikaavioon on merkitty ainoastaan ohjelman sovellusolioiden keskinäiset metodikutsut. Kello-luokan oliohan kutsuu myös Javan standar-

dikirjaston System.out-olion metodia print, mutta kutsu on jätetty merkitsemättä koska se on tässä yhteydessä epäoleellinen.




Kuva 63: Kellon eteneminen ja ajan näyttäminen sekvenssikaaviona

Kuvassa 64 vielä tilanne, jossa kello etenee siten, että sekä sekuntti, minuutti että tuntiviisari pyörähtävät nollaan. Huomaa, että kuvaan ei ole merkitty viisareiden aika-metodien kutsuja. Sekvenssikaavioon kannataakin aina merkitä vain sen verran tietoa, mikä on luettavuuden ja ymmärrettävyyden kannalta tarpeen.




Kuva 64: Kellon etenemistä tasatunnilla kuvaava sekvenssikaavio

Takaisinmallinnuksessa kannattaa edetä yleensä siten, että luokkamallia ja sekvenssikaavioita piirretään rinnakkain esim. suorittamalla ohjelmaa askel askeleelta debuggerilla ja samalla piirtäen sekvenssikaavioon ohjelman etenemistä ja täydentäen luokkamalliin suorituksessa käytettäviä luokkia.

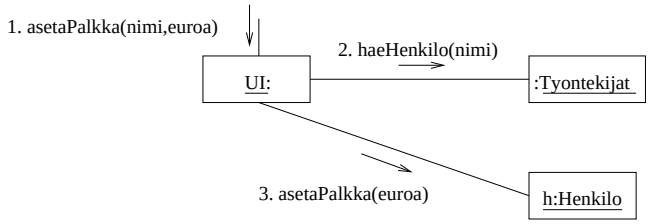
Useinkaan ei ole tärkeää, että esim. takaisinmallinnettu luokkamalli sisältää ohjelman kaikki luokat ja niiden väliset suhteet. Tärkeintä takaisinmallinnuksessa on päästä selville ohjelman toteutuksen logiikasta ja havaita mitkä luokat ja metodit ovat kulloisenkin käyttötarpeen kannalta oleellisia.

Jotkut sovelluskehittimet pystyvät luomaan ainakin luokkamalleja annetusta koodista automaattisesti.

## 4.6 Kommunikaatiokaavio

Vaihtoehtoinen tapa esittää olioiden yhteistoimintaa on *kommunikaatiokaavioiden* (engl. communication diagram) käyttö.[9] Sekvenssikaavioiden tapaan kommunikaatiokaaviot kuvaavat yksittäisen tapahtumaskenaarion aikaisen olioiden yhteistoiminnan.

Kuvassa 65 on esitetty kommunikaation avulla sivun 54 työntekijähallintajärjestelmäesimerkin tilanne, jossa asetetaan parametrina annetulle henkilölle henkilölle palkka.

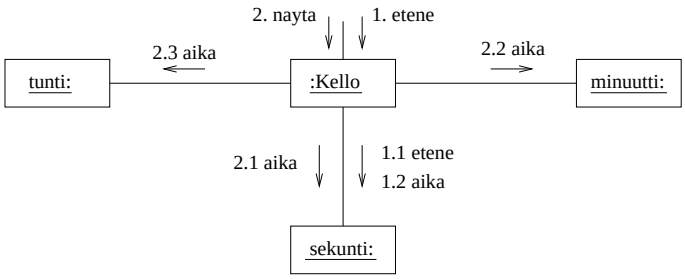



Kuva 65: Palkkatiedon päivitys kommunikaatiokaaviona

Kommunikaatiokaaviossa keskitytään kuvaamaan miten olioiden yhteistyö hyödyntää olioiden välisiä yhteyksiä. Kaavio esittelee tiettyyn suoritukseen liittyvät oliot ja niiden yhteydet sekä suoritukseen liittyvät operaatiokutsut. Operaatiokutsujen ajallinen järjestys selviää numeroinnista.

Kuvassa 66 kellon toimintaa esittelevä kommunikaatiokaavio (vastaava sekvenssikaavio kuvassa 63). Esimerkissä on käytetty operaatioille vaihtoehtoista numerointitapaa, eli hierarkista numerointia. Skenaario alkaa operaatiolla *etene*, joka aiheuttaa sekuntiviisarin kasvatuksen, eli viisari suorittaa operaatiot *etene* ja *aika*. Kelloon kohdistuva operaatio on numeroltaan 1 ja sen aiheuttamat kellon suorittamat operaatiokutsut on numeroitu 1.1 ja 1.2. Seuraavaksi suoritetaan kellon operaatio *nayta*, joka aiheuttaa jokaiselle viisarille operaation *aika*. Nämä on numeroitu $2, 2.1, 2.2$ ja $2.3$

Käytettäessä kommunikaatiokaaviossa hierarkista operaationumerointia, suoritusjärjestys siis on $1, 1.1, 1.2, \ldots, 2, 2.1, 2.2, \ldots, 3., 3.1, 3.2, \ldots$. Ideana on, että ylemmän tason operaation aiheuttamat olioiden vuorovaikutukset numeroidaan ylätason numeron alle tulevana numerointina.




Kuva 66: Kellon toimintaa kuvaava kommunikaatiokaavio

[9]Kommunikaavioiden nimi oli vanhan UML-standardin mukaan yhteistoimintakaavio (engl. collaboration diagram).

Kommunikaatiokaavio on sekvenssikaaviota parempi valinta yhteistoiminnan kuvaamiselle niissä tapauksissa, missä halutaan korostaa olioiden välisiä yhteyksiä. Vaikka kommunikaatiokaaviokin sisältää tiedon operaatioiden suoritusjärjestyksestä, tulee asia selkeämmin esille sekvenssikaaviossa. Jos viestejä on paljon, saattaa kommunikaatiokaavio olla kompaktimpi esitysmuoto kuin sekvenssikaavio.

# 5 UML:n soveltaminen ohjelmiston suunnittelussa

Edellisessä luvussa käsittelimme UML:n eri kaaviotyyppien yksityiskohtia. Seuraavassa luvussa tarkastellaan, miten eri kaavioita voidaan soveltaa tilanteessa, jossa edetään tehtäväkuvauksesta vaatimusmäärittelyyn ja suunnitteluun. Luku esittelee myös uuden kaaviotyypin, *pakkauskaavion* (engl. package diagram).

On olemassa lukuisia kirjoja, joissa esitellään menetelmä olioperustaiseen ohjelmistokehitykseen, eli *olioanalyysi* (engl. object oriented analysis, OOA), *oliosuunnittelu* (engl. object oriented design, OOD) ja toteutushahmotelma *olio-ohjelmointi*kielellä (engl. object oriented programming OOP).

Eri lähteissä (esim. [11, 3, 18, 16, 22]) esitellyt menetelmät sisältävät paljon samaa, poiketen joissain yksityiskohdissa ja painotuksissa. Noudatamme seuraavassa luvussa hyvin pitkälti Craig Larmanin kirjan *Applying UML and patterns, An Introduction to Object-Oriented Analysis and Design and Iterative Development* [16] esittelemää menetelmää. Emme tosin voi kurssin suppeuden takia käydä läpi menetelmästä muuta kuin perusteet. Larmanin kirja on monien arvioijien mielestä seikkaperäisin aloittelijalle tarkoitettu johdanto aihepiiriin. Kirjan vahvuus on erityisesti olioiden suunnittelussa, eli erittäin haastavassa aiheessa, joka monissa aloittelijoille suunnatuissa teoksissa sivuutetaan hyvin nopeasti.

Huomionarvoista on, että mikään menetelmä, tässäkään monisteessa esiteltävä, ei toimi kaikissa tilanteissa. Menetelmien tunteminen on kuitenkin eduksi, sillä menetelmiin sisältyy usein tiivistyneenä kokeneiden ohjelmistosuunnittelijoiden hyviksi havaitsemia toimintatapoja. Aloittelijan on vaikea lähteä liikkeelle ilman mitään konkreettista ohjeistusta ja jonkun menetelmän on oltava se ensimmäinen, jonka kautta oliosuunnitteluun tutustuminen aloitetaan.

# 6 Kirjaston tietojärjestelmä

Tavoitteena on määritellä ja suunnitella tietojärjestelmä, jonka avulla hallitaan kirjaston lainaustapahtumia.[10]Järjestelmä on alkuvaiheessa tarkoitettu ainoastaan yksittäisen kirjaston käyttöön. Järjestelmä toteutetaan ketterien menetelmien hengessä eli iteratiivisesti, siten että 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. Tässä monisteessa käydään läpi ainoastaan ensimmäisen iteraation vaatimusmäärittely- ja suunnitteluvaihe.

## 6.1 Järjestelmän toiminnalle asetettuja vaatimuksia

Asiakasta haastattelemalla on kerätty lista järjestelmältä toivotusta toiminnallisuudesta:

[10]Tässä luvussa esitetty kirjastojärjestelmä on mukaelma kirjan [8] sovellusesimerkistä.

- 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

### 6.1.1 Sanasto

Vaatimusten analysoinnin yhteydessä kannattaa laatia *sanasto* (engl. glossary) sovelluksen kohdealueen keskeisistä termeistä. Sanasto laaditaan esim. asiakkaan haastattelujen pohjalta. Pelkän termin nimen lisäksi sanastossa voidaan tarvittaessa tarkentaa termin merkitystä ja termien suhteita. Useimmat sanaston termeistä tulevat aikanaan löytymään sovelluksen kohdealueen luokkamallista.

Edellisen listan pohjalta laadittu keskeisten termien sanasto on seuraavassa:

**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.

**Kirjasto**

Paikka jossa kirjat ovat ja jonka toimintaa tuotettava järjestelmä tehostaa.

**Kirjastonhoitaja**

Järjestelmän käyttäjä, tekee lainat, varaukset ja palautukset.

**Laina**

Lainaus kohdistuu tietyyn fyysiseen kirjaan.

**Varaus**

Tiettyyn nimikkeeseen kohdistuva varaus.

**Palautus**

Tietyn fyysisen kirjan palautus.

**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 lainoja ja palautuksia suoraan automaatin avulla.

Suurin osa termien merkityksestä on itsestäänselvää, ainoa huomionarvoinen seikka on termien *nimike* ja *kirja* suhde.[11]

### 6.1.2 Käyttötapaushahmotelmat

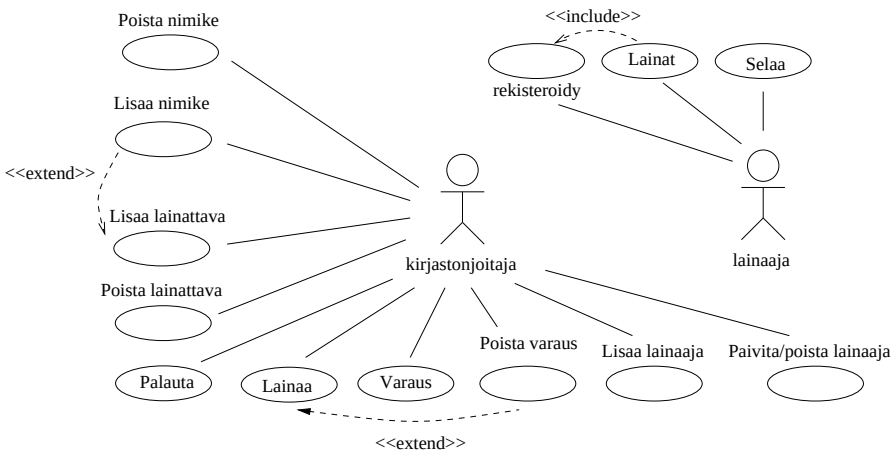
Järjestelmän käyttäjät ovat siis *kirjastonhoitaja* ja *lainaaja*. Järjestelmän käyttötapauksiksi tunnistetaan aluksi seuraavat:

- 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 kuvassa 67.

Tässä tapauksessa käyttötapausten suhteet ja käyttäjät ovat melko selkeät, joten käyttötapauskaavion tarjoama informaatioarvo on hyvin vähäinen. Yleensäkin käyttötapauskaavio ei sinäänsä kerro paljoakaan ja ratkaisevaa käyttötapauksissa on tekstuaalinen kuvaus. Kaavio kannattaa kuitenkin piirtää, sillä se tarjoaa hyvän yleiskuvan järjestelmän tarjoamasta toiminnallisuudesta.

[11] Termit nimike ja kirja eivät välttämättä ole täysin onnistuneesti valittu. Kirjan sijasta olisi ehkä ollut parempi käyttää termiä nide.




Kuva 67: Kirjaston käyttötapauskaavio

## 6.2 Vaatimusmäärittely - iteraatio 1

Kuten luvussa 1.1.4 todettiin, ketterässä ohjelmistotuotannossa järjestelmä rakennetaan iteratiivisesti, eli ensimmäiseen iteraatioon valitaan osa järjestelmän halutusta toiminnallisuudesta, joka määritellään, suunnitellaan ja toteutetaan. Tämän jälkeen valitaan asiakkaan toiveiden mukaisesti seuraaville iteraatiolle taas lisää toiminnallisuutta toteutettavaksi.

### 6.2.1 Käyttötapaukset

Valitaan ensimmäisessä iteraatiossa toteutettavaksi seuraavat käyttötapaukset:

- Lainaa kirja
- Palauta kirja
- Lisää nimike
- Lisää kirja
- Lisää lainaaja

Ensimmäiseen iteraatioon kannattaa yleensä valita toteutettavaksi järjestelmän ydintoiminnallisuus. Tällöin asiakas näkee hyvin pian onko järjestelmän kehitystyö lähtenyt liikkeelle oikeaan suuntaan ja vaatimuksia voidaan tarkentaa tarpeen mukaan jo ennen toista iteraatiota.

Seuraavaksi tarkennetaan iteraatioon valitut käyttötapaukset. Yhden iteraation aikana ei ole välttämätöntä toteuttaa käyttötapauksien toimintaa täydessä laajuudessaan. On myös mahdollista määritellä käyttötapauksesta perustoiminnallisuus, jota myöhemmissä iteraatioissa mahdollisesti laajennetaan. Rajoitumme ensimmäisessä iteraatiossa tilanteeseen, missä ohjelma toimii kokonaisuudessaan keskusmuistissa, eli levylle ei talleteta mitään. Ohjelman tietojen tallentaminen levylle (esim. tietokantaan) toteutetaan vasta myöhemmissä iteraatioissa.

Todellisuudessa käyttötapausten tarkentaminen kannattaa tehdä yhdessä asiakkaan kanssa. Koska useimmissa tapauksissa toiminta on niin ilmeistä, ei esi- ja jälkiehtoja ole kirjattu. Ensimmäisessä iteraatiossa osa käyttötapauksista ei vielä kata kaikkea mahdollista käyttötapaukseen liittyvää toiminnallisuutta, esim. poikkeusten osalta. Käyttötapauksia tarkennetaan tarvittaessa myöhempien iteraatioiden aikana. Kaikissa käyttötapauksissa ainoana käyttäjänä on kirjastonhoitaja.

**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

*Poikkeusellinen 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

*Huomioita:* Täytyykö vaiheita 1 ja 2 tehdä? Voi olla myöhemmin tarpeen, jos esim. kirjaan on varaus.

**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

*Poikkeusellinen 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.

### 6.2.2 Muut vaatimukset

Kaikkia vaatimuksia ei ole luontaista ilmaista käyttötapauksina. Tälläisiä ovat esim.

- *ei-toiminnalliset vaatimukset* (engl. non-functional requirements), kuten
  - käytettävyydelle asetetut vaatimukset

  - suorituskykyyn liittyvät ominaisuudet (esim. järjestelmä pystyy tallentamaan 100000 kirjan tiedot, järjestelmä suoriutuu sadasta lainaustapahtumasta minuutissa)
  - suoritusympäristö (esim. toimii Windows XP -, Vista - ja 7 -käyttöjärjestelmillä)
  - toteutusympäristö (esim. toteutettu Javalla, käyttöliittymä tehty Swing-komponenttikirjastoa käyttäen, tietokantana MySQL)
- toiminnot, jotka eivät liity erityisesti mihinkään käyttötapaukseen, esim. vaatimus kaikkien tapahtumien kirjaamisesta loki-tiedostoon

Oikeassa projektissa näihin on otettava kantaa alusta lähtien, nyt ainoa ei-toiminnallinen vaatimus on määritellä toteutuskieleksi Java.

### 6.2.3 Kohdealueen luokkamalli

Käyttötapausten hahmottelun jälkeen laaditaan sovelluksen kohdealueen alustava luokkamalli. Tässä vaiheessa luokkamalli ainoastaan jäsentää ongelma-aluetta ja toteutukseen ei vielä oteta mitään kantaa.

Ensin tehdään lista luokista. Potentiaalisia luokkia voi etsiä monella tavalla. Perinteinen tapa on substantiivien etsiminen järjestelmän vaatimusten kuvauksesta ja käyttötapauksista (ks. luku 3.3). Jos vaatimusten analysoinnin yhteydessä on laadittu sanasto, siitä todennäköisesti saadaan suoraan suurin osa luokkakandidaateista. On myös laadittu tiettyjä yleisiä kategorioita, joiden edustajia sovellusten ongelma-alueilla yleensä on (katso esim. Larmanin kirjan [16] luku 9.5.). Molempia tapoja kannattaa käyttää ja listata kaikki potentiaaliset luokat, joista voi sitten karsia ylimääräiset.

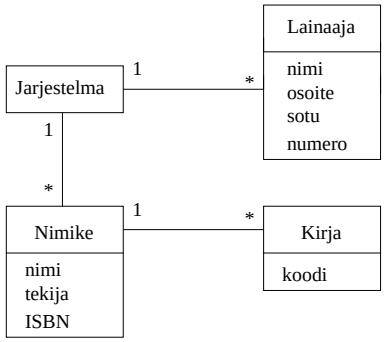
Yleisestä vaatimuslistasta, sen pohjalta laaditusta sanastosta ja käyttötapauksista löytyvät seuraavat substantiivit (tässä vaiheessa on jätetty pois ne käsitteet, jotka eivät liity ensimmäisen iteraation aikana toteutettavaan toiminnallisuuteen):

- kirjasto
- kirjastonhoitaja
- **lainaaja**
- **kirja**
- koodi
- tunniste
- **laina**
- **nimike**
- **järjestelmä**

Tummentamattomat luokkaehdokkaat on jätetty pois seuraavista syistä: Kirjasto on paikka, johon järjestelmä sijoitetaan, joten sitä ei tarvinne mallintaa. Kirjastohoitaja on järjestelmän käyttäjä, ainakaan tässä vaiheessa ei ole syytä esim. pitää kirjaa erillisistä kirjastonhoitajista. Koodi ja tunniste erittelevät kirjan ja lainaajan, eli ne eivät ole luokkia, vaan pikemminkin luokkien attribuutteja.

Seuraavaksi alkaa luokkakaavion hahmotteleminen.

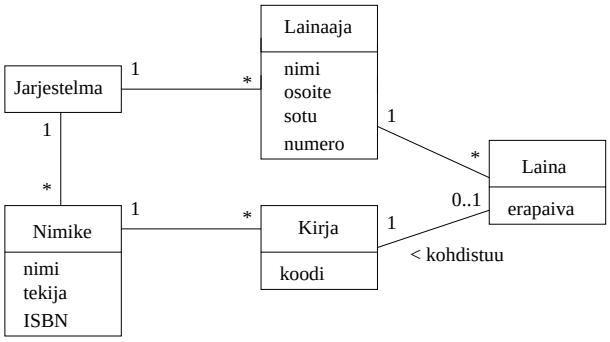
*Järjestelmä* pitää kirjaa sekä *lainaajista* että lainattavissa olevista *nimikkeistä*. Yksittäistä nimikettä kohti on useita *kirjoja*. Näiden huomioiden jälkeinen luokkamalli kuvassa 68. Kuvaan on merkitty myös muutamia attribuutteja. Yhteyksiä eikä yhteyksien rooleja ei ole vielä nimetty, sillä mallin ymmärtämisessä ei liene vaikeuksia ilman nimeämistäkään.




Kuva 68: Kohdealueen luokkakaavio, 1. versio

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.

Kuvassa 69 luokkamalli, joka sisältää kaikki ensimmäisen iteraation kannalta oleelliset käsitteet ja niiden väliset suhteet.




Kuva 69: Kohdealueen luokkakaavio, 2. versio

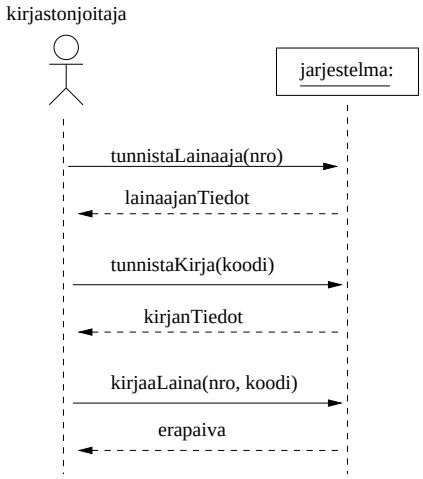
Kohdealueen luokkamalli on siis vielä täysin toteutusriippumaton malli, jonka tarkoitus on jäsentää sovellusalueen käsitteistöä ja käsitteiden suhteita. Tässä vaiheessa ei kannata vielä yrittääkään miettiä mitä operaatioita luokilla on. Luokat saavat operaatiot vasta oliosuunnitteluvaiheessa.

### 6.2.4 Järjestelmätason sekvenssikaaviot ja järjestelmän tarjoamat palvelut

Toimiakseen halutulla tavalla, on järjestelmän tarjottava ne palvelut, jotka käyttötapausten läpiviemiseen vaaditaan. Yksittäisen käyttötapauksen vaatiman toiminnallisuuden to-

teuttamiseksi järjestelmä joutuu yleensä toteuttamaan muutaman erilaisen yksittäisen palvelun. Joissain tilanteissa on hyödyllistä dokumentoida tarkasti, mitä yksittäisiä operaatioita käyttötapauksen toiminnallisuuden toteuttamiseksi järjestelmältä vaaditaan. Dokumentointiin sopivat hyvin luvussa 4.2 mainitut *järjestelmätason sekvenssikaaviot*, eli sekvenssikaaviot, joissa koko järjestelmä ajatellaan yhtenä oliona. Näin siis saadaan selkeästi esille ne yksittäiset toiminnot, joita järjestelmään kohdistetaan sen toiminnan aikana.

Kuvassa 70 käyttötapausta *Lainaa kirja* vastaava järjestelmätason sekvenssikaavio.




Kuva 70: Käyttötapauksen *Lainaa kirja* järjestelmätason sekvenssikaavio

Järjestelmätason sekvenssikaaviosta näemme selkeästi, mitä kommunikaatiota käyttäjän (eli kirjastonhoitajan) ja järjestelmän välillä on. Erityisesti näemme järjestelmän operaatiot eli palvelut, jotka järjestelmän on suoritettava käyttötapauksen aikana. Yksittäiset operaatiot on sekvenssikaaviossa nimetty, näin operaatioihin on helpompi viitata kuin käyttötapauksen yksittäisiin askeliin.

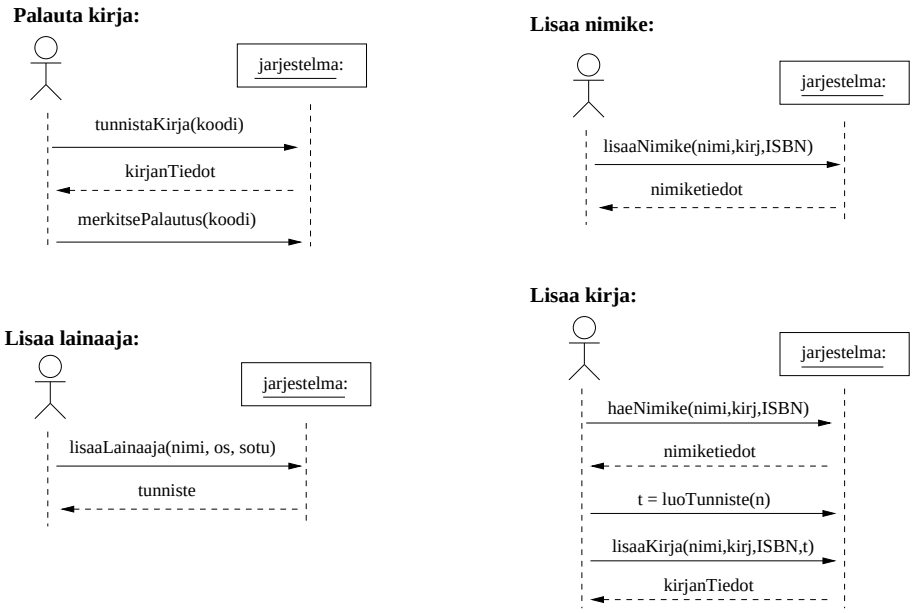
Kuvassa 71 muita käyttötapauksia vastaavat järjestelmätason sekvenssikaaviot.

Järjestelmätason sekvenssikaavioista nähdään, mitä yksittäisiä operaatioita järjestelmän on toteutettava eri käyttötapauksia suorittaessaan. Huomaamme, että sama operaatio, esim. *tunnistaKirja(koodi)* voi esiintyä monessa kaaviossa, eli samaa operaatiota saatetaan tarvita useampaa käyttötapausta suoritettaessa. Pystyäkseen siis suorittamaan kaikki vaaditut käyttötapaukset, on järjestelmän toteutettava kaikki järjestelmätason sekvenssikaavioissa ilmenevät operaatiot.

Listataan seuraavassa vielä järjestelmän operaatiot ja jokaisen 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.

**tunnistaLainaaja(nro)**

Lainaajan kirjastokortissa olevaa lainaajanumeroa (nro) vastaavat tiedot tulostetaan.




Kuva 71: Muiden käyttötapausten järjestelmätason sekvenssikaavio

**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 kohdealueen oliomalliin.

**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.

**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*.

Tässä vaiheessa alkaa olla suhteellisen selvää, mitä ensimmäisessä iteraatiossa toteutettavalta järjestelmän osalta odotetaan ja on aika siirtyä suunnitteluvaiheeseen.

Korostettakoon tässä vaiheessa sitä, että vaikka esim. vaatimusanalyysivaihe näyttää tässä monisteessa edenneen siististi vaihe vaiheelta, tapahtuu vaatimusten hahmottelu yhdenkin iteraation sisällä tosiasiassa paljon "epämääräisemmin", käyttäen kynää ja paperia, eri vaiheiden edetessä rinnakkain ja koko ajan tarkentuen. Eikä ole edes todennäköistä, että iteraation määrittelyvaihe tehdään kokonaisuudessaan ennen suunnittelua ja ohjelmointia.

Ohjelmiston kehityksessä edetään siis kokonaisuudessaan iteratiivisesti. Ensimmäiseen iteraation aikana toteutettavaksi valitaan osajoukko järjestelmältä haluttua toiminnallisuutta. Iteraation aikana valittu toiminnallisuus määritellään, suunnitellaan ja toteutetaan. Iteraatiossa tapahtuu ensin yleensä jonkun verran määrittelyä, sitten suunnittelua, jonka jälkeen ohjelmointia.

Ensimmäisen iteraation jälkeen valitaan taas lisää toiminnallisuutta, joka määritellään, suunnitellaan ja toteutetaan toisessa iteraatiossa. Tämän jälkeen seuraa tarpeellinen määrä iteraatioita, kunnes ohjelma alkaa olla valmis.

Jokaisen iteraation sisäiset vaiheet eli vaatimusmäärittely, suunnittelu, toteutus ja testaus etenevät joko peräkkäin tai osin rinnakkain noudatetusta prosessimallista riippuen.

## 6.3 Suunnittelu - iteraatio 1

Suunnitteluvaiheessa tarkoituksena on löytää sellaiset oliot, jotka pystyvät yhteistoiminnallaan toteuttamaan edellisessä luvussa listatut järjestelmältä vaadittavat operaatiot.

Suunnittelu jakautuu karkeasti ottaen kahteen vaiheeseen. 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. Suunnitteluvaiheessa tärkeäksi seikaksi nouseekin olioiden välinen yhteistyö, eli se vuorovaikutus, jolla oliot saavat aikaan halutun toiminnallisuuden.

Ennen kun lähdemme suunnitteluun, on syytä korostaa erästä seikkaa. Luvun 6.2.3 kohdealueen luokkamallissa esiintyvät luokat edustavat vasta sovellusalueen yleisiä käsitteitä, eivät suunnitteluvaiheen luokkia, jotka toteutusvaiheessa ohjelmoidaan. Vaatimusanalyysivaiheen luokille ei edes merkitä vielä mitään operaatioita.

Kuten pian tulemme näkemään, monet kohdealueen luokkamallin luokat tulevat siirtymään myös suunnittelu- ja toteutustasolle. Osa kohdealueen 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.

### 6.3.1 Arkitehtuurisuunnittelu

Ohjelmiston *arkkitehtuurilla* tarkoitetaan[12] karkeasti ottaen ohjelmiston korkean tason rakennetta, eli sen jakautumista erillisiin komponentteihin ja näiden rakennekomponenttien suhteita.

Komponentilla tässä tarkoitetaan yleensä kokoelmaa toisiinsa loogisesti liittyviä olioita, jotka suorittavat jotain tiettyä tehtävää ohjelmassa, esim. käyttöliittymän voitaisiin ajatella olevan yksi komponentti. Toisaalta ison komponentin voidaan ajatella koostuvan useista alikomponenteista, esim. sovelluslogiikkakomponetti voisi sisältää komponentin, joka huolehtii sovelluksen alustamisesta ja yhden komponentin kutakin järjestelmän toimintokokonaisuutta varten. Toisaalta komponenttijako voi perustua myös tietosisältöihin, esim. kirjastojärjestelmässä voisi olla omat komponentit lainaajia, lainoja ja kirjakokoelmaa varten.

Jos ajatellaan pelkkää ohjelman jakautumista komponenteiksi, puhutaan *loogisesta arkkitehtuurista*. Looginen arkkitehtuuri ei vielä ota kantaa siihen miten eri komponentit sijoitellaan, eli toimiiko esim. käyttöliittymä samassa koneessa kuin sovelluksen käyttämä tietokanta.

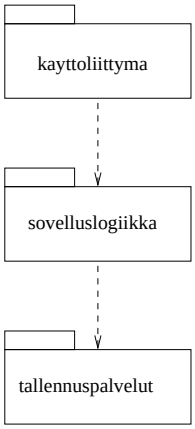
Sovelluksen loogisen arkkitehtuurin kuvaamiseen sopivat UML:n *pakkauskaaviot* (engl. package diagram). Kuvassa 72 kirjastojärjestelmän karkean tason arkkitehtuuria[13] vastaava UML:n pakkauskaavio.

Kirjastojärjestelmän rakenne noudattaa ns. *kerrosarkkitehtuuria* (engl. layered architecture), joka on yksi hyvin tunnettu *arkkitehtuurinen malli* (engl. architecture pattern), eli periaate, jonka mukaan tietynlaisia ohjelmia kannattaa pyrkiä rakentamaan. Kerros on kokelma 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 ja alimpana tallennuspalveluiden kerros, eli esim. tietokanta, jonne sovelluksen olioita voidaan tarvittaessa tallentaa.

Pakkauskaaviossa yksi komponentti kuvataan *pakkaussymbolilla*, eli laatikolla, jonka vasem-

[12] Arkkitehtuurin käsite on laaja ja hieman monisyisempi kun tässä esitetty. Arkkitehtuurin käsitettä tarkennetaan 2. vuoden kevään kurssilla Ohjelmistotuotanto sekä Ohjelmistojärjestelmien linjan syventävissä opinnoissa kurssilla Ohjelmistoarkkitehtuurit.

[13] Puhuttaessa arkkitehtuurista, tarkoitetaan tässä luvussa usein nimenomaan loogista arkkitehtuuria eli ei oteta kantaa komponenttien sijoittelusta fyysisille tietokoneille.




Kuva 72: Kirjastojärjestelmän looginen arkkitehtuuri

massa ylänurkassa on pieni laatikko. Pakkauksen nimi on joko kuvan 72 tapaan keskellä pakkaussymbolia tai ylänurkan laatikossa.

Pakkausten välillä olevat riippuvuudet ilmaistaan katkoviivalla, joka suuntautuu pakkaukseen, johon riippuvuus kohdistuu. Kerrosarkkitehtuurissa siis on pyrkimyksenä, että riippuvuuksia on ainoastaan alapuolella oleviin kerroksiin. Kirjastojärjestelmän käyttöliittymäkerros siis riippuu sovelluslogiikkakerroksesta. Riippuvuus tarkoittaa käytännössä sitä, että käyttöliittymän oliot kutsuvat sovelluslogiikan olioiden metodeja. Sovelluslogiikkakerros taas on riippuvainen tallennuspavelukerroksesta.

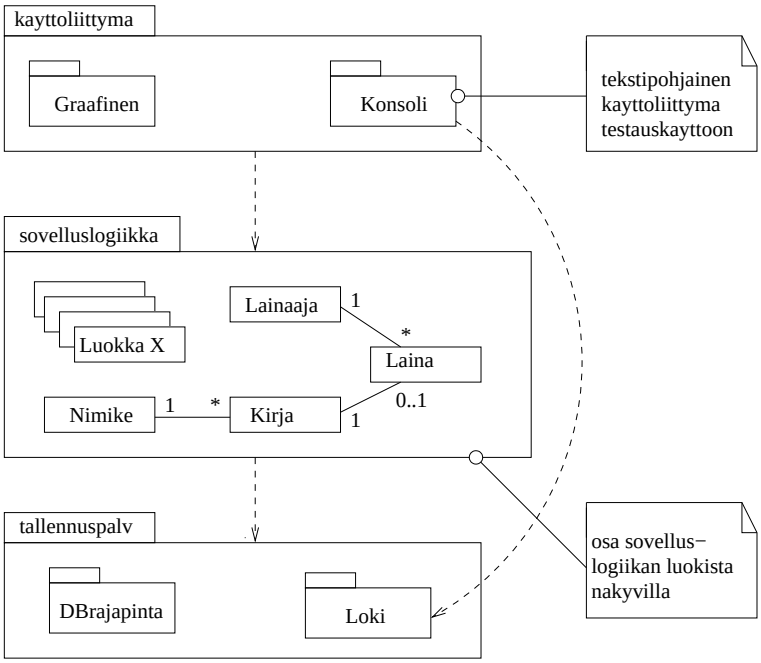
Pakkauksen sisältö on mahdollista piirtää pakkaussymbolin sisään, näin on tehty kuvassa 73, eli kirjastojärjestelmän arkkitehtuurin tarkemmassa versiossa. Pakkauksen sisällä voi olla muutakin kuin alipakkauksia, kuten esim. luokkia.

Sovelluslogiikkapakkauksen sisään on piirretty osa sovelluslogiikan luokista. Kuvassa ei siis kaikkia luokkia näytetä ja kuva tulee tarkentumaan myöhemmin. Käyttöliittymäpakkaus koostuu kahdesta alipakkauksesta, joista toinen on graafinen käyttöliittymä ja toinen testauskäyttöön tarkoitettu tekstipohjainen konsolikäyttöliittymä. Käyttöliittymäpakkaus on riippuvainen sovelluslogiikkapakkauksesta ja sovelluslogiikkapakkaus tallennuspalvelupakkauksesta. Kuvasta ilmenee myös, että konsolikäyttöliittymän toteuttava pakkaus käyttää suoraan tallennuspalvelupakkauksen loki-tiedostoon kirjoittamisen toteuttavaa alipakkausta. Kysessä on alipakkauksien keskinäinen riippuvuus, graafisen käyttöliittymän toteuttava pakkaus ei siis ole riippuvainen tallennuspalveluista.

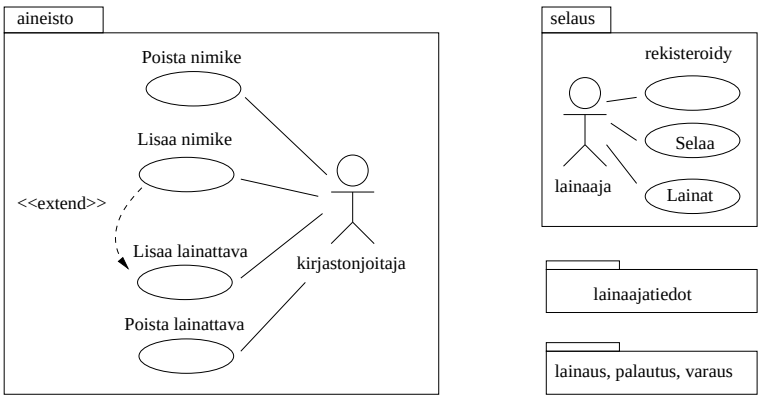
Pakkausten yhteydessä voidaan käyttää nimeämiskäytäntöä, jossa komponentin (esim. alipakkauksen) nimi on useampiosainen, esim. *Kayttoliittyma::Konsoli* viittaa pakkauksen Kayttoliittyma sisällä olevaan Konsoli-nimiseen komponenttiin.

UML:n pakkaussymbolilla voidaan ryhmitellä mitä tahansa UML-komponentteja, eli pakkauksia, luokkia, olioita, käyttötapauksia, .... Voitaisiin esim. jakaa kirjastojärjestelmän käyttötapaukset ylläpidon helpottamiseksi tai dokumentoinnin selkeyttämiseksi neljään eri pakkaukseen kuten kuvassa 74, jossa ainoastaan kahden pakkauksen sisältö näkyvillä.

Jos pakkauksessa on paljon sisältöä, voi sisällön näyttäminen piirtämällä sisältyvät kompo-

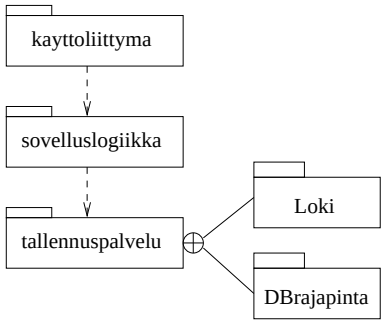



Kuva 73: Kirjastojärjestelmän looginen arkkitehtuuri tarkemmin




Kuva 74: Kirjastojärjestelmän käyttötapausten jakautuminen pakkauksiin

nentit pakkauksen sisäpuolelle olla joskus ongelmallista. Tällöin voidaan käyttää vaihtoehtoista tapaa, jossa pakkaukseen sisältyvät komponentit piirretään pakkauksen ulkopuolelle mutta liitetään ne sisältävään pakkaukseen käyttämällä kuvassa 75 esiteltyä merkintää, sisältyvyysrelaatiota. Kuvaassa 75 on esitetty kirjastojärjestelmän arkkitehtuuri yleistasolla. Tallennuspalvelupakkauksen sisältö näytetty käyttäen sisältyvyysrelaatiota.




Kuva 75: Vaihtoehtoinen tapa näyttää pakkauksen sisältö

### 6.3.2 Kerrosarkkitehtuurin etuja

Kerrosarkkitehtuurilla on monia etuja. Kerroksittaisuus helpottaa ylläpitoa, sillä jos tietyn kerroksen palvelurajapintaan (eli muille kerroksille näkyvään osaan) tehdään muutoksia, aiheuttavat muutokset ylläpitotoimenpiteitä ainoastaan ylemmän kerroksen riippuvuuksia omaavissa pakkauksessa. 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 lokitiedostoon kirjoitusta tai tietokantayhteyksiä voidaan uusiokäyttää mahdollisesti myös muissa sovelluksissa.

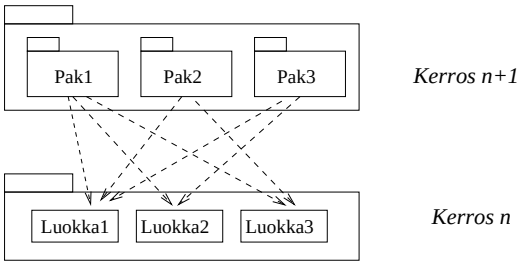
Myös kerrosten sisällä ohjelman loogisesti toisiinsa liittyvät komponentit kannattaa ryhmitellä omiksi pakkauksiksi[14]. Yksittäisistä pakkauksista kannattaa tehdä mahdollisimman yhtenäisiä toiminnallisuudeltaan, eli sellaisia, joiden osat kytkeytyvät tiiviisti toisiinsa ja palvelevat ainoastaan yhtä selkeästi eroteltua tehtäväkokonaisuutta. Samalla pyrkimyksenä on, että erilliset pakkaukset ovat mahdollisimman löyhästi kytkettyjä toisiinsa, eli pakkausten välisiä riippuvuuksia pyritään minimoimaan.

Ohjelman selkeä jakautuminen mahdollisimman riippumattomiin pakkauksiin eristää koodiin ja suunnitelmaan tehtävien muutosten vaikutukset mahdollisimman pienelle alueelle, eli ainoastaan riippuvuuden omaaviin pakkauksiin. Tämä helpottaa ohjelman ylläpitoa ja tekee sen laajentamisen helpommaksi. Selkeä jakautuminen pakkauksiin myös helpottaa työn jakamista suunnittelu- ja ohjelmointivaiheessa.

Pelkkä kerroksittaisuus ei tee ohjelman arkkitehtuurista automaattisesti hyvää. Kuvassa

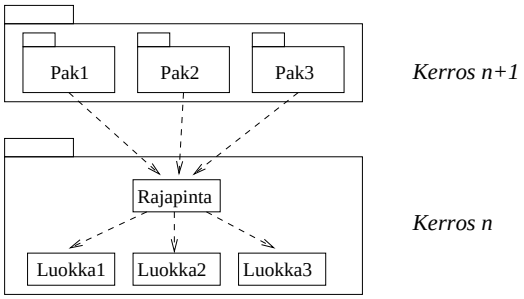
[14]Parempi termi kuin pakkaus tälläisestä mahdollisimman itsenäisestä osajärjestelmästä olisi esim. komponentti, käytetään kuitenkin nyt termiä pakkaus koska komponenttijako kuvataan pakkauskaavioiden avulla. Kurssilla ohjelmistotuotanto käsitellään UML:n komponenttikaavio, joka oikeastaan sopii pakkauskaaviota paremmin paremmin ohjelman arkkitehtuurin kuvaamiseen.

76 tilanne, missä kerroksen n+1 kolmella alipaketilla on kullakin paljon riippuvuuksia kerroksen n sisäisiin komponenttiin (tässä esimerkissä yksittäisiä luokkia). Esim. muutos kerroksen n luokkaan 1 aiheuttaa nyt muutoksen hyvin moneen ylemmän kerroksen pakettiin.




Kuva 76: Kerroksella ei selkeää rajapintaa

Mahdollinen ratkaisu ongelmaan on määritellä kerrosten välille selkeä *rajapinta* (engl. interface). Yksi tapa toteuttaa rajapinta on luoda kerroksen sisälle erillinen rajapintaolio, jonka kautta ulkoiset yhteydet tapahtuvat. Tätä periaatetta sanotaan *fasaadimalliksi* (engl. Facade pattern). Kuvassa 77 on luotu rajapintaolio kerrokselle n. Kaikki kommunikointi kerroksen kanssa tapahtuu rajapinnan kautta, eli ylemmän kerroksen riippuvuudet kohdistuvat ainoastaan rajapintaolioon. Nyt muutos esim. luokkaan 1 ei vaikuta kerroksen n+1 komponentteihin millään tavalla. Ainoat muutokset on tehtävä rajapintaolion sisäiseen toteutukseen.




Kuva 77: Kerroksella hyvin määritelty rajapinta

Kerrosarkkitehtuurin lisäksi on olemassa monia erilaisia arkkitehtuurimalleja eli hyväksi havaittuja tapoja jakaa järjestelmä kokonaisuuksiin, ks. esim [5].

### 6.3.3 Sovelluslogiikan ja Käyttöliittymän erottaminen

Ennen oliosuunnitteluun siirtymistä on syytä vielä nostaa esille yksi kerrosarkkitehtuuriin liittyvä tärkeä seikka. Kerrosarkkitehtuurin ylimpänä kerroksena on yleensä käyttöliittymä. Yleensä pidetään järkevänä, että ohjelman sovelluslogiikka on täysin erotettu käyttöliittymästä. 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 ja välittävät käyttäjän komennot eteenpäin sovelluslogiikalle.

Käytännössä erottelu tehdään liittämällä käyttöliittymän ja sovellusalueen olioiden väliin erillisiä komponentteja, jotka koordinoivat käyttäjän komentojen aiheuttamien toimenpiteiden suoritusta sovelluslogiikassa. Erottelun pohjana on Ivar Jacosonin [11] kehittämä idea oliotyyppien jaoittelusta kolmeen osaan, *rajapintaolioihin, ohjausolioihin* ja *sisältöolioihin*. Käyttöliittymän (eli rajapintaolioiden) ja sovelluslogiikan (eli sisältöolioiden) yhdistävät *ohjausoliot* (engl. control objects). Käyttöliittymä ei siis ole suoraan yhteydessä sovelluslogiikkaan luokkiin, vaan ainoastaan välittää käyttäjien komentoja ohjausolioille, jotka huolehtivat sovelluslogiikan olioiden hyödyntämisestä. Seuraavassa luvussa esiteltävä Larmanin [16] *ohjausperiaate* on hyvin lähellä Jacobsonin ideaa[15].

Sovellusalueen dataan tulevat muutokset tulee kyetä näyttämään myös käyttöliittymään. Jyrkässä kerrosarkkitehtuuriperiaatteessahan palvelupyyntöjen pitäisi aina kulkea ylhäältä alaspäin, eli muuttuneiden tietojen välitys käyttöliittymälle aiheuttaa hankaluuksia kerroksellisuuden suhteen. Sovellusalueen tietojen muutosten välittäminen käyttöliittymälle hoidetaan oikeaoppisesti käyttäen ns. *tarkkailija*-periaatetta (engl. observer pattern)[16] Käsittelemme asiaa kuitenkin vasta kurssilla Ohjelmistotuotanto.

Sovelluslogiikan erottaminen käyttöliittymästä mahdollistaa myös sen, että samasta sovellusalueen datasta voi olla olemassa useita erilaisia näkymiä yhtäaikaa. Eli sama data pystytään näyttämään eri tavoin käyttäjän tarpeista riippuen, esim. joko tekstuaalisessa muodossa tai graafisena diagrammina.

### 6.3.4 Oliosuunnittelu

Aloittelevalle ohjelmoijalle ohjelmiston elinkaaren vaiheista yksi haastavimpia lienee oliosuunnittelu. Ainakin osa ohjelman vaatimuksista on helppo kartoittaa, mutta miten löydetään, keksitään ja suunnitellaan sopivat oliot, jotka toteuttavat ohjelman vaatimukset järkevällä tavalla?

[15]Sovelluslogiikan ja käyttöliittymien erottelun yhteydessä puhutaan usein MVC-mallista [5]. MVC:tä tunteville tarkennettakoon, että Jacobsonin ja Larmanin ohjausoliot eivät ole tarkalleen ottaen aivan sama asia kuin MVC-mallin kontrolleri, ainakaan siinä mielessä kun MVC:tä ajatellaan esim. Javalla tapahtuvan käyttöliittymäohjelmoinnin yhteydessä [1]. Javan kontrolleri on Jacobsonin ajattelussa rajapintaolio. Koska MVC-mallin mielekäs käsittely vaatii käyttöliittymäohjelmoinnin tuntemista, tutustutaan periaatteeseen vasta kurssilla Ohjelmistotuotanto.

[16]Tarkkailijaperiaateessa käyttöliittymän komponentit rekisteröivät itsensä sovelluslogiikalle odottamaan päivityskäskyjä. Käyttöliittymän komponentit eivät kuitenkaan rekisteröi itseään suoraan vaan ainoastaan ns. tarkkailurajapinnan kautta. Sovelluslogiikka tunteekin ainoastaan joukon sitä tarkkailevia rajapintoja, konkreettisia käyttöliittymän komponentteja jotka toteuttavat tarkkailurajapinnat ei sovelluslogiikka tunne [10].

Ohjelman on siis toteutettava käyttötapauksista johdetut operaatiot (esimerkkimme operaatiot sivulla 72) toimiakseen vaatimustensa mukaisesti. Ohjelmalta vaadittavien operaatioiden voidaan ajatella olevan ohjelman *vastuita* (engl. responsibilities). Eli hoitamalla vastuunsa ohjelma toimii, niinkuin sen odotetaan toimivan.

Ohjelma toteuttaa vastuunsa olioiden yhteistyön avulla. Haasteena suunnittelussa 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 hahmotella periaatteita, jotka auttavat ohjelman vastuut toteuttavien olioiden löytämisessä. Periaatteet noudattavat melko pitkälti Larmanin kirjan [16] esitystä. Menetelmästä käytetään yleisesti nimitystä *vastuupohjainen oliosuunnittelu* (engl. responsibility driven object design). Tämän kurssin puitteissa ei ole mahdollisuutta asian kovin syvälliseen käsittelyyn. Toisaalta oliosuunnittelussa tarvitaan aina luovuutta. Mikään yksittäinen menetelmä ei toimi kaikenlaisiin ongelmiin vaan antaa ainoastaan virikkeitä alkuun pääsemiseen. Larmanin kirjan lisäksi vastuupohjaista oliosuunnittelua käsitellään esim. Wirfs-Brockin ja kumppaneiden kirjoissa [23, 22].

Suunnittelumenetelmässä on muutamia periaatteita, joita pyritään pitämään mielessä suunnittelun edetessä. Periaatteet[17] on listattu seuraavassa. Niihin palataan tarkemmin kehitellessämme esimerkkisovellustamme.

Periaatteet ovat seuraavat[18]:

- Luvussa 6.3.3 jo mainittiin, että hyvien tapojen mukaista on erottaa sovelluslogiikka käyttöliittymästä. Ensimmäinen periaate liittyykin siihen, *kuka ottaa vastaan käyttöliittymäolioilta tulevat komennot*, eli miten esim. tietyn käyttöliittymän painikkeen klikkaamista vastaava tieto välitetään sovelluslogiikalle. Vastauksen tuo **ohjausperiaate**, jonka sisältö tarkentuu pian.
- Vastuita on pääpiirteittäin kahdenlaisia, *tietämistä* (engl. knowing) ja *tekemistä* (engl. doing) koskevia. Kirjastojärjestelmän vastuista tietämistä edustaa esim. vastuu *tunnistaLainaaja(nro)* eli tunnistetaan lainaaja kirjastokortin numeron perusteella. Tekemistä taas 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.[19] Usein mikään olio ei yksistään osaa hoitaa koko vastuuta. Yksittäinen vastuu pitääkin jakaa osavastuihin, jotka kokonaisvastuun omaava olio *delegoi* osavastuiden eksperteille.
- Kaikki oliot täytyy luonnollisesti luoda. **Luontiperiaatteen** mukaan olion luo se, joka

  – sisältää tai säilyttää olion

[17] Larmanin [16] on listattu yhteensä 9 periaaatetta, joista käytetään nimitystä GRASP patterns.

[18] Periaatteiden englanninkieliset nimet ovat melko hyvät, suomenkieliset nimet taas ovat omakeksimiäni ja osin melko kömpelöitä.

[19] Tämä periaate kuulostaa suorastaan typerän itsestäänselvältä.

  - pitää kirjaa oliosta tai
  - tuntee olion alustustuksessa tarvittavan datan

  Oliolla saattaakin näiden periaatteiden nojalla olla useita luojakandidaatteja. Eri ratkaisuista pitää valita tarkoituksenmukaisin vertailemalla ratkaisukandidaatteja esim. kahden seuraavaksi esitettävän periaatteen kriteerein.
- Kaikissa suunnitteluratkaisuissa tulee pääsääntöisesti pyrkiä **olioiden väliseen riippuvuuden vähäisyyteen** (engl. low coupling). Järjestelmän erilaisten komponenttien välisten riippuvuuksien minimointia perusteltiin jo kerrosarkkitehtuurin yhteydessä luvussa 6.3.2.
- Pelkkä riippuvuuksien minimointi ei tuota kaikin osin hyvää ratkaisua. Oliosuunnittelussa tulee pyrkiä myös **toiminnallisuudeltaan yhtenäisiin** (engl. high cohesion) olioihin, eli olihin, joiden vastuut (eli käytännössä julkiset metodit) liittyvät mahdollisimman saman asian tekemiseen. 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 sekä lainaajiin että lainoihin liittyviä vastuita.

  Kaikkia suunnitteluratkaisuja tulee siis punnita olioiden välisten riippuvuuksien ja olioiden toiminnallisen yhtenäisyyden kannalta ja pyrkiä näiden periaatteiden kannalta mahdollisimman hyviin ratkaisuihin.
- Osa suunnittelutason luokista saadaan vaatimusmäärittelyn aikana tehdyn kohdealueen luokkamallin pohjalta. On siis oletettavaa, että kirjastojärjestelmään otetaan suunnittelutasolle mukaan ainakin luokat lainaaja, laina, kirja ja nide.

  Jos suunnittelutasolle ei oteta muita luokkia kuin kohdealueen luokkamallista omaksutut luokat, joudutaan olioiden vastuita suunnitellessa helposti huonoihin ratkaisuihin, esim. toiminnallisesti epäyhtenäisiin olioihin sekä liiallisiin riippuvaisuuksiin.

  Tälläisissä tilanteissa **keksitään uusia, "keinotekoisia"oliota**, joita vastaavia käsitteitä ei välttämättä sovellusalueella ole.

Järjestelmän käynnistyessä on suoritettava erinäisiä alustustoimenpiteitä, jotka esim. lukevat alustustiedostoja ja tietokantaa ja luovat ohjelman suoritusaikana tarvitsemat oliot. Koska suunnittelun alkuvaiheessa ei ole vielä selvyyttä ohjelman tulevasta oliorakenteesta, kannattaa alustustoimenpiteet suunnitella vasta iteraation suunnitteluvaiheen lopussa, kun ohjelman oliorakenne alkaa olla selvillä.

Yksi tapa tehdä suunnittelua on edetä käyttötapauksittain. Eli otetaan yksi käyttötapaus kerrallaan tarkasteluun ja 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 olemassaolevaan suunnitelmaan muutoksia.

### Käyttötapaus Lainaa kirja

Otetaan ensin tarkasteluun käyttötapaus, joka kuvaa kirjan lainaustapahtuman. Tapahtumien kulku käyttötapauksessa siis on seuraava (sivulta 68):

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

Käyttötapauksen toteuttavat operaatiot tarkemmin eriteltynä (sivulta 72) ovat:

**tunnistaLainaaja(nro)**

Lainaajan kirjastokortissa olevaa lainaajanumeroa (nro) vastaavat 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*.

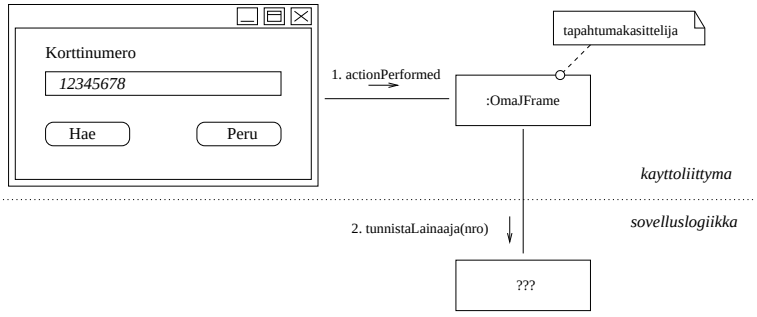
Käyttötapauksen toteuttamiseksi on siis toteutettava kolme operaatiota, joiden kunkin suorittaminen annetaan jonkin sovelluksen olion vastuulle.

Kuten aiemmin mainittiin, omaksutaan osa ohjelmiston suunnitteluvaiheen olioista kohdealueen luokkamallissa kuvatuista käsitteistä, joka on sivulla 71 kuvassa 69. Näyttää ilmeiseltä, että suunnitteluvaiheen luokiksi otetaan mukaan ainakin *Lainaaja*, *Laina* ja *Kirja*. Koska Kirja-oliot ovat siis tiettyyn nimikkeeseen liittyviä, on mukaan otettava mitä ilmeisimmin myös luokka *Nimike*. Eli alustavasti analyysivaiheessa hahmoteltu kohdealueen luokkamalli otetaan mukaan suunnitteluvaiheeseen lähes sellaisenaan.

Luvussa 6.3.3 mainitun periaatteen mukaisesti sovelluslogiikka täytyy erottaa käyttöliittymästä. Hyvä periaate on se, että käyttötapaukset toteuttavat operaatiot ovat sovelluslogiikkakerroksen olioiden vastuulla. Käyttöliittymä ainoastaan reagoi käyttäjien komentoihin, ja kutsuu sitten sopivaa sovelluslogiikkakerroksessa toteutettua operaatiota. Periaate ilmenee kuvan 78 vapaamuotoisesti piirretystä kommunikaatiokaaviosta. Kuvassa käyttötapauksen *Lainaa kirja* alussa tapahtuva lainaajan tunnistus, eli virkailija syöttää käyttöliittymäkomponentilla toteutettuun dialogi-ikkunaan kirjastokorttinumeron, jotta lainaaja voidaan tunnistaa. Javan Swing -käyttöliittymäkomponenteilla toteutettaessa painikkeen klikkaaminen tuottaa herätteen (action performed) ohjelmoijan toteuttamalle tapahtumakäsittelijäoliolle (joka tyypillisesti luokan JFrame-aliluokan ilmentymä). Tapahtumakäsittelijä kutsuu sitten jotain sovelluslogiikan olioa, joka hoitaa operaation.[20]

Minkä olion tulisi ottaa vastuulleen käyttöliittymän kutsuman operaation suoritus, eli mikä on kuvan 78 kysymysmerkein nimetty luokka? Tähän vastauksen tarjoaa aiemmin mainittu

[20]Tämä Javaan liittyvä selvennyksenä asiaa tunteville. Ohjelmoinnin jatkokurssin lopussa käsitellään hieman käyttöliittymäohjelmointia. Tälle kurssille asia ei kuulu.




Kuva 78: Käyttöliittymäkerros delegoi tehtävän sovelluslogiikalle.

**ohjausperiaate**, jonka yleisperiaatteena on selitetty luvussa 6.3.3. Periaatteena on asettaa käyttöliittymän ja varsinaisten sovellusolioiden väliin *ohjausolio*, joka ottaa vastaan käyttöliittymästä tulevan operaatiokutsun ja kutsuu edelleen sovelluslogiikan olioita, jotka suorittavat varsinaisen tehtävän.

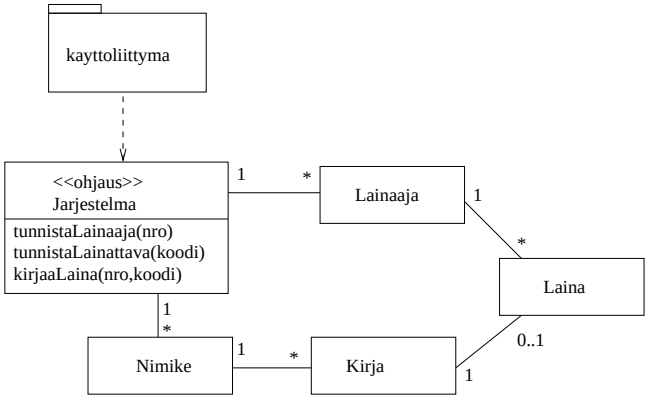
Ohjausolio voi olla joko ohjelman kaikkien operaatioiden yhteinen ohjausolio tai vaihtoehtoisesti jokaisella käyttötapauksella voi olla oma ohjausolionsa. Välimuodotkin ovat mahdollisia, eli joillain käyttötapauksella voi olla oma ohjausolio ja jotkut taas käyttävät yhteistä ohjausolioa.[21] Yksinkertaisissa järjestelmissä selvitään ehkä yhdellä ohjausoliolla. Jos järjestelmässä on paljon toiminnallisuutta, kannattanee kullakin käyttötapauksella olla oma ohjausolionsa. Näin ohjausoliot säilyvät toiminnallisesti yhtenäisinä ja helpommin ylläpidettävinä. Käyttötapauspohjaisten ohjausolioiden etuna on myöskin se, että niiden avulla voidaan helposti hallita monimutkaisen käyttötapauksen etenemistä helpommin kuin koko järjestelmälle yhteisen ohjausolion avulla.

Yksinkertaisessa ohjelmassa yhteisenä ohjausoliona voi joissain tapauksissa myös toimia sovellusalueen olio, joka vastaa koko järjestelmää. Päädymme ainakin alustavasti kaikille käyttötapauksille yhteisen ohjausolion käyttöön ja ohjausolioksi valitsemme koko järjestelmää vastaavan olion, jonka nimi on *Jarjestelma*. Kuvan 79 luokkakaavio dokumentoi tilanteen. Selvyyden vuoksi ohjausluokan rooli on merkitty kuvaan stereotyypin *«ohjaus»* avulla. Käyttöliittymä on näytetty ainoastaan pakkauksena, joka on riippuvainen ohjausluokasta. Kuvassa on myös mukana myös muut sovellusalueen luokkamallista omaksutut alustavat luokkaehdokkaat. Ensimmäisessä iteraatiossa ei vielä oteta kantaa olioiden tietokantaan tallettamiseen, eli tallennuspalvelupakkausta ei ole kuvaan merkitty.

Aloitetaan jakamaan operaatioiden suoritusvastuita luokille samalla tarkaentaen tarpeen mukaan luokkarakennetta.

Koska Jarjestelma-olio tuntee Lainaaja-oliot, voisi se **ekspertti-periaatteen** mukaan ottaa vastuulleen operaation *tunnistaLainaaja(nro)* suorittamisen. Periaatteen mukaanhan vastuu tulee antaa oliolle, jolla on parhaat tiedot operaation suorittamiseksi. Kukin Lainaaja-olio tuntee ainoastaan yhden lainaajan eli itsensä, joten se ei tule kyseeseen, mutta Jarjestelma-olio tuntee kaikki lainaajat, joten se on siinä mielessä sovelias operaation suorittajaksi.

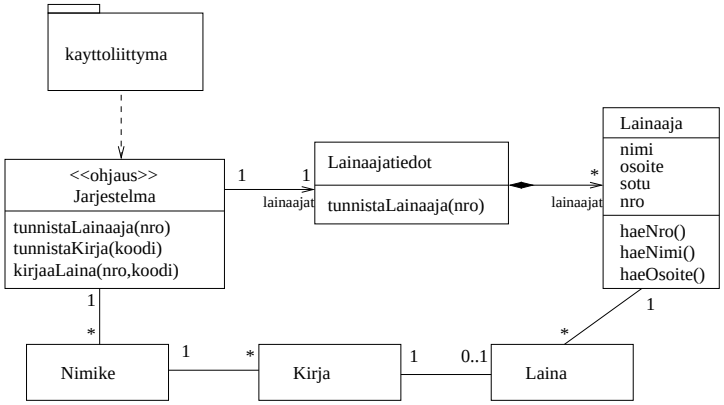
[21] Jacobsonin [11] menetelmässä periaatteena on, että jokaisella käyttötapauksella on oma ohjausolionsa.




Kuva 79: Koko järjestelmää kuvaava olio toimii ohjausoliona ja ottaa vastaan käyttöliittymältä tulevat operaatiokutsut.

Tarkastellessa muitakin operaatioita (esim. lisaaLainaaja, lisaaNimike, ...), huomataan, että Jarjestelma-oliosta uhkaa muodostua yleisolio, joka suorittaa suuren joukon operaatioita, jotka eivät toiminnallisesti liity toisiinsa. Näin ollen rikottaisiin *toiminnallisen yhtenäisyyden* periaatetta. Ohjelman luokkarakennetta kannattaakin jalostaa siten, että saadaan aikaan pienempiä, enemmän yhdelle asialle omistautuneita luokkia.

Lisätään luokka Lainaajatiedot, jonka tehtävänä on huolehtia yksittäisistä lainaajista, eli Lainaaja-oliot siirretään lainaajatietojen alle, ks. kuva 80. Nyt siis yhteys on muutettu kompositioksi, sillä yksittäiset lainaajat ovat alisteisia koko järjestelmän lainaajajoukolle, josta Lainaajatiedot pitää kirjaa. Kuvaan on merkitty myös kahden yhteyden roolinimet sekä kahden yhteyden navigointisuunta. Yhteyksien suunnista huomaamme, että esim. Jarjestelma-oliosta on mahdollista navigoida Lainaajatiedot-olioon, mutta ei päinvastoin.




Kuva 80: Suunnitelmaan lisätty uusi luokka, jonka vastuulla Lainaajatiedot.

Itse operaation *tunnistaLainaaja(nro)* sisällön voimme kuvata esim. pseudokoodina. Oletetaan, että operaatio palauttaa kutsujalle (eli käyttöliittymälle) tunnistetun lainaajan tiedot merkkijonomuodossa.

Operaatio alkaa sillä, että käyttöliittymä kutsuu Jarjestelma-olion, joka siis toimii ohjausoliona, operaatiota *tunnistaLainaaja(nro)*. Uudessa luokkarakenteessa Jarjestelma ei enää

tunne lainaajia, mutta se tuntee Lainaajatiedot-olion, joka taas tuntee lainaajat. Eli 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. Jos etsittyä lainaajaa ei löytynyt, palautetaan merkkijono *tuntematon*. Jarjestelma-olion suorittama operaatio pseudokoodina seuraavassa:

```
class Jarjestelma{
  // muut metodit

  String tunnistaLainaaja(int nro){
     Lainaaja vast = lainaajat.tunnistaLainaaja(nro)

     jos vast null palauta "tuntematon"

     String nimi = vast.haeNimi()
     String os = vast.haeOsoite()

     palauta nimi+os;
  }
}
```

Lainaajatiedot-oliolle on siis delegoitu vastuu annettua numeroa vastaavan Lainaaja-olion löytämisestä.

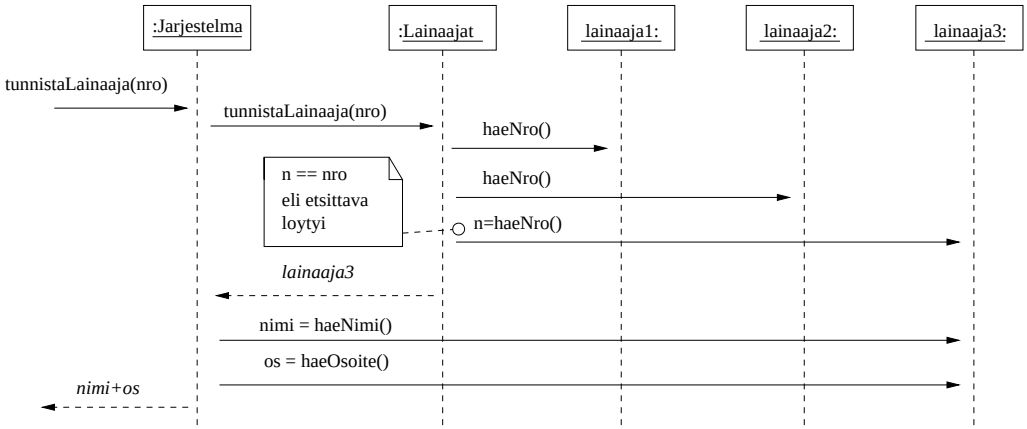
Lainaajatiedot-olio tarkastaa jokaiselta sisältämältään Lainaaja-olioilta, onko se etsitty. Alla olevassa pseudokoodissa kysymyksen toisto on kirjoitettu muodossa *tee kaikille l joukossa lainaajat*, eli *lainaajat* viittaa Lainaaja-olioihin, jotka Lainaajatiedot tuntee ja *l* on yksi kerrallaan kukin lainaajaolioista.[22]. Jos jokin Lainaaja-olioista on etsitty, palautetaan kutsujalle viite löydettyyn Lainaaja-olioon. Jos lainaajaa ei tunnisteta, palautetaan null-viite, eli viite ei mihinkään.

```
class Lainaajatiedot{
  // muut metodit

  Lainaaja tunnistaLainaaja(int nro){
     tee kaikille l joukossa lainaajat {
        int n = l.haeNro()

        // löytyikö etsitty henkilö
        jos n == nro  palauta  l
   }
  // ei löytynyt lainaajaa, jolla annettu kirjastokortin numero
  palauta null
```

[22] Javassa tämä voitaisiin toteuttaa esim. for- tai while-lauseella jos yhteys olisi toteutettu taulukkona tai listarakenteena. Luvussa 6.4 esitellään yksi tapa yhteydessä olevien olioiden läpikäyntiin. Etsimisnopeuden kannalta järkevin tapa toteuttaa yhteys olisi ns. Map-rajapinnan toteuttavan säiliöolion käyttö. Asia ei kuitenkaan kuulu tämän kurssin aihepiiriin.

```
}
```

Operaation suoritus on esitetty sekvenssikaaviona kuvassa 81. Kuvatussa skenaariossa kolmas läpikäytävä Lainaaja-olio on etsitty. Vertaa pseudokoodia ja sekvenssikaaviota ja varmista, että ymmärrät miten operaation toiminta etenee olioiden välisenä yhteistyönä.



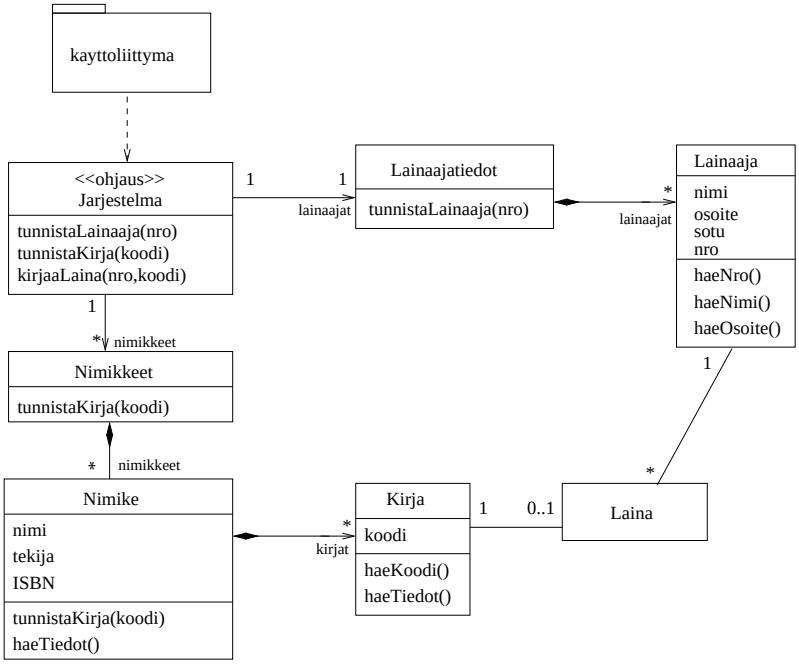

Kuva 81: Operaatio tunnistaLainaaja sekvenssikaaviona

Seuraava tarkasteltava operaatio on *tunnistaKirja(koodi)*, jonka tehtävä on etsiä tiettyä kirjaa 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 tehtävä on vastaava kuin Lainaajatiedot-luokan oliolla, eli pitää kirjaa ja suorittaa operaatioita yksittäisille Nimike-olioille. Päivitetty luokkamalli kuvassa 82. Yhteyksiin on jälleen liitetty roolinimiä ja suuntia.

Operaation *tunnistaKirja(koodi)* toteutus tapahtuu monen olion yhteistyönä, joten se saattaa vaikuttaa aluksi sekavalta. Valotetaan operaation toimintaa jälleen sekä pseudokoodina että sekvenssikaaviona. Ensin pseudokoodi. Koodin seuraamisen rinnalla kannattaa ehkä katsoa jo kuvassa 83 olevaa sekvenssikaaviota. Oletamme koodissa, että operaatio palauttaa kutsujalle (eli käyttöliittymälle) merkkijonoesityksen tunnistetusta kirjasta.

Operaatio alkaa sillä, että käyttöliittymämodulista kutsutaan 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. Jos etsittyä Kirja-




Kuva 82: Suunnitelmaan lisätty uusi luokka, jonka vastuulla Nimike-oliot.

olioa ei löytynyt, palautetaan merkkijono *tuntematon*.

```
class Jarjestelma{
  // muut metodit

  String tunnistaKirja(int koodi){
     Kirja k = nimikkeet.tunnistaKirja(koodi)

     jos  k == null  palauta "tuntematon"

     String vast = k.haeTiedot()

     palauta vast
  }
}
```

Suorittaakseen vastuunsa kirjan etsimiseksi, Nimikkeet-olio kysyy jokaiselta sisältämältään yksittäiseltä Nimike-oliolta, tunteeko se etsityn kirjan. Komento on kirjoitettu pseudokoodiin muodossa *tee kaikille n joukossa nimikkeet*, eli *nimikkeet* viittaa Nimike-oliojoukkoon ja $n$ on yksi kerrallaan kukin Nimike-olioista. Jos jokin Nimike-olioista tunnistaa kirjan (eli kirjalla on etsitty koodi), palautetaan viite löytyneeseen Kirja-olioon kutsujalle. Jos taas kirjaa ei tunnisteta, palautetaan null-viite. Nimikkeet-olio siis suorittaa vastuunsa pilkkomalla sen yksittäisten Nimike-olioiden osavastuiksi.

```
class Nimikkeet{
   // muut metodit

   Kirja tunnistaKirja(int koodi){
      tee kaikille n joukossa nimikkeet {
         Kirja vast = n.tunnistaKirja()

         jos vast ei ole null palauta vast
      }
      // jos ei löytynyt koodia vastaavaa kirjaa
      palauta null
   }
}
```

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.

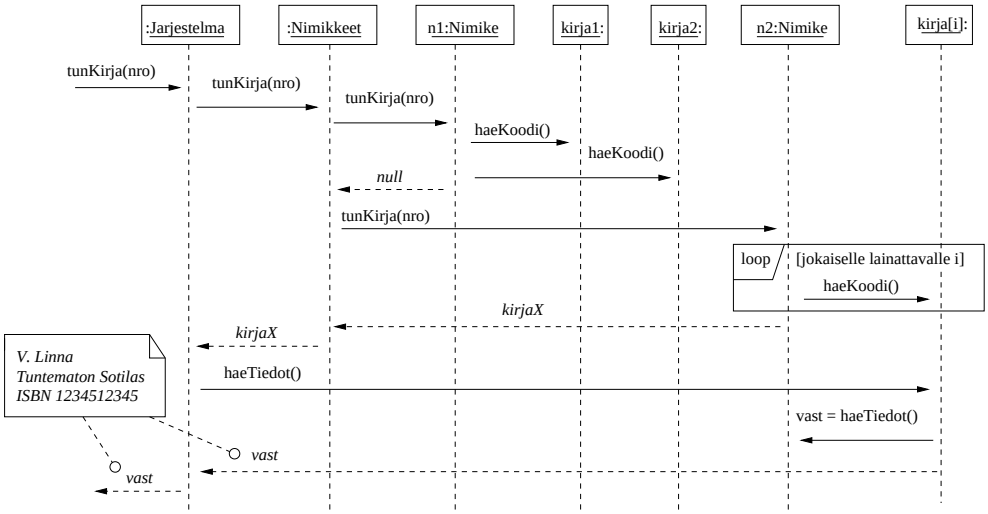
```
class Nimike{
   // muut metodit

   Kirja tunnistaKirja(int koodi){
      tee kaikille k joukossa kirjat {
         int vast = k.haeKoodi()

         jos vast == koodi palauta  k
      }
      // jos nimikkeen alta ei löytynyt koodia vastaavaa kirjaa
      palauta null
   }
}
```

Kuvassa 83 tilannetta vastaava sekvenssikaavio. Kuten kuvasta huomaamme, on UML:n merkintätapoja käytetty osin luovasti, mm. pitkä operaationimi *tunnistaKirja(koodi)* on lyhennetty ja vastauksen tarkka sisältö esitetään kommentin avulla.

Kuvatussa skenaariossa tutkitaan ensin Nimike-olion *n1* sisältämät Kirja-oliot *kirja1* ja *kirja2*, joista kumpikaan ei ole etsitty kirja. Nimike-olion *n2* alta löytyy etsitty Kirja-olio. Huomaa, miten usealta erilliseltä $n2$:n alta olevalta kirjalta (kuvassa *kirja[i]*) tapahtuva kysely on kuvattu toistorakenteella. Toiston kaikki viestit siis kohdistuvat erillisiin olioihin. Kuvassa $i$ on ikäänkuin vaihtuva muuttuja, jolla eritellään yksittäiset Kirja-oliot. Etsitty löytyy $n2$:n alta ja viite siihen (kuvassa $kirjaX$) palautetaan aina Jarjestelma-oliolle asti.

Jarjestelma-olion kuuluu palauttaa etsityn kirjan tiedot merkkijonona. Saatuaan viitteen etsittyyn Kirja-olioon, Jarjestelma-olio kysyy sen tietoja, jotka se palauttaa kutsujalleen.

Sekvenssikaavioon liittyy vielä yksi tärkeä seikka. Kirja-olio ei tiedä itse muuta kuin koodinsa, muut tietonsa (nimi, tekija, ISBN) se kysyy omalta Nimike-olioltaan. Tämän takia Yhteys Nimike- ja Kirja-olion välillä on kaksisuuntainen vaikka sitä ei UML:ssä voi kompositioihin merkitäkään.




Kuva 83: Operaation tunnistaKirja-suoritusta hahmottava sekvenssikaavio.

Enemmän ohjelmointia harrastanut huomaa helposti, että ratkaisu on tehoton, sillä tietyn kirjan löytymiseksi on pahimmassa tapauksessa käytävä läpi kaikki nimikkeet ja niiden alla olevat kirjat. Voikin olla, että toteutusvaiheessa ohjelmoija päättää toteuttaa operaation tavalla, joka muuttaa hieman myös luokkarakennetta. Suunnittelun aikana tehdyt mallit eivät ole koskaan kiveenhakattuja. Mallien pääasiallisena tarkoituksena onkin jäsentää ajattelua ja mahdollistaa erilaisten suunnitteluratkaisujen vertailu. Malli on ajanut asiansa myös siinä tapauksessa, että se todetaan huonoksi ja päätetään hylätä.

Käyttötapauksen *Lainaa kirja* viimeinen suunniteltava operaatio on *kirjaaLaina(nro,koodi)*. Operaation on siis luotava Laina-olio, josta luodaan yhteys lainaajaan, jonka kirjastokortin numero *nro* ja kirjaan jolla tunnisteena *koodi*.

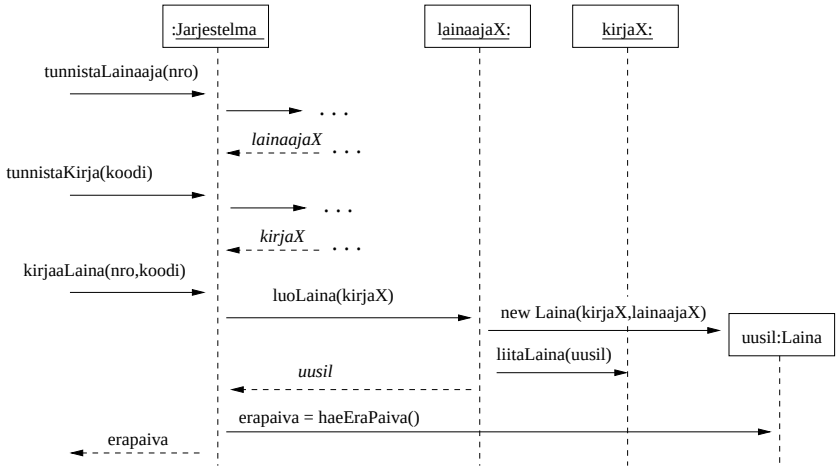
Sivulla 81 maninitun **luontiperiaatteen** mukaan olion luo se, joka

- sisältää tai säilyttää olion
- pitää kirjaa oliosta
- tuntee olion alustustuksessa tarvittavan datan

Periaatetta soveltaen Laina-olion voisi luoda Lainaaja (säilyttää lainaa) tai Kirja (pitää kirjaa lainasta). Kolmas vaihtoehtoinen luojakandidaatti on 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.

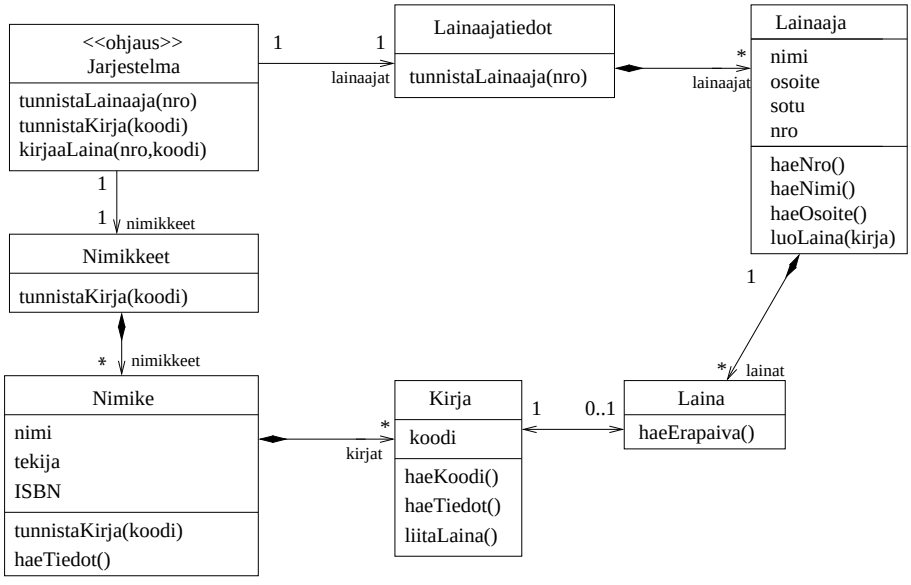
Operaation toteutus kuvassa 84. Ennen uuden lainan kirjaamista on suoritettu operaatiot *tunnistaLainaaja* ja *tunnistaKirja*, joiden perusteella tiedetään, mihin Kirja- ja Lainaaja-olioon uusi Laina-olio liitetään. Sekvenssikaaviossa näistä olioista on käytetty merkintää $kirjaX$ ja $lainaajaX$. Jarjestelma-olio 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.




Kuva 84: Operaation lisaaLaina-suoritusta hahmottava sekvenssikaavio.

Suunniteltuamme käyttötapauksen *Lainaa kirja*, päädymme kuvan 85 luokkakaavioon. Tilan säästämiseksi käyttöliittymäkerrosta kuvaava pakkaus on jätetty kuvasta pois.




Kuva 85: Luokkakaavio Lainaa kirja -käyttötapauksen suunnittelun jälkeen

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 voitaisiin tuoda luokkakaaviossa esiin piirtämällä Jarjestelma-oliosta yhteydet luokkiin Kirja ja Koodi. Nämä yhteydet siis olisivat merkki siitä, että Jarjestelma-olio muistaa tietyllä hetkellä viimeksi käsittelyssä ol-

leet Kirja- ja Lainaaja-oliot. Koska kysessä ovat vain tilapäiset, yhden käyttötapauksen suorituksen aikana olemassa olevat yhteydet, ei niitä ole tapana merkitä luokkakaavioon yhteyksinä. Yhteyksinä on ainoastaan tapana merkitä pysyvämpiluontoisia olioiden välisiä yhteyksiä. Jos asia halutaan tuoda luokkakaaviotasolla esille, on se parempi merkitä *riippuvuutena* Jarjestelma-luokasta luokkiin Kirja ja Lainaaja ja tuoda riippuvuuden luonne esiin esim. kommenttina tai stereotyyppinä. Riippuvuudet on merkitty ensimmäisen iteraation lopulliseen suunnittelutason luokkamalliin eli sivulla 96 olevaan kuvaan 90.

### Käyttötapaus Palauta kirja

Otetaan seuraavaksi tarkasteluun käyttötapaus, joka kuvaa kirjan palautustapahtuman. Tapahtumien kulku käyttötapauksessa siis on seuraava (sivulta 68):

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

Käyttötapauksen toteuttavat operaatiot (sivulta 72) ovat:

**tunnistaKirja(koodi)**

Tunnistetaan yksittäinen kirja ja tulostetaan sen tiedot.

**merkitsePalautus(koodi)**

Kirjaa, jonka tunniste *koodi*, vastaava Laina-olio tuhotaan.

Myös tämän käyttötapauksen ohjausoliona toimii Jarjestelma-olio. Huomaamme, että ensimmäinen operaatioista on jo suunniteltu edellisen käyttötapauksen yhteydessä. Riittää siis että annamme toisen operaation vastuun jollekin suunnitelman olioista.

Ekspertti-periaatteen nojalla sopivimmat oliot lainan tuhoamisvastuun kannalta ovat Lainaaja ja Kirja, sillä molemmat tuntevat tuhottavan Laina-olion. Kuvan 83 sekvenssikaavion perusteella on selvää, että ensimmäisen operaation suorituksen jälkeen ohjausoliolla on viite palautettavana olevaa kirjaa vastaavaan Kirja-olioon. Kirja-olio löytyy siis ilman erillistä vaivaa, joten vastuu lainan tuhomaisesta on viisainta delegoida sille. Kirja-olio ei tunne kirjan lainaajaa, joten se delegoi Laina-oliolle vastuun lainan poistumisen tiedottamisesta lainaajalle. Lainaaja-olion on syytä tietää Laina-olion tuhoutumisesta, sillä sen on poistettava yhteys tuhoutuvaan Laina-olioon. Lopulta Laina-olio tuhoaa itsensä[23].

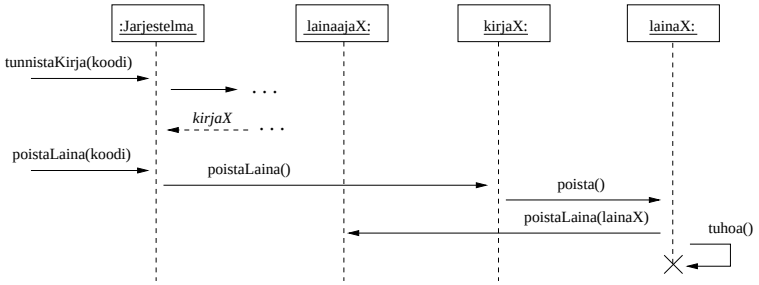
Suunnitelmaa vastaava sekvenssikaavio kuvassa 86. Luokkakaavion tulee muutamia uusia operaatioita:

- Jarjestelma-luokalle *poistaLaina(koodi)*

[23] Automaattisen roskienkeruun sisältävissä kielissä, kuten Javassa olio tuhoutuu itsestään siinä vaiheessa kun siihen ei ole mistään viitteitä. Joissain kielissä, esim. C++:ssa tuhoaminen täytyy tehdä erikseen käyttäen olion destruktoria.

- Kirja-luokalle *poistaLaina()*
- Laina-luokalle *poista()*
- Lainaaja-luokalle *poistaLaina(viite)*.

Täydennämme luokkaakaaviota operaatioiden osalta vasta myöhemmin.




Kuva 86: Käyttötapauksen palautus toimintaa vastaava sekvenssikaavio

**Käyttötapaus Lisää lainaaja**

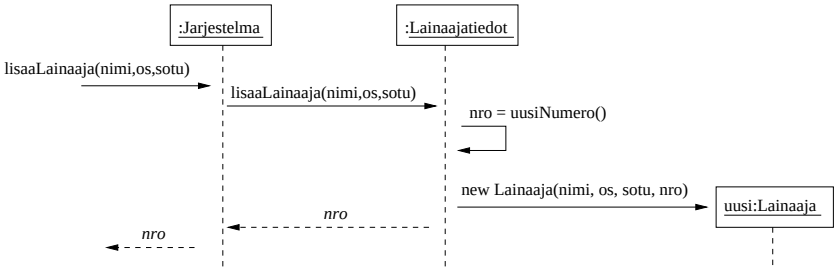
Käyttötapauksen ainoa tarvitsema operaatio on *lisaaLainaaja(nimi, osoite, sotu)*. Operaation on palautettava uusi kirjastokorttinumero, joka toimii luodun lainaajan yksikäsitteisenä tunnisteena. Edelleen ohjausoliona toimii Jarjestelma-olio.

Palautamme mieliimme luontiperiaatteen sivulta 81. Lainaajatiedot-olio sisältää kaikki Lainaaja-oliot, joten vastuu uuden lainaajan luomisesta annetaan Lainaajatiedot-oliolle.

Operaation toteutus on melko ilmeinen, eli Jarjestelma-olio delegoi operaation suorittamisen Lainaajatiedot-oliolle. Lainaajatiedot-olio generoi uudelle lainaajalle kirjastokorttinumeron, luo lainaajaolion ja luo siihen yhteyden. Operaatioita vastaava sekvenssikaavio kuvassa 87. Luokkakaavion uudet operaatiot ovat:

- Jarjestelma-luokalle *lisaaLainaaja(nimi, os, sotu)*
- Lainaajat-luokalle *lisaaLainaaja(nimi, os, sotu)* ja *uusiNumero()*

Operaatioista *uusiNumero()* on ainoastaan Lainaaja-luokan sisäisesti käyttämä. Tämän takia se on merkitty kuvan 90 luokkakaavioon private-määreellä varustettuna.




Kuva 87: Uuden lainaajan lisäys sekvenssikaaviona

### Käyttötapaus Lisää nimike

Uuden nimikkeen lisäämistä vastaava käyttötapaus on hyvin samankaltainen kuin uuden lainaajan lisääminen joten sekvenssikaaviota ei esitetä. Ainoan toteutettavan operaation *lisaaNimike(nimi, kirj, ISBN)* toteuttamiseksi on luotava uusi Nimike-olio. Koska Nimikkeet-olio sisältää kaikki yksittäiset Nimike-oliot, on selvää, että uuden olion luontivastuu delegoidaan sille.

### Käyttötapaus Lisää kirja

Ensimmäisen iteraation viimeisenä suunnittelun kohteena on käyttötapaus, joka huolehtii uuden kirjan lisäämisen järjestelmään. Uudelle kirjalle luodaan yksikäsitteinen tunniste, ja liitetään kirja sitä vastaavaan Nimike-olioon. Jos kirjaa vastaavaa nimikettä ei vielä ole olemassa, on nimike ensin luotava. Oletetaan seuraavassa tarkastelussa, että kirjaa vastaava nimike on jo olemassa.

Tapahtumien kulku käyttötapauksessa siis on seuraava (sivulta 69):

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. Merkitään kirja järjestelmään

Käyttötapauksen toteuttavat operaatiot tarkemmin eriteltynä (sivulta 72) ovat:

**haeNimike(nimi,kirj, ISBN)**

Tulostetaan tietoja vastaavaa nimikettä vastaavat tiedot.

**luoTunniste(nimi, kirj, ISBN)**

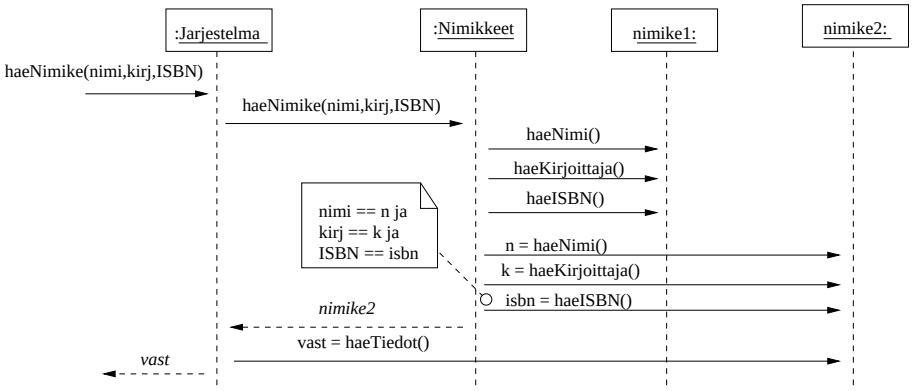
Pyydetään tietoja vastaavan nimikkeen uudelle kirjalle yksikäsitteinen tunnistenumero.

**lisaaKirja(nimi, kirj, ISBN, tunniste)**

Luodaan tietoja vastaavalle nimikkeelle uusi Kirja-olio, jolla tunnistekoodina parametrina oleva *tunniste*.

Koska Nimikkeet-olio tuntee kaikki yksittäiset Nimike-oliot, delegoidaan operaatio *haeNimike(nimi, kirj, ISBN)* sen vastuulle.

Operaation toteutuksen periaatteet ovat kuvan 88 sekvenssikaaviossa. Jarjestelma-olio delegoi operaation Nimikkeet-oliolle, joka käy läpi sisältämiään Nimike-olioita. Kun etsitty Nimike-olio löytyy (kuvassa toisena), palautetaan viite Jarjestelma-oliolle. Kuvassa etsitty nimike on toisena läpikäynnissä vastaan tuleva *nimike*2. Operaatiokuvauksen mukaan tulee kutsujalle (eli käyttöliittymälle) palauttaa nimikettä vastaavat tiedot.

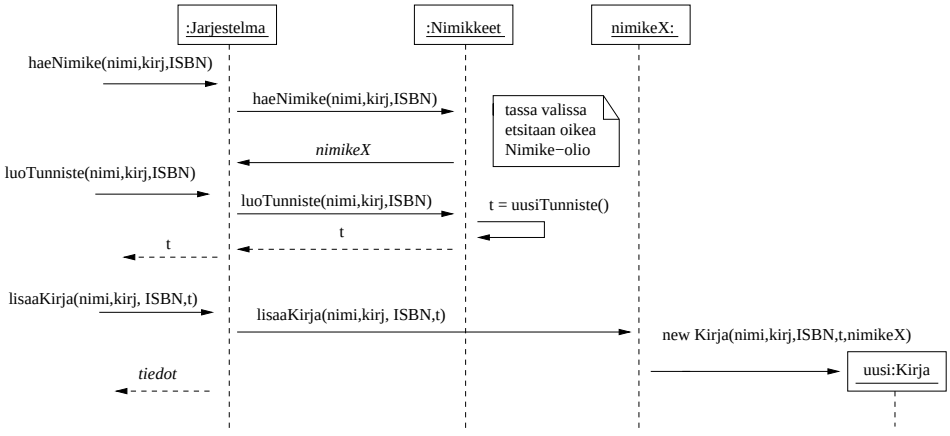



Kuva 88: Operaation haeNimike toimintaa kuvaava sekvenssikaavuo

Seuraava operaatio on luoTunniste(nimi, kirj, ISBN). Tunnisteen on siis eriteltävä kirja yksikäsitteisesti. Yksittäiset Nimike-oliot tuntevat ainoastaan omat kopionsa, ne siis eivät sovellu tunnisteen luojiksi. Luonnollinen vastuullinen olio operaation suhteen onkin Nimikkeet-olio.

Käyttötapauksen viimeinen operaatio on *lisaaKirja(nimi, kirja, ISBN, tunniste)*. On siis luotava uusi Kirja-olio, joka liittyy tiettyyn Nimike-olioon. Olion luominen kannattaa antaa sen Nimike-olion vastuulle, jonka alle uusi olio tulee. Viite oikeaan Nimike-olioon on jo tiedossa operaation suoritushetkellä aiemmin suoritetun haeNimike-operaation palauttamana.

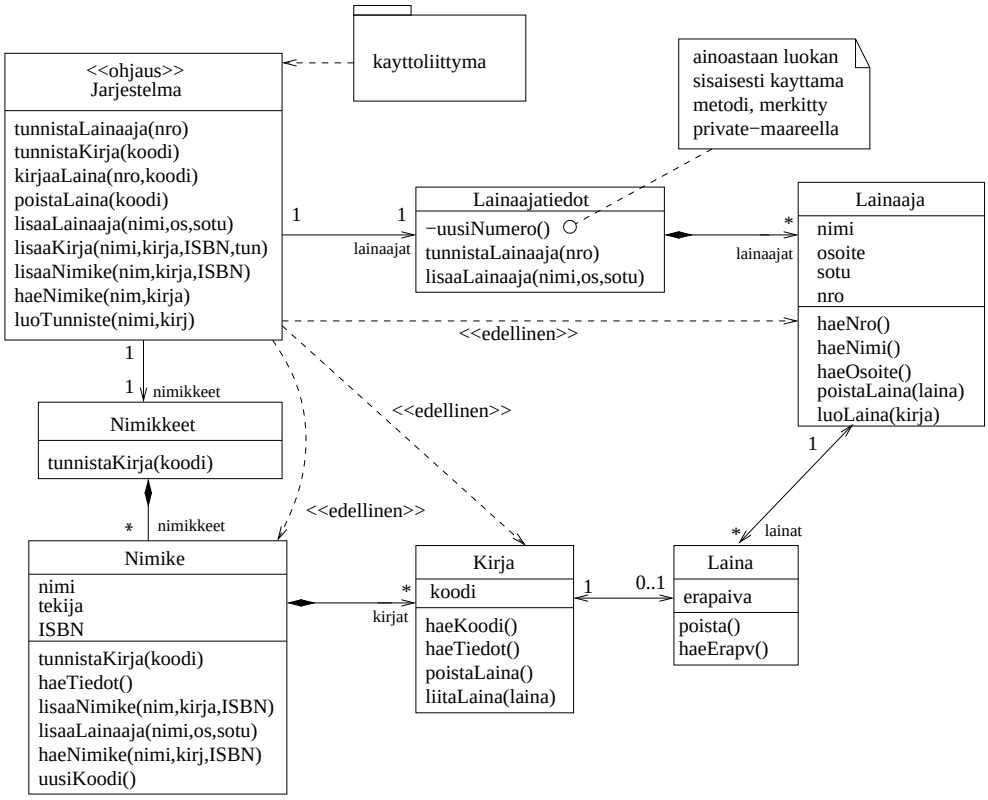
Käyttötapauksen *Lisää Kirja* toiminta on kokonaisuudessaan kuvan 89 sekvenssikaaviosta.




Kuva 89: Käyttötapauksen Lisää kirja toiminta.

Ensimmäiseen iteraatioon valittujen käyttötapausten toiminnallisuus on nyt suunniteltu. Kuvassa 90 tuloksena oleva suunnittelutason luokkamalli.

Tarkastellaan kaikkien käyttötapausten ohjauksesta vastaavaa Jarjestelma-luokkaa. Luokan metodit ovat toiminnallisuudeltaan varsin yksinkertaisia, sillä ne ainoastaan delegoivat vastuita muille luokille. Luokan rajapinnasta on kuitenkin muodostunut turhan epäyhtenäinen. Järjestelmään lisätään uutta toiminnallisuutta myöhemmissä iteraatioissa, eli



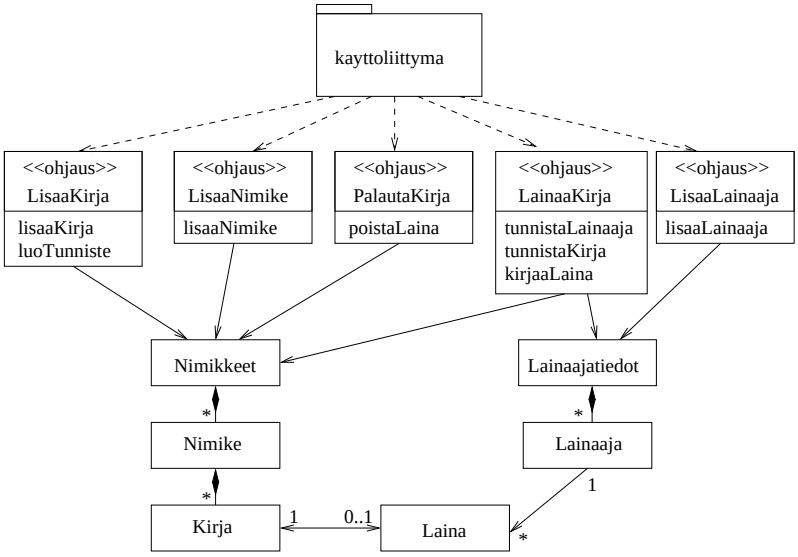

Kuva 90: Kirjastojärjestelmän ensimmäisen iteraation suunnittelutason luokkamalli

luokan rajapinta uhkaa tulevaisuudessa epäyhtenäistyä entisestään.

Sivulla 84 mainittiin, että vaihtoehtona yhdelle kaikesta huolehtivalle ohjausoliolle on varata jokaiselle käyttötapaukselle oma ohjausolio. Päätämme tehdä näin, ja järjestelmän luokkarakenne muuttuu kuvan 91 mukaiseksi. Nyt järjestelmän toiminnallisuuden laajentaminen kattamaan uusia käyttötapauksia voidaan hoitaa suunnittelemalla uusia ohjausluokkia entisten jäädessä ennalleen. Myös muutos tietyn käyttötapauksen toiminnallisuuden muuttamiseen paikallistuu ainoastaan yhteen luokkaan.

## 6.4 Toteutus ja testaus

Ensimmäiseen iteraatioon valitun toiminnallisuuden suunnittelun jälkeen on korkea aika aloittaa toteutus. Toteutusvaiheessa käy usein niin, että huomataan suunnitelmassa puutteita ja myös suunnitelmaa joudutaan muuttamaan. Jos on päätetty pitää suunnittelutason mallit ajantasaisina, aiheutuu muutoksista luonnollisesti vaivaa. Joskus taas valitaan strategia, missä UML-malleja käytetään yhden iteraation aikaisen suunnittelun apuvälineinä ja toteutuksen yhteydessä tulleita muutoksia ei heijasteta suunnittelun aikana tuotettuihin malleihin. Seuraavan iteraation yhteydessä piirretään sitten tarpeen vaatiessa uusia kaavioita suunnittelun tueksi. Työkaluavusteista takaisinmallinnusta voidaan tarvittaessa hyödyntää uuden iteraation alussa, eli tuotetaan takaisinmallinnustyökalun avulla edellisen iteraation aikana toteutetusta järjestelmästä UML-kaavioita, joita käytetään suunnittelun




Kuva 91: Luokkamalli, missä jokaisella käyttötapauksella on oma ohjausolio.

lähtökohtina.

Ketterissä menetelmissä suositellaan testauksen aloittamista mahdollisimman aikaisessa vaiheessa toteutusta. Tekemämme suhteellisen huolellinen mallinnus antaa hyvät lähtökohdat myös testaukselle, sillä järjestelmän luokkien operaatioiden halutusta toiminnallisuudesta on nyt suhteellisen selkeä kuva. Toteutuksessa kannattaakin edetä siten, että toteutettaville luokille ja operaatioille laaditaan automaattisesti suoritettavat testitapaukset esim. *JUnit-testauskehyksen* [12] avulla. Automaattisesti suoritettavat testitapaukset ajetaan aina, kun järjestelmään lisätään uutta toiminnallisuutta. Uuden toiminnallisuuden lisäämisen yhteydessä testitapausjoukkoa kasvatetaan uuden toiminnallisuuden varmistavilla testitapauksilla. Automaattisen testauksen avulla voidaankin varmistaa, että järjestelmään myöhempien iteraatioiden aikana lisättävä toiminnallisuus ei riko mitään olemassaolevaa.

Emme mene tällä kurssilla tarkemmin testauksen yksityiskohtiin. Toteutuksestakin tarkastelemme seuraavassa lähinnä sitä, miten luokkien väliset yhteydet voidaan toteuttaa Javassa.

Tarkastellaan luokkaa *Kirja* (ks. kuva 90). Luokasta on yhteys luokkiin *Laina* ja *Nimike*. Kirja-olioon liittyy 0 tai 1 Laina-olioa ja tasan yksi Nimike-olio. Yhteys, joka liittää olion maksimissaan yhteen olioon, toteutetaan Javalla käyttämällä normaalia olioviitettä, eli muuttujaa jonka tyyppinä on luokka. Luokan Kirja toteutus on alla.

```
public class Kirja {
    private int koodi;
    private Nimike nimike;
    private Laina laina;

    public Kirja(int k, Nimike n){
        koodi = k;
```

```
        nimike = n;
        laina = null;
    }


    public void liitaLaina(Laina l){ laina = l; }

    public void poistaLaina(){
        laina.poista();
        laina = null;
    }

    public int haeKoodi(){ /* koodia */ }
    public String haeTiedot(){ /* koodia */ }
}
```

Kun kirja luodaan, liitetään se vastaavaan nimikkeeseen. Nimike luo kirjan (ks. kuva 89), joten se välittää viitteen itseensä Kirja-luokan konstuktorissa. Alussa kirja ei ole lainassa, joten viite Laina-olioon saa arvon null. Myös palautuksen yhteydessä viitteen Laina-olioon arvoksi asetetaan null, sillä palautettu kirja ei ole liitetty mihinkään lainaan.

Kun kirja lainataan, huolehtii Lainaaja-olio Laina-olion luomisesta, ks. kuva 84. Laina-olio liitetään Kirjaan metodin *liitaLaina* avulla. Metodi asettaa Laina-viitteelle parametrinaan saamansa arvon.

Tarkastellaan seuraavaksi luokkaa Nimike. Nimike-olioon on liitetty useita $(0 \ldots n)$ Kirja-olioita. Tämäntyyppisen yhteyden toteuttaminen onnistuu parhaiten käyttämällä jotain Javassa olevaa *säiliöluokkaa*, esim. Arvo Wiklan [21] kurssimateriaalin luvussa 5.3 mainittua *ArrayList*:iä. ArrayList käyttäytyy ikäänkuin vaihtuvamittainen taulukko, jonka alkiomäärä voi elää suoritusaikana ja alkioiden maksimimäärää ei ole tarpeen tietään ArrayList-olioa luodessa.

Seuraavassa luokan Nimike toteutus:

```
public class Nimike {
    private String nimi;
    private String kirj;
    private String isbn;
    private ArrayList<Kirja> kirjat;

    public Nimike(String n, String k, String i){
        nimi = n; kirj = k; isbn = i;
        kirjat = new ArrayList<Kirja>();
    }

    public void lisaaKirja(int koodi){
        Kirja uusiKirja = new Kirja(koodi, this);
        kirjat.add( uusiKirja );
    }
```

```
    public Kirja tunnistaKirja(int koodi){
        for ( Kirja k: kirjat ){
            if ( k.haeKoodi()==koodi ) return k;
        }

        return null;
    }

    public String haeNimi(){ /* koodia */ }
    public String haeKirj(){ /* koodia */ }
    public String haeIsbn(){ /* koodia */ }
}
```

Luokan kentän *kirjat* tyyppi on siis *ArrayList<Kirja>*, eli olio, joka sisältää viitteitä kirjoihin. Konstruktori luo nimikkeelle tyhjän kirjalistan:

```
kirjat = new ArrayList<Kirja>();
```

Kirjan lisäys aiheuttaa uuden Kirja-olion luomisen sekä luodun Kirja-olion lisäämisen kirjalistalle käyttämällä *ArrayList*:issä määriteltyä operaatiota *add*.

Operaation *tunnistaKirja* (ks. kuva 83) tarkoitus on etsiä liittyykö nimikkeeseen kirja, jolla on parametrina annettu koodi. Nimike toteuttaa operaation vertaamalla jokaisen *ArrayList*:issa olevan kirjan koodia parametrina annettuun koodiin.

Kirjojen läpikäynti on toteutettu käyttäen Javan for-each-toistorakennetta (ks. Arto Wiklan [21] kurssimateriaalin luku 3.5):

```
for ( Kirja k: kirjat ){
    if ( k.haeKoodi()==koodi ) return k;
}
```

Muuttuja *k* viittaa nyt yksi kerrallaan kuhunkin *ArrayList*:issa *kirjat* olevaan kirjaan.

Emme käsittele järjestelmän toteutusta enempää. Kiinnostuneita varten kirjastojärjestelmän ensimmäisen iteraation toteutushahmotelma löytyy verkosta [13]. Toteutus noudattaa melko pitkälti edellisessä luvussa tehtyä oliosuunnittelua. Ainoastaan muutamia käyttöliittymältä ohjausolioille annettavia parametreja on muutettu hiukan.

## 6.5 Järjestelmän jatkokehitys myöhempien iteraatioiden aikana.

Seuraavissa iteraatioissa valitaan toteutettavaksi uusia käyttötapauksia tai laajennetaan jo toteutettujen käyttötapausten toiminnallisuutta.

Jokainen iteraatio sisältää yleensä samat vaiheet kuin läpikäymämme ensimmäinen iteraatio, eli ensin määritellään iteraation aikana toteutettava toiminnallisuus, laajennetaan

suunnitelmaa uuden toiminnallisuuden osalta ja toteutetaan sekä testataan toiminnallisuus.

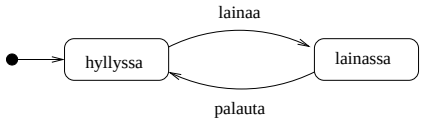
Ehkä haasteellisin asia, jonka rajasimme ensimmäisen iteraation ulkopuolelle on järjestelmän tiedon tekeminen pysyväksi, eli kirjoja ja lainausta koskevien tietojen tallettaminen levylle tai tietokantaan. Aihetta käsitellään jonkin verran kurssilla Tietokantojen suunnittelu.

# 7 Lisää UML:ää

## 7.1 Tilakaavio

Yksittäisten olioiden käyttäytymistapa voi olla erilainen eri tilanteissa. Esim. kirjastoesimerkissä luokan Kirja oliot käyttäytyvät eri tavalla ollessaan lainassa kuin ollessaan hyllyssä. Olion käyttäytyminen siis riippuu sen *tilasta*. Kun kirja on lainassa, ei sille voi suorittaa operaatiota lainaa. Kun kirja palautetaan, vaihtuu sen tila jälleen sellaiseksi, että uusi lainaus on mahdollista.

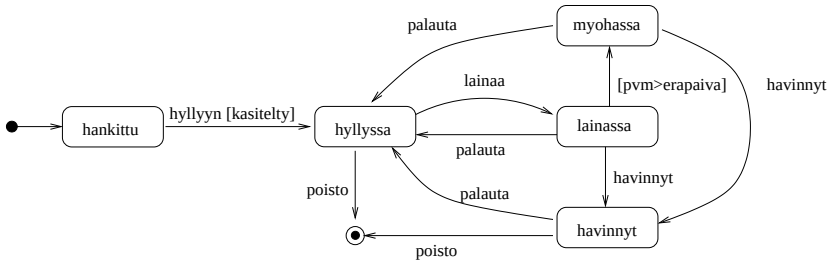
UML:n *tilakaavioiden* (engl. state machine diagram) avulla on mahdollista kuvailla olion tilasta riippuvaa käyttäytymistä. Kuvassa 92 Kirja-olion tilakaavio. Kirjalla on 2 tilaa, *hyllyssä*, *lainassa*. Tämän lisäksi kirjalla on myös mustana pallona kuvattu *alkutila* (engl. initial state). Tilojen välillä on *siirtymiä* (engl transition). Siirtymän saa yleensä aikaan jokin tapahtuma tai heräte, joka on esim. olion vastaanottama viesti eli operaatiokutsu. Siirtymässä ei välttämättä ole herätettä ollenkaan. Esim. alkutilasta siirrytään itsestään tilaan *hyllyssä*, eli syntyessään kirja menee heti hyllyyn.




Kuva 92: Kirjan toimintaa kuvaava tilakaavio

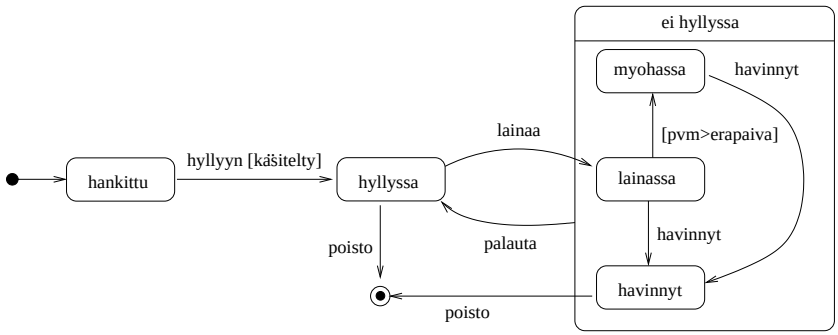
Käytännössä tila tarkoittaa tiettyjen atribuuttien arvon sopivaa kombinaatiota tai yhteyksien tilaa. Kun kirja on lainassa, liittyy siihen Laina-olio. Hyllyssä olevaan kirjaan taas ei liity lainaa. Eli kirjan tila selviää Laina-olioon yhteyden olemassaolon perusteella. Jos erillistä lainaolioa ei olisi, voisi tila selvitä esim. totuusarvoisen attribuutin *lainassa* perusteella.

Yksityiskohtaisempi tilakaavio kirjasta on kuvassa 93. Kuvassa on mukana *lopputila* (engl. final state), eli jos kirjalle suoritetaan operaatio *poisto*, menee kirja lopputilaan, eli olio tuhoutuu. Siirtymään tilasta *lainassa* tilaan *myohassa* liittyy nyt *ehto*, eli jos eräpäivä on ylitetty, suoritetaan siirtymä. Myös siirtymä tilasta *hankittu* tilaan *hyllyssä* sisältää ehdon, eli operaation *hyllyyn* voi suorittaa vain, jos ehto *käsitelty* on tosi (eli kirjan tiedot on kirjattu järjestelmään).



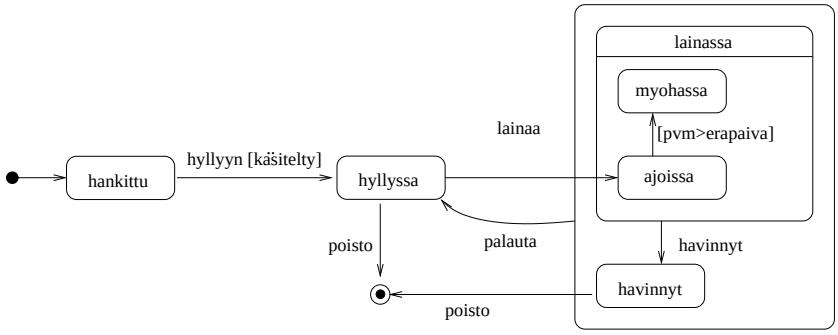

Kuva 93: Kirjan yksityiskohtaisempi tilakaavio

Tilat *myohassa*, *lainassa* ja *havinnyt* ovat siinä mielessä samanlaisia, että niistä kaikista on tapahtumalla *palauta* siirtymä tilaan *hyllyssä*. Tämäntyyppisissä tilanteissa kaaviota on mahdollista yksinkertaistaa *sisäkkäisten tilojen* avulla. Kuvaan 94 on lisätty *ylitila* (engl. superstate) *ei hyllyssä*, joka sisältää edellä mainitut kolme samankaltaista *alitilaa* (engl. substate). Ylitila sisältää siirtymän tapahtumalla *palauta* tilaan *hyllyssä*. Kyseessä on lyhennysmerkintä sille, että kyseinen siirtymä olisi jokaisessa alitilassa. Siirtymä ylitilan sisälle tapahtuu edelleen suoraan tilaan *lainassa*. Huomattavaa on myös, että siirtymä *poisto* liittyy ainoastaan alitilaan *hävinnyt*.




Kuva 94: Ylitilan käyttö

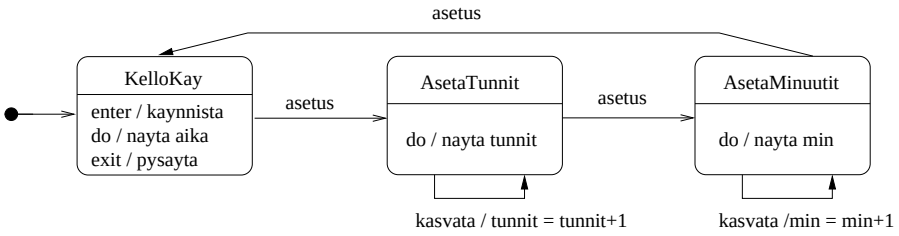
Koska tilat *myohassa* ja *lainassa* sisältävät samanlaisen siirtymän tilaan *havinnyt*, on kuvassa 95 ylitilan sisälle lisätty toinen ylitila, jonka avulla yhteinen siirtymä saadaan esitettyä. On makuasia missä määrin sisäkkäisiä ylitiloja kannattaa käyttää. Jossain vaiheessa liiallinen ylitilojen käyttö alkaa vähentää kaavion luettavuutta.




Kuva 95: Sisäkkäiset ylitilat

Kuvassa 96 on esitetty yksinkertaisen digitaalikellon toiminta tilakaaviona. Kellolla on kolme tilaa. Hyödynnämme tässä erimerkissä mahdollisuutta liittää tiloihin toimintoja. Oltaessa tilassa *KelloKay* näytetään aikaa, tämä ilmaistaan tilaan liitetyllä toiminnalla *do / nayta aika*. Tilaan tultaessa kello käynnistyy, tätä ilmaisee tilaantulotapahtuma *enter / kaynnista*. Tilasta poistuttaessa kello pysähtyy, eli suoritetaan tilasta poistumistapahtuma *exit / pysayta*. Toiminnolla *asetus* siirrytään tilaan, jossa voidaan asettaa tuntiviisarin (tai tuntinumeron koska kyseessä digitaalikello) aika toiminnolla *kasvata*. Tapahtuman aikaansaama toimenpide on merkitty muodossa *kasvata / tunnit = tunnit +1* eli ensin on merkitty tapahtuma (operaatiokutsu *kasvata*), jonka perässä /-merkin jälkeen tapahtuman

aikaansaama toimenpide (muuttujan *tunnit* arvon kasvatus). Tilassa oltaessa näytetään tuntiviisarin aikaa. Vastaavasti mallissa on tila minuuttiajan asettamiselle.




Kuva 96: Kellon tilakaavio

Tarkastellaan vielä hieman suurempana esimerkkinä tarkennettua mallia kellosta. Kellon tilakaavio kuvassa 97 (a). Kello siis koostuu moottorista ja näytöstä, joiden toimintalogiikka on kuvattu omana tilakaavionaan.

Moottori, jonka tilakaavio on kuvassa 97 (b), voi olla käynnissä tai poissa päältä. Ollessaan käynnissä moottori laskee kuluneita sekunteja ja 60 sekunnin välein se kutsuu näytön operaatiota *tick*. Tilakaavioon tämä on merkitty tapahtumana $^{\wedge}naytto.tick$, joka siis suoritetaan kun sekunti kului ja edellisestä tapahtumasta on kulunut 60 sekuntia. Muussa tapauksessa aina sekunnin kuluttua ainoastaan kasvatetaan laskuria, joka laskee edellisestä näytölle suoritetusta *tick*-operaatiosta kulunutta aikaa. Toiselle oliolle kohdistettu operaatiokutsu siis merkitään siten, että ensin tulee ^-merkki, jonka perässä operaation kohdeolio, jonka jälkeen pisteellä erotettuna kohdeoliolle suoritettava operaatio.

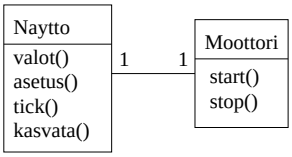
Näyttö on kuvassa 97 (c). Näyttö sisältää edellisen esimerkkimme ajan asetusominaisuuden. Muutoksena nyt se, että kun kello siirtyy tuntien asetukseen, eli pois ylitilasta *NaytaAika*, pysäyttää se moottorin tapahtumalla $^{\wedge}moottori.stop$. Vastaavasti palatessaan ajannäyttötilaan käynnistää näyttö moottorin tapahtumalla $^{\wedge}moottori.start$. Näyttäessään aikaa, kellon näyttö voi olla joko valaistu tai valaisematon. Kello valaistaan tapahtumalla *valot*. Tilamallista näemme että kellon aikaa ei voi asettaa valon ollessa päällä, sillä siirtymä *AsetaTunnit*-tilaan on ainoastaan tilasta *Valaisematon*.

*tick*-tapahtuma (mikä siis on moottorin aikaansaama) muuttaa kellon aikaa minuutilla eteenpäin. Kellonajasta riippuen myös tuntien aika saattaa muuttua. Tilamallissa haarautuvaa siirtymää kuvataan salmiakki-symbolilla $\diamond$ ja siihen liittyvillä ehdoilla. Eli kun *tick* tapahtuu, jos minuuttien arvo on eri kuin 59, tehdään tominto $min{+}{+}$, muussa tapauksessa (eli minuuttiviisari nollautuu) asetetaan tuntiviisarin uusi arvo. Huomioidaan, että myös tuntiviisari voi nollautua.

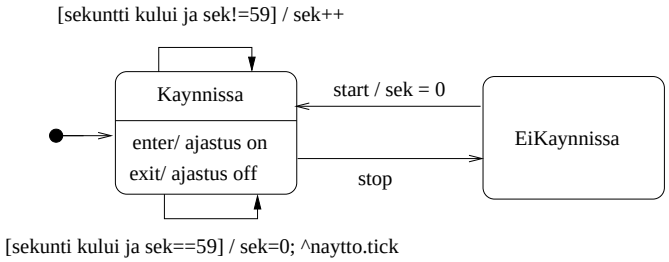
Ylitila *NaytaAikaa* sisältää nyt oman alkutilansa. Tämä merkitsee sitä, että jokainen ylitilaan kohdistuva siirtymä johtaa alkutilaan, eli käytännössä aina mentäessä ylitilaan *NaytaAikaa*, päädytään alitilaan *Valaisematon*. Eli jos näyttö on valaistu, se muuttuu valaisemattomaksi minuuttien edetessä eli tapahtuman *tick* yhteydessä. Samoin ajan asetuksesta palataan aina valaisemattomaan tilaan.

Entä jos kellon näyttö olisi mallinnettava siten, että ajan asettaminen on mahdollista aloittaa myös näytön ollessa valaistuna, ja että tässä tapauksessa ajan asetuksen jälkeen pa-

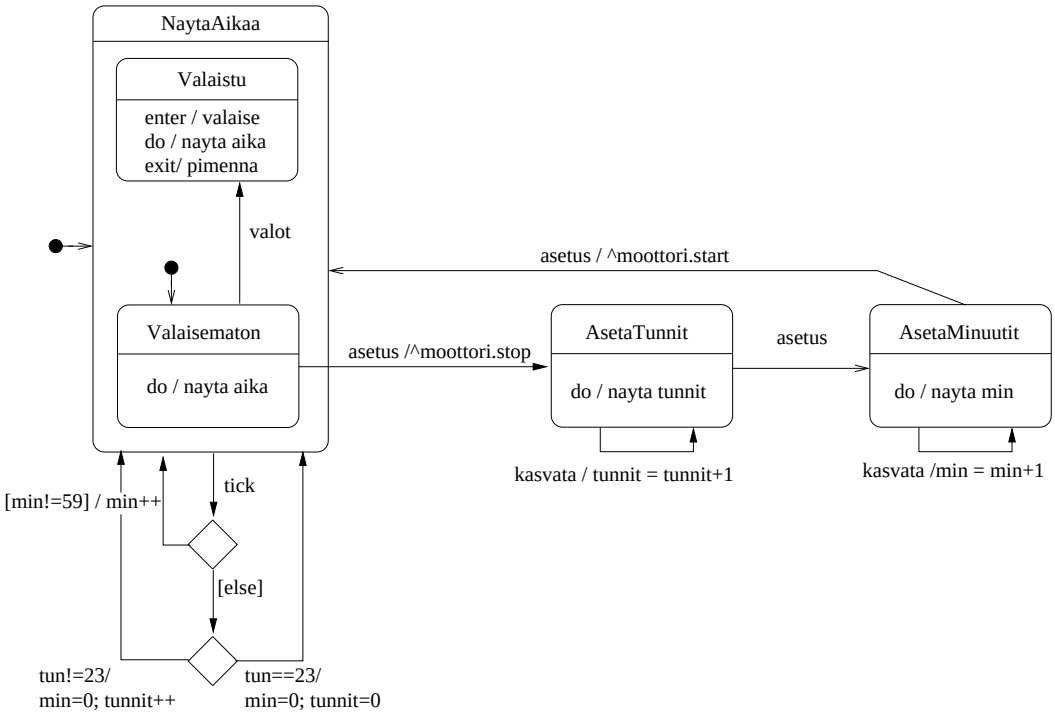
(a) Kellon luokkakaavio




(b) Moottorin tilakaavio

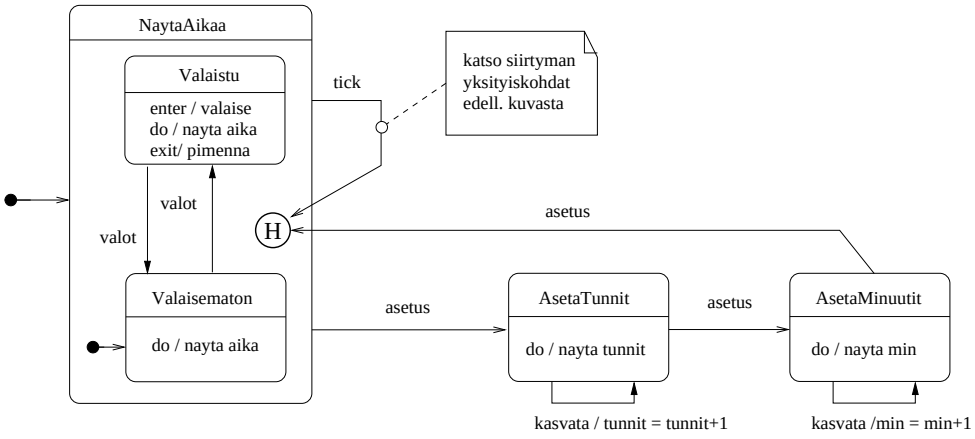



(c) Nayton tilakaavio




Kuva 97: Tarkemmin kuvattu kello tilakaaviona

lataan takaisin tilanteeseen missä näyttö on valaistu. Ratkaisu on kuvassa 98. Siirtymä tapahtumalla *asetus* on nyt merkitty ylitilalle *NaytaAikaa*. Eli tuntien asetustilaan voidaan siirtyä myös näytön ollessa valaistuna. Paluusiirtymän kohdetilana on nyt ylitilan sisällä oleva ympäröity H eli *historiatila*. Historiatilaan palaaminen tarkoittaa, että palataan siihen alitilaan, missä oltiin kun ylitilasta poistuttiin. Eli jos ajan asetukseen on menty tilasta *Valaisematon*, palataan samaan tilaan, vastaavasti jos ajan asetukseen mentiin tilasta *Valaistu*. Myös *tick*-tapahtuma johtaa nyt takaisin siihen alitilaan missä siirtymä tapahtuu, eli kello säilyy valaistuna ajan edetessäkin. Valaisemattomaan tilaan päästään takaisin toistamalla toiminto *valot*.

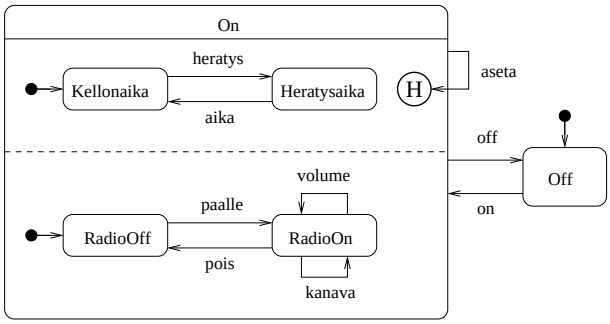



Kuva 98: Historiatilan käyttö

Äskeisissä esimerkeissä käyttämämme alitilat olivat ns. tai-alitiloja, eli järjestelmä on yhdessä alitilassa kerrallaan. UML:ssä on mahdollisuus määritellä myös *rinnakkaisia alitiloja*, jolloin yhden tilan sisällä on ikänkuin useita rinnakkaisia tilakaavioita. Tarkastellaan esimerkkinä yksinkertaista herätyskelloradioa. Jos laite on päällä, näyttää se kellonaikaa tai herätysaikaa ja radio voi olla päällä tai poissa päältä. Päällä ollessaan laitteella on siis neljä eilaista toimintakombinaatiota:

- kellonaika, radio poissa päältä
- kellonaika, radio päällä
- herätysaika, radio poissa päältä
- herätysaika, radio päällä

Periaatteessa nämä kaikki voitaisiin kuvata omina tiloinaan. Selkeämpään ratkaisuun päädytään, jos laitteen päälläolo kuvataan ylitilana (*on*), joka sisältää rinnakkaiset tilat, jotka kuvaavat erikseen kello-osaa ja radio-osaa. Ratkaisu on esitetty kuvassa 99. Ylitilan sisällä rinnakkaiset osat on erotettu katkoviivalla. Käynnistyessään laite siis on yhtäaikaa tiloissa *Kellonaika* ja *RadioOff*. Kello-osan tapahtumat eivät vaikuta radioon ja päinvastoin. Huomaa, että sekä kellolle, että herätykselle voi asettaa uuden ajan toiminnolla *aseta*. Toiminto palaa historiatilaan, eli samaan tilaan mistä lähdettiin. Kun laite suljetaan toiminnolla *off*, poistutaan rinnakkaisista tiloista.




Kuva 99: Herätyskelloradio

Tilakaavioita voi käyttää ohjelmiston vaatimusmäärittelyvaiheessa tai suunnittelussa. Kirjan ja herätyskelloradion tilamallit ovat esimerkkejä määrittelyvaiheen tilamalleista, joissa lähinnä kuvaillaan sovellusalueen olion toimintalogiikkaa ottamatta millään tavalla kantaa toteutukseen.

Kuvan 96 pelkistetty versio kellosta ottaa lähinnä kantaa kellon ulkoiseen toimintaan, joten kyseessä on määrittelyvaiheen tilamalli. Kuvassa 97 esitetty kellon tilamalli taas ottaa jo jonkin verran kantaa kellon sisäiseen rakenteeseen (jakautuminen näyttöön ja moottoriin), joten kyseessä on suunnitteluvaiheessa määritelty tilamalli. Tarkasti määritellyistä tilamalleista voidaan generoida jossain määrin myös olion toteuttavaa ohjelmakoodia.

Tilamalli on mielekästä tehdä ainoastaan luokista, joiden olioilla on selkeä elinkaari, joka sisältää erilaisia toimintatiloja, joissa olio on ulkoiselta käyttäytymiseltään erilainen. Tilamallinnus on avainasemassa esim. tietoliikenneprotokollien tai reaaliaikajärjestelmien mallinnuksessa. Kirjastojärjestelmän tyyppisten tiedonkäsittelyjärjestelmien olioiden tilakäyttäytyminen on yleensä aika triviaalia ja tilamallit eivät yleensä ole tarpeen.

## 7.2 Aktiviteettikaavio

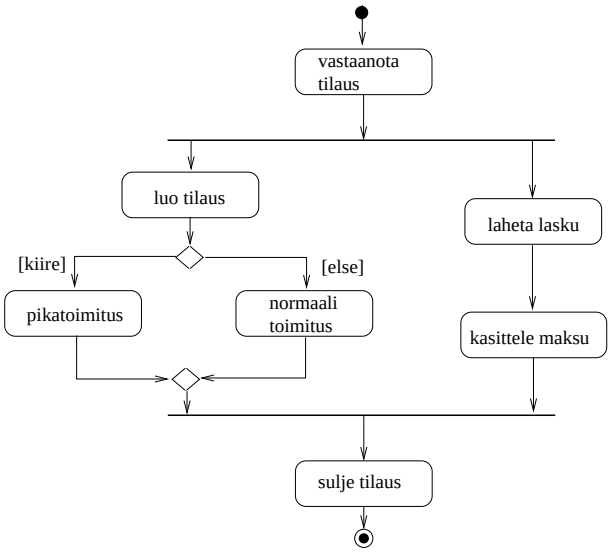
Tilakaaviot kuvaavat yksittäisen olion toimintaa. *Aktiviteettikaavioilla* (engl. activity diagram) taas on mahdollisuus kuvata suurempaa toiminnallista kokonaisuutta, esim. kokonaista liiketoimintaprosessia tai tiedon ja työn kulkua järjestelmässä monen toimijan kannalta. Joissain tapauksissa aktiviteettikaaviot sopivat myös käyttötapausten kuvaamiseen.

Tarkastellaan ensin aktiviteettikaavioiden käyttöä liiketoimintaprosessien kuvaamisessa. Kuvassa 100 esitetään mitä toimintoja liittyy tilauksen vastaanottamiseen, toimittamiseen ja laskutukseen.

Aloitussymboli ohjaa ensimmäiseen *toimintoon* (engl. action), eli tilauksen vastaanottoon. Tämän jälkeen kontrolli *haarautuu* (engl. fork) kahteen rinnakkain etenevään toimintosarjaan. Vasemman ja oikean haaran toiminnot siis etenevät rinnakkain toisistaan riippumattomina. Oikea haara kuvaa laskutuksen (laskun lähetys ja maksun vastaanotto) ja vasen haara toimituksen. Toimitus sisältää vielä haarautumisen pikatoimitukseen ja normaaliin toimitukseen, näistä siis valitaan ainoastaan toisen haaran toiminto. Laskutus- ja toimintohaara *yhdistyvät* (engl. join). Eli yhdistymissymbolin (viiva johon saapuu kaksi nuolta ja

josta lähtee yksi nuoli) jälkeen kontrolli jatkaa siis ainoastaan yhdessä haarassa ja toiminnoissa ei ole enää rinnakkaisuutta. Viimeisen toiminnon (sulje tilaus) jälkeen aktiviteetti loppuu.

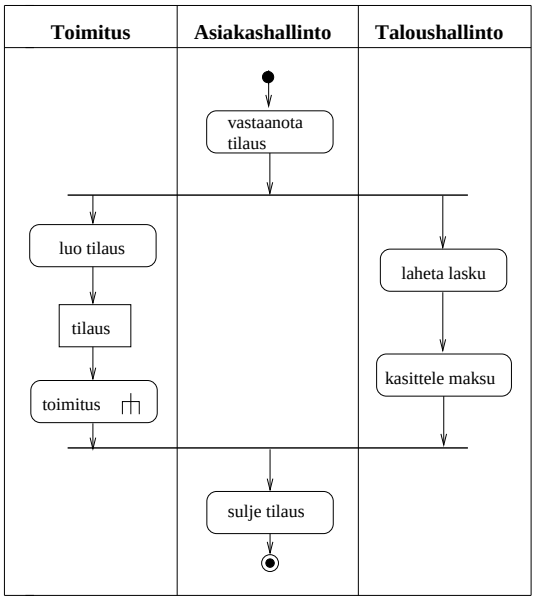
Samankaltaisuudesta huolimatta aktiviteettikaavioita ei pidä sekoittaa tilakaavioihin. Aktiviteettikaavioissa siis kuvataan sarja toimintoja ja toimintojen suoritusjärjestys. Toiminnot on kuvattu pyöreäreunaisina suorakulmioina (aivan kuten tilat tilakaaviossa) ja toimintojen peräkkäisyys niitä yhdistävinä nuolina ja rinnakkaisuus haarautumisen avulla.



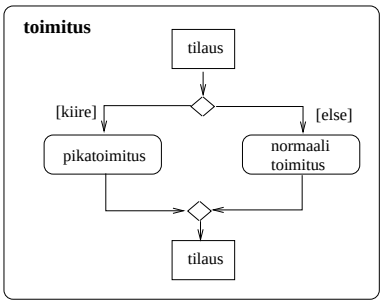

Kuva 100: Tilauksen vastaanotto, toimitus ja laskutus aktiviteettikaaviona

Esimerkissämme aktiviteettikaavio siis kuvaa mitä yhdelle tilaukselle tapahtuu sen elinkaaren aikana. Kaavion eri toiminnot ovat todennäköisesti organisaation eri toimijoiden suorittamia. Tätä voidaan korostaa, jakamalla kaavio *kaistoihin* (engl. swimlane), eli erillisiin osiin, jotka jaottelevat sen kuka on vastuussa toiminnon suorittamisesta. Tilauksen käsittely jaettuna toimituksen, asiakashallinnon ja taloushallinnon vastuisiin on esitetty kuvassa 101. Toiminnon *luo tilaus* seurauksena syntyvä *Tilaus*-olio on otettu malliin mukaan. Tilaus-olio siirtyy parametrina toimintoon *toimitus*, joka on nyt mallinnettu ainoastaan karkealla tasolla sisältäen viitteen (haarukkasymboli) tarkentavaan aktivitettikaavioon, joka on kuvassa 102.

Kuvassa 103 on mallinnettu aktivitettikaaviona laitoksen Pro Gradu -tutkielmien hyväksymiseen liittyvä käytäntö. Eli gradun valmistuttua opiskelija toimittaa sen tarkastajille. Tarkastettuaan gradun tarkastajat laativat lausunnon, joka toimitetaan sekä opiskelijalle että linjan vastuuprofessorille. Tässä kohdassa toiminta haarautuu ja samaan aikaan tarkastajat toimittavat gradun tiedot kansliaan. Saatuaan opiskelijalta tiedoksisaanti-ilmoituksen ja tarkastajalta gradun tiedot, toimittaa kanslia gradusta yhteenvedon graduvastaavalle. Tarkastettuaan yhteenvedon, toimittaa graduvastaava sen johtoryhmälle. Johtoryhmä täydentää kokouskutsua gradun osalta ja pitää kokouksen. Kokouksen jälkeen johtoryhmä pyytää graduvastaavaa päivittämään laitoksen gradutilastoa. Graduvastaava toimittaa tiedon johtoryhmän hyväksynnästä kansliaan, jossa kirjataan opiskelijalle suoritusmerkintä.



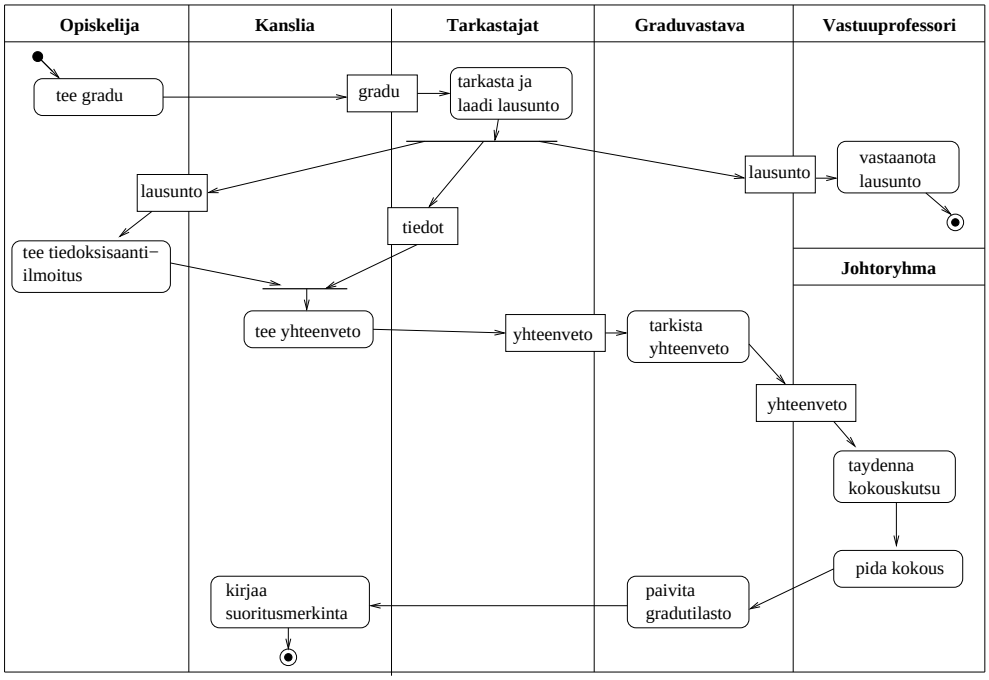

Kuva 101: Tilauksen käsittely aktiviteettikaaviona




Kuva 102: Toiminnon toimitus tarkemmin määrittelevä aliaktivitettikaavio

Tämän jälkeen kaikki gradun hyväksymiseen liittyvät toimenpiteet on suoritettu.

Aktiviteettikaaviona laadittu prosessin kuvaus on pakostakin hiukan ylimalkainen. Esim. eri toimenpiteiden välillä välitettävä tieto on määriteltävä tarkemmin kaavion ulkopuolella, esim. luokkakaavioiden avulla. Aktivitettikaavio kuitenkin antaa joissain tilanteissa pelkkää tekstuaalista kuvausta paremman yleiskuvan prosessin kulusta.

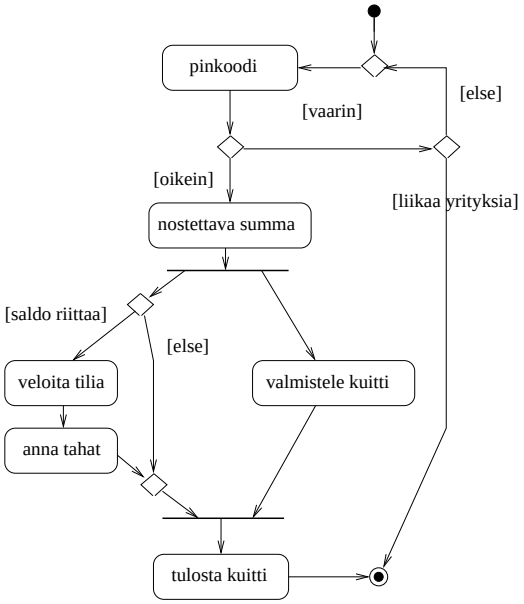



Kuva 103: Pro Gradu -tutkielmien hyväksymismenettely

Aktiviteettikaaviot ovat tällä kurssilla esitellyistä UML-kaavioista varmasti kaikkein vähiten käytännössä hyödynnetty kaaviotyyppi, jos ajatellaan ohjelmistojen määrittelyä ja suunnittelua. Jos aktivitettikaavioita halutaan soveltaa ohjelmistotuotannossa, tapahtunee soveltaminen lähes yksinomaan vaatimusmäärittelyn aikana. Aktiviteettikaavioiden avulla voidaan ohjelmiston vaatimuksien kartoitusvaiheessa esim. kuvailla työprosesseja[24], joita halutaan automatisoida kehitettävän ohjelmiston avulla. Näin ohjelmiston kehittäjät oppivat ymmärtämään paremmin sovellusaluetta ja pystyvät kommunikoimaan asiakkaan kanssa paremmin ja vaatimusten määrittely helpottuu.

Joissain tapauksissa aktiviteettikaaviota voidaan hyödyntää yksittäisen käyttötapauksen toimintalogiikan kuvaamiseen. Esimerkiksi käyttötapaus *nostotapahtuma pankkiautomaatista* voitaisiin mallintaa kuvan 104 aktiviteettikaaviolla. Esimerkin aktiviteettikaavio siis kuvaa ainoastaan yhden käyttötapauksen aikaisia toimintoja, eikä esimerkiksi yritäkään kuvailla pankkiautomaatin toimintaa kokonaisuudessaan.

[24]Sivulla 3 vesiputousmallin havainnollistamisessa käytetty diagrammi on aktiviteettikaavio!




Kuva 104: Nostotapahtuman aktivitettikaavio

# 8 Loppusanat

Monisteessa on käyty läpi komponenttikaavioita lukuunottamatta tärkeimmät UML-kaaviot. Kaikkiin kaavioiden yksityiskohtiin ei ole ehditty paneutua. UML-standardi on valtavan laaja ja jokaisen detaljin tunteminen ei ole UML:n aktiivisellekaan soveltajalle tarpeen. Tärkeämpää on osata soveltaa kaavioita tarkoituksenmukaisesti. Sovellusosaamisen myötä voi tarvittaessa myös laajentaa UML-osaamistaan. Koska yksi mallien täkeimpiä tarkoituksia on kommunikoida määrittely- ja suunnitteluideoita useiden ihmisten kesken, onkin mallinnuksessa syytä pitäytyä sellaisissa UML:n piirteiden käytössä jota ovat yleisesti ymmärrettyjä.

Itse UML-standardi ei anna mitään ohjeistusta sille, kuinka UML:ää tulisi soveltaa ohjelmistoprosessin aikana. Luvussa 6 hahmotellaan yhtä tapaa, miten ohjelmiston kehitysprosessia voi viedä eteenpäin UML-malleihin nojautuen, eli *malliperustaisesti*. Kurssin puitteissa ei varsinkaan oliosuunnitteluun ehditty paneutumaan kuin hyvin pintapuolisesti.

Seuraavana askeleena kurssin jälkeen kannattaakin harkita tarkempaa tutustumista oliosuunnitteluun. Larmanin kirjan [16] lisäksi hyvä lähde on Robert Martinin *Agile Software Development, Principles, Patterns, and Practices* [17]. Molemmissa kirjoissa käsitellään laajasti *suunnittelumallien* (engl. design patterns) soveltamista ohjelmistosuunnittelussa. Suunnittelumallilla tarkoitetaan yleistä tapaa jonkin usein esiintyvän ongelman ratkaisemiseksi. Suunnittelumalleja on alettu dokumentoimaan tietojenkäsittelyssä 1990-luvun alkupuolelta asti. Suunnittelumallien käsitteen laajempaan tietoisuuteen tuonut Gamman ja kumppanien kirja *Design Patterns: Elements of Reusable Object-Oriented Software* [10] on edelleen ajankohtainen lähde asiasta.

Toisen vuoden keväällä ohjelmassa oleva kurssi *ohjelmistotuotanto* jatkaa tämän kurssin

aihepiirin parissa. Kurssilla paneudutaan tarkemmin ohjelmiston elinkaaren eri vaiheisiin sekä prosessimalleihin. Tärkeä näkökulma kurssilla on ohjelmistojen tuottaminen useita henkilöitä sisältävissä projekteissa.

# Viitteet


[1] Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice-Hall, second edition, 2003.

[2] Ken Beck and Cynthia Anders. *Extreme Programming Explained: Embrace Change*. Addison-Welsey, second edition, 2004.

[3] Grady Booch. *Object Oriented Design with Applications*. Benjamin/Gummings, 1991.

[4] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley, second edition, 2005.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, and Michael Stal. *Pattern-Oriented Software Architecture. A System of Patterns*, volume 1. John Wiley and Sons, 1996.

[6] Peter Coad, David North, and Mark Mayfield. *Object Models. Strategies, Patterns and Applications*. Prentice Hall, 1997.

[7] Alistair Cockburn. *Writing Effective Use Cases*. Addison Wesley, 2001.

[8] Hans-Erik Eriksson, Magnus Penker, Brian Lyons, and David Fado. *UML 2 Toolkit*. Wiley Publishing, Inc, 2004.

[9] Martin Fowler. *UML Distilled, A Brief Introduction to the Standard Object Modeling Language*. Addison-Wesley, third edition, 2004.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[11] Ivar Jacobson, Magnus Christiansen, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering, A Use Case Driven Approach*. Addison Wesley, 1995.

[12] http://www.junit.org/.

[13] http://www.cs.helsinki.fi/u/mluukkai/ohma/kirjasto/.

[14] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, second edition, 2000.

[15] Craig Larman. *Agile and Iterative Development: A Managers Guide*. Addison Wesley, 2003.

[16] Craig Larman. *Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, third edition, 2005.

[17] Robert Martin. *Agile Software Development, Principles, Patterns, and Practice*. Prentice Hall, 2002.

[18] James Rumbaugh, Michael Blaha, William Lorensen, Frederick Eddy, and William Premerlani. *Object-Oriented Modeling and Design*. Prentice-Hall, 1992.

[19] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2001.

[20] Ian Sommerville. *Software Engineering*. Addison-Welsey, eight edition, 2005.

[21] http://www.cs.helsinki.fi/u/wikla/Ohjelmointi/Sisalto/.

[22] Rebecca Wirfs-Brock and Allan McKean. *Object Design. Roles, Responsibilities and Collaborations*. Addison Wesley, 2003.

[23] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.