

Ohjelmistotekniikan menetelmät

Matti Luukkainen, Olli-Pekka Mehtonen

Helsingin Yliopisto, TKTL

Kesällä 2016

Wikipedia:

Ohjelmistotuotanto on yhteisnimitys niille työntekijöille ja työnjohtajille, joita käytetään, kun tuotetaan tietokoneohjelmia sekä monista tietokoneohjelmista koostuvia tietokoneohjelmistoja.

Ohjelmistotuotanto laajasti ymmärrettynä kattaa kaiken tietokoneohjelmistojen valmistukseen liittyvän prosessinhallinnan sekä kaikki erilaiset tietokoneohjelmien valmistamisen menetelmät.

Ohjelmistotuotanto kattaa siis kaikki aktiviteetti, jotka tähtää tietokoneohjelmien tai -ohjelmistojen valmistukseen.

Mallintaminen

- ▶ Tietokoneohjelman valmistamisen menetelmiin liittyy **mallintaminen**, eli kyky tuottaa erilaisia kuvaauksia, joita tarvitaan ohjelmiston kehittämisen yhteydessä
- ▶ Mallit toimivat kommunikoinnin välineinä
- ▶ *Mitä ollaan tekemässä, miten ollaan tekemässä, mitä tehtiin?*

Kurssista

- ▶ Aikataulu ja kurssimateriaali:
 - ▶ <https://github.com/Ooppa/OTM-kesa16>
- ▶ Laskarit:
 - ▶ Aloitetaan jo ensimmäisellä viikolla
 - ▶ Yhteensä 6kpl, 3h per tilaisuus
- ▶ Arvostelu:
 - ▶ Kurssin kokonaispistemäärä on 36p
 - ▶ Kurssikoe 22p
 - ▶ Laskareista 14p, jotka koostuu...
 - ▶ Paikanpääällä tehtävistä 6p (1p per kerta)
 - ▶ Etukäteen tehtävistä 8p (90% → 8p)
 - ▶ Noin 32p → arvosana 5
 - ▶ Läpipääsy vaatii puolet kurssikokeen pisteistä ja puolet laskaripisteistä, eli ainakin 18p

Laskarit

► Etukäteen tehtävät:

- ▶ Keskimäärin 6 tehtävää viikossa
- ▶ Tehdään etukäteen niin, että vastauksia voidaan tarkastella ryhmissä, eli tulostettuna, läppärillä tai verkossa
- ▶ Pisteitä saa vaikka kaikki ei ole oikein
- ▶ Yrityksestä jää kuitenkin aina jälki!

► Paikanpäällä tehtävät:

- ▶ Ryhmätyöskentelyä
- ▶ Tehdään niin paljon kuin ehtii, mutta työskennellään aktiivisesti
- ▶ Laskarit eivät ole paja, paikalla on oltava alusta loppuun!

► Laskariajat

- ▶ Ryhmä 1: Ti ja To klo 10-13 B222
- ▶ Ryhmä 2: Ti ja To klo 13-16 B222

Ohjelmistotuotantoprosessi

Miksi prosessi kun voi vain tehdä?

Ohjelmistotuotantoprosessi

Miksi prosessi kun voi vain tehdä?

- ▶ Pienissä itselle tehtävissä projekteissa voidaan tuottaa sovellusta noudattamatta systematiikkaa
- ▶ Voidaan helposti väsätä kasaan sovellus, joka "*toimii*"
- ▶ Tämä menetelmä ei toimi isommille, monen hengen projekteissa asiakasta varten tuotetuille ohjelmille (Toimiiko sovellus niin kuin alunperin haluttiin?)
- ▶ Rakenne epämääriäinen → Ylläpidettävyys huono
- ▶ Ratkaisu: Kehitellyt erilaisia menetelmiä ohjelmistotuotantoprosessin systematisoimiseksi¹
- ▶ Mitä menetelmää tulisi käyttää? Hyvä kysymys!

¹https://en.wikipedia.org/wiki/List_of_software_development_philosopies

Ohjelmistotuotantoprosessin vaiheet

Yhteenvetö vaiheista

1. Vaatimusanalyysi- ja määrittely

- ▶ Mitä halutaan?

2. Suunnittelu

- ▶ Miten tehdään?

3. Toteutus

- ▶ Ohjelmoidaan

4. Testaus

- ▶ Varmistetaan että toimii niin kuin halutaan

5. Ylläpito

- ▶ Korjataan bugeja ja laajennetaan ohjelmistoa

Ohjelmistotuotantoprosessin vaiheet

Vaatimusanalyysi ja -määrittely

Kartoitetaan ja dokumentoidaan **mitä asiakas haluaa**

- ▶ Ei vielä puututa siihen miten järjestelmä tulisi toteuttaa
- ▶ Ei oteta kantaa ohjelman sisäisiin teknisiin ratkaisuihin, ainoastaan siihen miten toiminta näkyy käyttäjälle
- ▶ Sovelluksen **toiminnalliset vaatimukset**
 - ▶ Miten ohjelman tulisi toimia?
 - ▶ Toimintaympäristön asettamat rajoitteet
 - ▶ Toteutusympäristö
 - ▶ Suorituskykyvaatimukset
 - ▶ Luotettavuusvaatimukset
 - ▶ Käytettävyys

Ohjelmistotuotantoprosessin vaiheet

Vaatimusanalyysi ja -määrittely

Esim: Yliopiston kurssinhallintajärjestelmä

- ▶ Toiminnallisia vaatimuksia:
 - ▶ Opetushallinto voi syöttää kurssin tiedot järjestelmään
 - ▶ Opiskelija voi ilmoittautua valitsemalleen kurssille
 - ▶ Opettaja voi syöttää opiskelijan suoritustiedot
 - ▶ Opettaja voi tulostaa kurssin tulokset
- ▶ Toimintaympäristön rajoitteita:
 - ▶ Kurssien tiedot talletetaan jo olemassa olevaan tietokantaan
 - ▶ Järjestelmää käytetään www-selaimella
 - ▶ Toteutus Javalla
 - ▶ Kyettävä käsittämään vähintään 100 ilmoittautumista minuutissa

Ohjelmistotuotantoprosessin vaiheet

Vaatimusanalyysi ja -määrittely

- ▶ Jotta toteuttajat ymmärtäisivät mitä pitää tehdä, joudutaan ongelma-alueutta analysoimaan
 - ▶ Esimerkiksi jäsennetään ongelma-alueen käsittelytöitä
 - ▶ Tehdään ongelma-alueesta malli eli yksinkertaistettu kuvaus
- ▶ Vaatimusmäärittelyn pääteeksi yleensä tuotetaan **määrittelydokumentti**
 - ▶ Kirjaa sen mitä ohjelman halutaan
 - ▶ Käytetään ohjeena suunnittelun ja toteutukseen
- ▶ Määrittelydokumentin sijaan määrittely (tai ainakin sen osa) voidaan myös ilmaista ns. hyväksymistesteinä. Tällöin ohjelma toimii "määritelmänsä mukaisesti" jos se läpäisee kaikki määritellyt hyväksymistestit

Ohjelmistotuotantoprosessin vaiheet

Ohjelmiston suunnittelu

Miten saadaan toteutettua määrittelydokumentissa vaaditulla tavalla toimiva ohjelma?

1. Arkkitehtuurisuunnittelu

- ▶ Ohjelman rakenne karkealla tasolla
- ▶ Mistä suuremmista rakennekomponenteista ohjelma koostuu?
- ▶ Miten komponentit yhdistetään, eli komponenttien väliset rajapinnat

2. Oliosuunnittelu

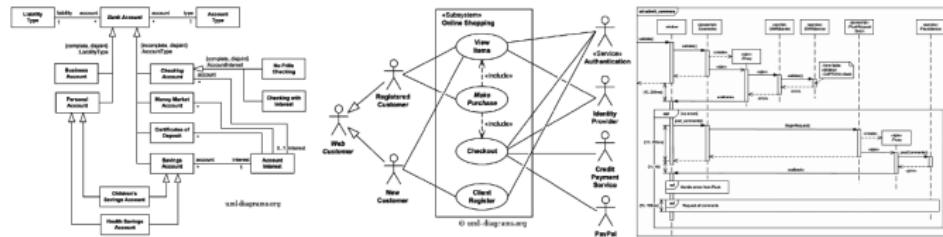
- ▶ yksittäisten komponenttien suunnittelu

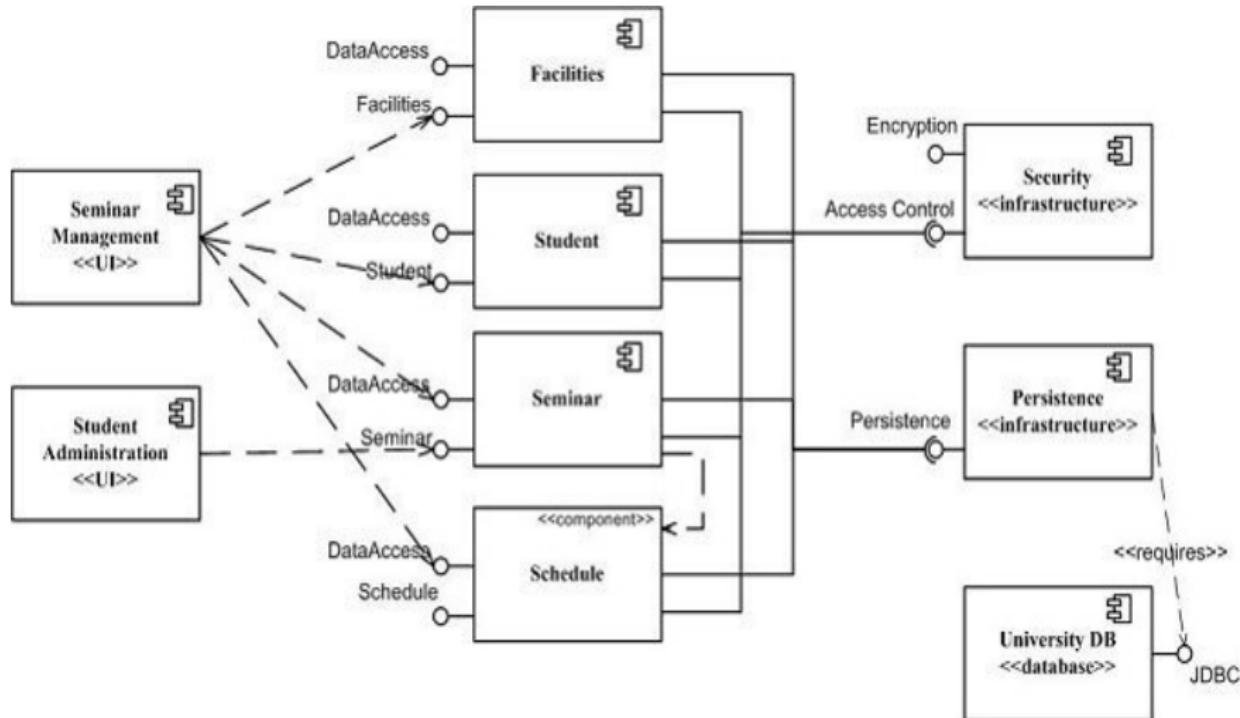
→ Lopputuloksena suunnitteludokumentti

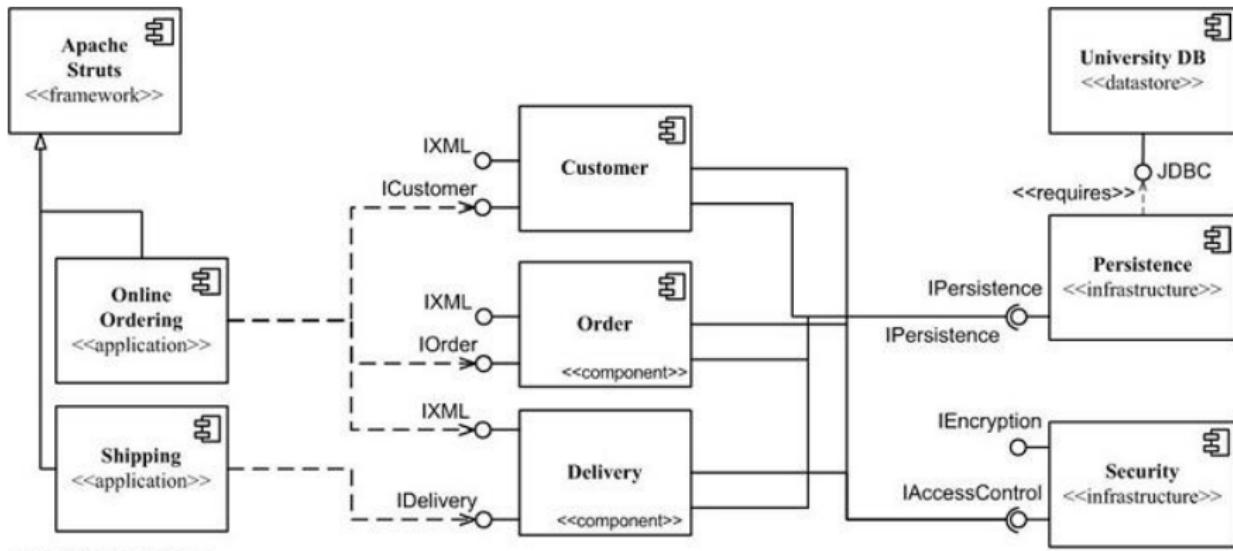
Ohjelmistotuotantoprosessin vaiheet

Ohjelmiston suunnittelu

- ▶ **Suunnitteludokumentti**
 - ▶ Ohje toteuttajille
 - ▶ Joskus/usein suunnittelu- ja ohjelmointivaihe ovat niin kiinteästi sidottuna toisiinsa, että tarkkaa suunnitteludokumenttia ei tehdä
 - ▶ Joskus koodi toimii dokumenttina
 - ▶ **Mallit liittyvät vahvasti suunnitteluun!**
 - ▶ Arkkitehtuurikuvaus
 - ▶ Järjestelmän alikomponentit
 - ▶ Komponenttien väliset rajapinnat
 - ▶ muista: suunnittelumallit (*design patterns*)







Copyright 2004 Scott W. Ambler

Ohjelmistotuotantoprosessin vaiheet

Toteutus, testaus ja ylläpito

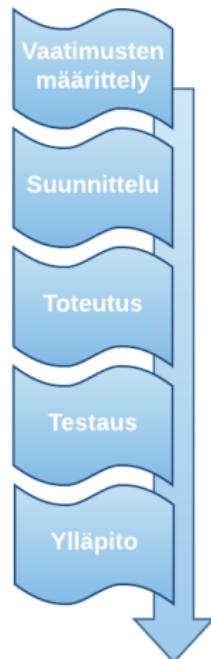
- ▶ Suunnittelun mukainen järjestelmä toteutetaan valittuja tekniikoita käyttäen
- ▶ Toteutuksen yhteydessä ja jälkeen testataan:
 - ▶ **Yksikkötestaus**
 - ▶ Toimivatko yksittäiset metodit ja luokat?
 - ▶ **Integraatiotestaus**
 - ▶ Varmistetaan komponenttien yhteentoimivuus
 - ▶ **Järjestelmätestaus**
 - ▶ Toimiiko kokonaisuus niin kuin vaatimusdokumentissa sanotaan (huom: hyväksymistestaus)?
- ▶ Valmiissakin järjestelmässä virheitä ja tarvetta laajennuksiin
- ▶ **Tämän kurssin painopiste on vaatimusmäärittelyssä ja suunnittelussa, mutta osin myös testaamisessa**

Ohjelmistotuotantoprosessi käytännössä

Vesiputousmalli

Vesiputousmalli on vaiheellinen ohjelmistotuotantoprosessi, jossa suunnittelu- ja toteutusprosessi etenee vaiheesta alas päin kuin vesiputouksessa.

- ▶ **Perinteinen tapa tehdä ohjelmistoja**
- ▶ Tuotantoprosessin vaiheet etenevät peräkkäin
- ▶ Vaiheen valmistuttua seuraavaan vaiheeseen
- ▶ Jokaisen vaiheen lopputulos dokumentoidaan tyypillisesti erittäin tarkasti



Ohjelmistotuotantoprosessi käytännössä

Vesiputousmalli

- ▶ **Vesiputousmallissa kuitenkin ongelmia:**
 - ▶ Järjestelmää testataan kokonaisuudessaan vasta kun “kaikki” on valmiina
 - ▶ Suunnitteluvaiheen virheet saattavat paljastua vasta testauksessa
 - ▶ Perustuu oletukselle, että käyttäjä pystyy määrittelemään ohjelman halutun toiminnallisuuden heti projektin alussa
 - ▶ Näin ei usein kuitenkaan tapahdu, asiakas ei osaa sanoa mitä haluaa, lisäksi projekti voi kestää pitkään, jolloin vaatimuksien muutokset ovat todennäköisempiä

Ohjelmistotuotannon yksi perustavanlaatuisimmista ongelmista on asiakkaan ja toteuttajien välinen kommunikointi!

Ohjelmistotuotantoprosessi käytännössä

Ketterä ohjelmistokehitys

- ▶ Lähdetään olettamuksesta, että asiakkaan vaatimukset muuttuvat ja tarkentuvat projektin kuluessa
 - ▶ Ei siis yritetäkään kirjoittaa alussa määrittelydokumenttia, jossa kirjattuna tyhjentävästi järjestelmältä haluttu toiminnallisuus
- ▶ Tuotetaan järjestelmä **iteratiivisesti**, eli pienissä paloissa!
 - ▶ Ensimmäisen iteraation aikana tuotetaan pieni osa järjestelmän toiminnallisuutta
 - ▶ määritellään vähän, suunnitellaan vähän ja toteutetaan ne
 - ▶ Lyhyitä iteraatioita - tyypillisesti muutaman viikon
 - ▶ Asiakas antaa palautetta iteraation päättäeksi → Korjausliike!
 - ▶ Seuraavassa iteraatiossa toteutetaan taas hiukan uutta toiminnallisuutta asiakkaan toiveiden mukaan

Jokaisen iteraation päättäeksi lopputuloksena ohjelmisto, jossa on mukana toiminnallisuutta.

Ohjelmistotuotantoprosessi käytännössä

Ketterä ohjelmistokehitys

Asiakkaan kanssa jatkuva komunikaatio on ketteryyden suola.

- ▶ Asiakkaan palaute on välitön
 - ▶ Vaatimuksia voidaan tarkentaa ja muuttaa
- ▶ Asiakas valitsee jokaisen iteraation aikana toteutettavat lisäominaisuudet
- ▶ → Todennäköisempää että aikaansaannos toiveiden mukainen
- ▶ Itraation sisällä määrittely, suunnittelu, toteutus ja testaus eivät välttämättä etene peräkkäin (usein jatkuvaa!)
 - ▶ Ketterissä menetelmissä dokumentoinnin rooli on yleensä kevyempi kun vesiputoosmallissa

Ohjelmistotuotantoprosessi käytännössä

Ketterä ohjelmistokehitys

Virheellinen johtopäätös on ajatella, että kaikki ei-perinteinen tapa tuottaa ohjelmistoja on ketterien menetelmien mukainen!

- ▶ Häkkerointi siis ei ole ketterä menetelmä!
- ▶ Ketteryyys ei tarkoita ettei ole sääntöjä!
- ▶ Monissa ketterissä menetelmissä (kuten eXtreme Programming) on päinvastoin erittäin tarkasti määritelty miten ohjelmien laatua hallitaan
- ▶ Pariohjelmointi, jatkuva integraatio, automatisoitu testaus, Testaus ensin -lähestymistapa (TDD), ...
- ▶ Eli myös ketteryys (voi) vaatii kurinalaisuutta, joskus jopa enemmän kuin perinteinen vesiputousmalli

Mallintaminen

Eli miin kurssilla keskitytää?

Mallintaminen

Eli mihin kurssilla keskitytään?

Malli on abstrakti kuvaus mielenkiinnon alla olevasta kohteesta

- ▶ Perinteiset insinöörialat perustuvat malleihin
 - ▶ Esim. siltaa rakentaessa tarkat lujuuslaskelmat (=malli)
 - ▶ Näihin perustuen tehdään piirustukset, eli malli siitä miten silta pitää toteuttaa (=edellistä hieman tarkempi malli)
- ▶ Kuvaaa olennaisen ja VAIN olennaisen
- ▶ Käyttötarkoitusta varten liian tarkat tai liian ylimalkaiset mallit epäoptimaalisia
- ▶ Mitä on olennaista, riippuu mallin käyttötarkoituksesta
 - ▶ Metron linjakartta on hyvä malli julkisen liikenteen käyttäjälle
 - ▶ Autoilija taas tarvitsee tarkemman mallin eli tiekartan
 - ▶ Helsingin keskustassa kävelijä taas tarvitsee tarkempaa kartta

Mallintaminen

Mallin näkökulma ja abstraktiotaso

- ▶ Mallien **abstraktiotaso** vaihtelee:
 - ▶ Abstraktimpi malli käyttää korkeamman tason käsitteitä
 - ▶ Konkreettisempi malli taas on yksityiskohtaisempi ja käyttää "matalamman" tason käsitteitä ja kuvaaa kohdetta tarkemmin
- ▶ Mallien **näkökulma** vaihtelee:
 - ▶ Jos kaikki yritetään mahduttaa samaan malliin, ei lopputulos ole selkeää
 - ▶ Malli kuvaaa usein korostetusti tiettyä näkökulmaa
 - ▶ Eri näkökulmat yhdistämällä saadaan idea kokonaisuudesta
- ▶ Käytännössä siis asunnon ostaja tarvitsee *pohjapiirrustuksen* ja sähköasentaja *sähkösuunnitelman* (näkökulmia talosta)

Oikea malli oikeaan tarkoitukseen!

Mallintaminen

Ohjelmistojen mallintaminen

Entä ohjelmistojen mallintaminen?

- ▶ Vaatimusdokumentissa mallinnetaan mitä järjestelmän toiminnallisuudelta halutaan
- ▶ Suunnitteludokumentissa mallinnetaan...
 - ▶ Järjestelmän arkkitehtuuri eli jakautuminen tarkempiin komponentteihin
 - ▶ Yksittäisten komponenttien toiminta
- ▶ Toteuttaja käyttää näitä malleja ja luo konkreettisen tuotteen
- ▶ Vaatimuksien mallit yleensä korkeammalla abstraktiotasolla kuin suunnitelman mallit
 - ▶ Vaatimus ei puhu ohelman sisäisestä rakenteesta toisin kuin suunnitelma

Mallintaminen

Ohjelmistojen mallintaminen

- ▶ Myös ohjelmistojen malleilla on erilaisia näkökulmia
- ▶ Jotkut mallit kuvaavat rakennetta...
 - ▶ Mitä komponentteja järjestelmässä on?
- ▶ Jotkut taas toimintaa...
 - ▶ Miten komponentit kommunikoivat?

Eri näkökulmat yhdistämällä saadaan idea kokonaisuudesta

Mallintaminen

Mallinnuksen kaksi suuntaa

- ▶ Usein mallit toimivat apuna kun ollaan rakentamassa jotain uutta, eli
 - ▶ Ensin tehdään malli, sitten rakennetaan esim. silta
- ▶ Toisaalta esim. fyysikot tutkivat erilaisia fyysisen maailman ilmiöitä rakentamalla niistä malleja ymmärryksen helpottamiseksi
 - ▶ Ensin on olemassa jotain todellista josta sitten luodaan malli
- ▶ Ohjelmistojen mallinnuksessa myös olemassa nämä **kaksi mallinnussuuntaa**
 - ▶ Ohje toteuttamiselle: malli → ohjelma
 - ▶ Apuna asiakkaan ymmärtämiseen: ohjelma → malli
 - ▶ ns. takaisinmallinnus

Ohjelmistojen mallinnuskäytännöt

Oliomallinnus ja UML

Ohjelmistojen mallinnuskäytännöt

Oliomallinnus ja UML

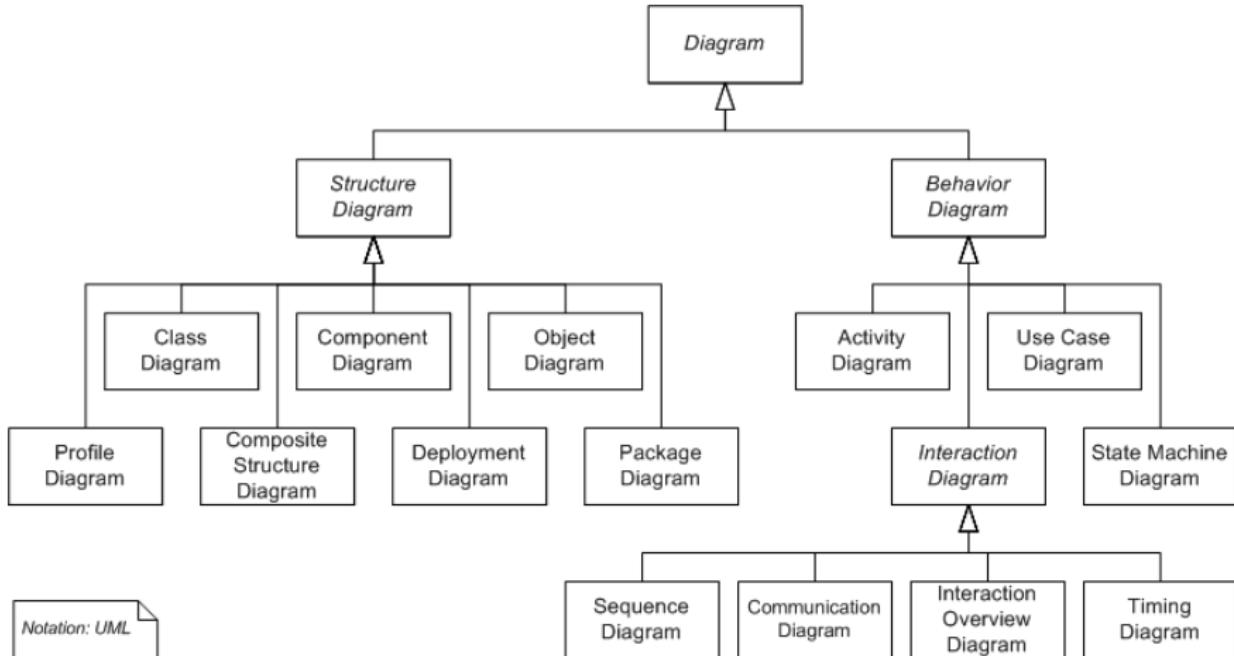
Miten mallintaa ohjelmistoja?

- ▶ Pitkään tilanne oli sekava ja on sitä osin edelleen
- ▶ Suosituimmaksi tavaksi on noussut **oliomallinnus**
- ▶ *Minkä tahansa järjestelmän katsotaan voivan muodostua olioista, jotka yhteistyössä toimien ja toistensa palveluja hyödyntäen tuottavat järjestelmän tarjoamat palvelut*
- ▶ Eli järjestelmä tarjoaa joukon palveluja, osa-järjestelmiä, jotka toteuttavat asiakkaan vaatimuksia/toiminnallisuuksia
- ▶ Mallinnetaan siis niitä olioita!

Ohjelmistojen mallinnuskäytännöt

Oliomallinnus ja UML

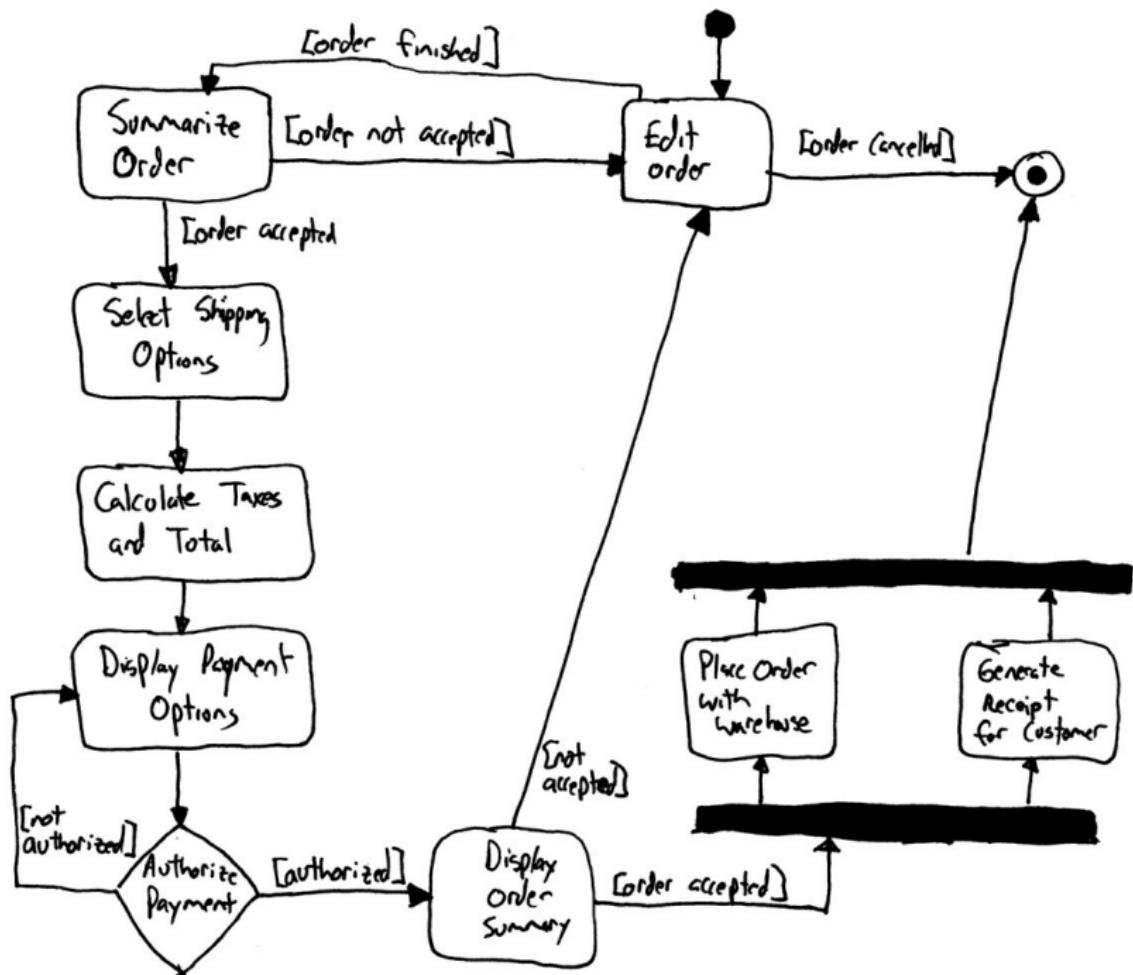
- ▶ Unified Modelling Language eli UML
- ▶ Oliomallinnusta varten kehitetty standardoitu kuvaustekniikka
 - ▶ 1990 luvulta, nykyinen versio 2.5 (vuonna 2015)
- ▶ UML:ssä nykyään 13 esityyppistä kaaviota
 - ▶ Eri näkökulmille omat kaavionsa
- ▶ UML standardi ei määrittele miten ja missä tilanteissa kaavioita tulisi käyttää
 - ▶ Tätä varten olemassa useita oliomenetelmiä



Ohjelmistojen mallinnuskäytännöt

UML:n käyttötapa

- ▶ UML-standardi määrittelee kaavioiden syntaksin eli oikeaoppisen piirtotavan suhteellisen tarkasti
 - ▶ Eri versioiden välillä pieniä muutoksia
 - ▶ Vanhojen standardien mukaisia kaavioita näkyy yhä
- ▶ Jotkut suosivat UML:n käytöä tarkasti syntaksia noudattaen
 - ▶ Kaaviot piirretään tällöin usein tietokoneavusteisella suunnitteluvälineellä
 - ▶ Nykyään tämä suuntaus alkaa olla jo melko harvinaista
- ▶ On myös UML:n luonnosmaisemman käytön puolestapuhujia
 - ▶ ns. ketterä mallinnus
 - ▶ Kaaviot ennenkaikcia kommunikoinnin apuväline
 - ▶ Tärkeimmät kuvat (ehkä) siirretään sähköiseen muotoon
 - ▶ Digikuva tai uudelleenpiirto editorilla



Käyttötapausmalli

Vaatimusanalyysi ja -määrittely

Käyttötapausmalli

Vaatimusanalyysi ja -määrittely

Sovelluksen vaatimukset:

- ▶ **Toiminnalliset vaatimukset**
 - ▶ Miten ohjelmsito vaikuttaa ympäristöönsä, eli mitä toimintoja ohjelmassa on?
 - ▶ *Opetushallinto voi syöttää kurssin tiedot järjestelmään*
 - ▶ *Opiskelija voi ilmoittautua valitsemalleen kurssille*
 - ▶ *Opettaja voi syöttää opiskelijan suoritustiedot*
- ▶ **Ei-toiminnalliset vaatimukset** (eli ympäristön rajoitteet)
 - ▶ Miten ohjelmiston tulee täyttää toiminnalliset vaatimukset
 - ▶ Toteutusympäristö, suorituskykyvaatimukset, ...

Ennenkuin voidaan toteuttaa ohjelmisto, joka ratkaisee ongelman on tiedettävä mikä ratkaistava ongelma on!

Käyttötapausmalli

Vaatimusanalyysi ja -määrittely

- ▶ **Vaatimusmäärittelyssä** ei oteta kantaa ohjelman sisäisiin teknisiin ratkaisuihin, ainoastaan siihen miten toiminta näkyy käyttäjälle
- ▶ Miten toiminnalliset vaatimukset tulisi ilmaista?
- ▶ Ratkaisuna **käyttötapausmallit**:
 - ▶ Tapa ohjelman toiminnallisten vaatimusten ilmaisemiseen
 - ▶ Ei-toiminnallisten vaatimusten ilmaisemiseen käyttötapausmalli ei juuri ota kantaa, vaan ne on ilmaistava muuten
 - ▶ On olemassa muitakin tapoja toiminnallisten vaatimusten ilmaisuun esim. kurssilla Ohjelmistotuotanto esiteltävät User storyt eli käyttäjätarintat

Käyttötapausmalli

Vaatimusanalyysi ja -määrittely

- ▶ Ohjelmisto tarjoaa käyttäjälle palveluita
 - ▶ Ohjelmiston toiminta voidaan kuvata määrittelemällä sen tarjoamat palvelut
- ▶ Palveluilla on käyttäjä
 - ▶ Henkilö, toinen järjestelmä, laite yms. taho, joka on järjestelmän ulkopuolella, mutta tekemisissä järjestelmän kanssa
 - ▶ Järjestelmän tiedon hyväksikäyttäjä tai tietojen lähde

Käyttötapausmalli

Käyttäjien tunnistaminen

Hyvä tapa aloittaa vaatimusmäärittely on tunnistaa/etsiä rakennettavan järjestelmän käyttäjät.

- ▶ Kysymyksiä jotka auttavat:
 - ▶ Kuka/mikä saa tulosteita järjestelmästä?
 - ▶ Kuka/mikä toimittaa tietoa järjestelmään?
 - ▶ Kuka käyttää järjestelmää?
 - ▶ Mihin muihin järjestelmiin kehitettävä järjestelmä on yhteydessä?
- ▶ Käyttäjä on oikeastaan **rooli**
 - ▶ Missä roolissa toimitaan järjestelmän suhteeseen
 - ▶ Yksi ihminen voi toimia monessa roolissa...

Käyttötapausmalli

Käyttäjien tunnistaminen

TKTL:n kurssi-ilmoittautumisjärjestelmä

- ▶ Käyttäjärooleja
 - ▶ Opiskelija
 - ▶ Opettaja
 - ▶ Opetushallinto
 - ▶ Suunnittelija
 - ▶ Laitoksen johtoryhmä
 - ▶ Tilahallintojärjestelmä
 - ▶ Henkilöstöhallintajärjestelmä

Osa käyttäjistä yhteydessä järjestelmään vain epäsuorasti.

- ▶ Osa “*käyttäjistä*” on muita järjestelmiä
 - ▶ Sana käyttäjä ei ole terminä tässä tilanteessa paras mahdollinen
 - ▶ Englanninkielinen termi **actor** onkin hieman suomenkielistä termiä kuvaavampi

Käyttötapausmalli

Käyttötapaus

- ▶ **Käyttötapaus** (engl. use case) kuvaaa käyttäjän ohelman avulla suorittaman tehtävän.
 - ▶ *Miten käyttäjä kommunikoi järjestelmän kanssa tietystä käyttötilanteessa?*
- ▶ Käyttötilanteet liittyvät käyttäjän **tarpeeseen** tehdä järjestelmällä jotain
 - ▶ Kurssi-ilmoittautumisjärjestelmä: Opiskelijan ilmoittautuminen
 - ▶ Mitä vuorovaikutusta käyttäjän ja järjestelmän välillä tapahtuu kun opiskelija ilmoittautuu kurssille?
- ▶ Yksi käyttötapaus on looginen, “isompi” kokonaisuus
 - ▶ Käyttötapauksella lähtökohta
 - ▶ Ja merkityksen omaava lopputulos
 - ▶ Eli pienet asiat, kuten “syötä salasana” eivät ole käyttötapauksia
 - ▶ Kyseessä pikemminkin yksittäinen operaatio, joka voi sisältyä käyttötapaukseen

Käyttötapausmalli

Käyttötapauksen kuvaaminen

- ▶ **Kuvataan tekstinä**
- ▶ Ei ole olemassa täysin vakiintunutta tapaa kuvaukseen (esim. UML ei ota asiaan kantaa)
- ▶ Kuvaussessä mukana usein tietyt osat:
 - ▶ Käyttötapauksen nimi
 - ▶ Käyttäjät
 - ▶ Laukaisija
 - ▶ Esiehdo
 - ▶ Jälkiehdo
 - ▶ Käyttötapauksen kulku
 - ▶ Poikkeuksellinen toiminta

- *Käyttäjä*: opiskelija
- *Tavoite*: saada kurssipaikka
- *Laukaisija*: opiskelijan tarve
- *Esiehto*: opiskelija on ilmoittautunut kuluvalta 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. Opiskelija 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*:
 - 4a. Opiskelija ei voi valita ryhmää, joka on täynnä
 - 6a. Opiskelija ei voi ilmoittautua, jos hänen on kirjattu osallistumisesta.

Käyttötapausmalli

Käyttötapauksen kuvaaminen

► Esiehsto

- ▶ Asioiden tila joka on vallittava, jotta käyttötapaus pystyy käynnistymään

► Jälkiehsto

- ▶ Kuva mikä on tilanne käyttötapauksen onnistuneen suorituksen jälkeen

► Laukuaisija

- ▶ Mikä aiheuttaa käyttötapauksen käynnistymisen, voi olla myös ajan kuluminen

► Käyttötapauksen kulku

- ▶ Kuva onnistuneen suorituksen, usein edellisen sivun tapaan käyttäjän ja koneen välisenä dialogina

► Poikkeuksellinen toimita

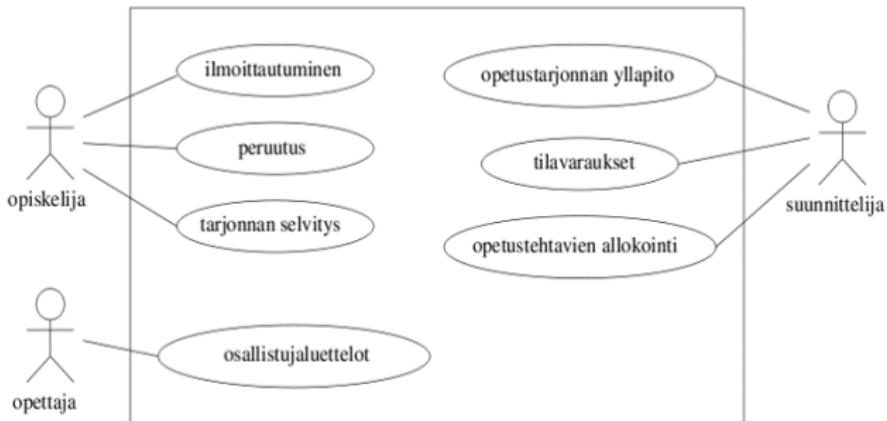
- ▶ Mitä tapahtuu jos tapahtumat eivät etene onnistuneen suorituksen kuvauksen mukaan
- ▶ Viittaa onnistuneen suorituksen dialogin numeroihin, esim. jos kohdassa 4 voi tapahtua poikkeus normaaliin kulkuun, kuvataan se askeleena 4a

- *Käyttäjä*: opiskelija
- *Tavoite*: perua ilmoittautuminen, välttää sanktiot
- *Laukaisija*: opiskelijan tarve poistaa ilmoittautuminen
- *Esiehto*: opiskelija on ilmoittautunut tietylle kurssille
- *Jälkiehto*: opiskelijan ilmoittautuminen kurssille on poistettu
- *Käyttötapauksen kulku*:
 1. Opiskelija valitsee toiminnon "omat ilmoittautumiset"
 2. Järjestelmä pyytää opiskelijaa tunnistautumaan
 3. Opiskelija tunnistautuu
 4. Järjestelmä näyttää opiskelijan ilmoittautumiset
 5. Opiskeljia valitsee tietyn ilmoittatumisensa ja peruu sen
 6. Järjestelmä ilmoittaa opiskelijalle ilmoittautumisen peruuntumisesta

Käyttötapausmalli

Käyttötapauskaavio

- ▶ UML:n käyttötapauskaavion avulla voidaan kuvata käyttötapausten ja käyttäjien (engl. actor) keskinäisiä suhteita
- ▶ Kurssi-ilmoittautumisjärjestelmän “korkean tason” käyttötapauskaavio:



Käyttötapausmalli

Käyttötapauskaavio

- ▶ Käyttäjät kuvataan tikku-ukkoina
 - ▶ Olemassa myös vaihtoehtoinen symboli, joka esitellään pian
- ▶ Käyttötapaukset taas kuvataan järjestelmää kuvaavan nelilön sisällä olevina ellipseinä
 - ▶ Ellipsin sisällä käyttötapauksen nimi
- ▶ Käyttötapausellipsiin yhdistetään viivalla kaikki sen käyttäjät
 - ▶ Kuvaan ei siis piirretä nuolia!

HUOM: Käyttötapauskaaviossa ei kuvata mitään järjestelmän sisäisestä rakenteesta

- ▶ Esim. vaikka tiedettäisiin että järjestelmä sisältää tietokannan, ei sitä tule kuvata käyttötapausmallissa

Käyttötapausmalli

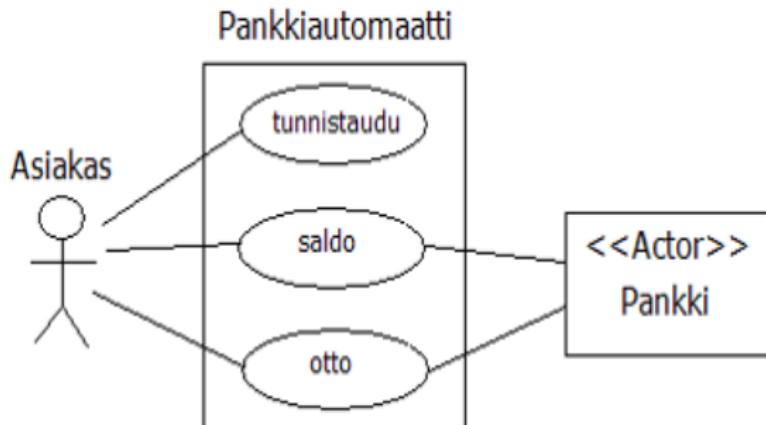
Käyttötapauskaavion käyttö

- ▶ Kaaviossa siis käyttötapauksista ainoastaan nimi
 - ▶ Käyttötapauksen sisältö kuvataan aina tekstuaalisena esityksenä
- ▶ Kaavio tarjoaa hyvän yleiskuvan järjestelmän käyttäjistä ja palveluista
 - ▶ Määrittelydokumentin alussa kannattaakin olla käyttötapauskaavio ”sisällysluettelona”
- ▶ Jokainen käyttötapaus tulee sitten kirjata tekstuaalisesti tarvittavalla tarkkuudella
 - ▶ Ei siis ole olemassa standardoitua tapaa käyttötapauksen kirjaamiseen
 - ▶ Ohjelmistoprojektissa tulee kuitenkin määritellä käyttötapauspohja, eli sopia joku yhteinen muoto, jota kaikkien käyttötapausten dokumentoinnissa noudatetaan

Käyttötapausmalli

Toinen esimerkki: pankkiautomaatin käyttötapaukset

- ▶ Käyttötapaukset ovat tunnistaudu, saldo ja otto
- ▶ Käyttötapausten käyttäjät eli toimintaan osallistuvat tahot ovat Asiakas ja Pankki
 - ▶ Alla on esitely tikku-ukolle vaihtoehtoinen tapa merkitä käyttäjää eli laatikko, jossa merkintä «actor»



Käyttötapaus 1: otto

Tavoite Asiakas nostaa tililtään haluamansa määrän rahaa

Käyttäjät Asiakas, Pankki

Esiehto Kortti syötetty ja asiakas tunnistautunut

Jälkiehto käyttäjä saa tililtään haluamansa määrän rahaa.
Jos saldo ei riitä, tiliä ei veloiteta

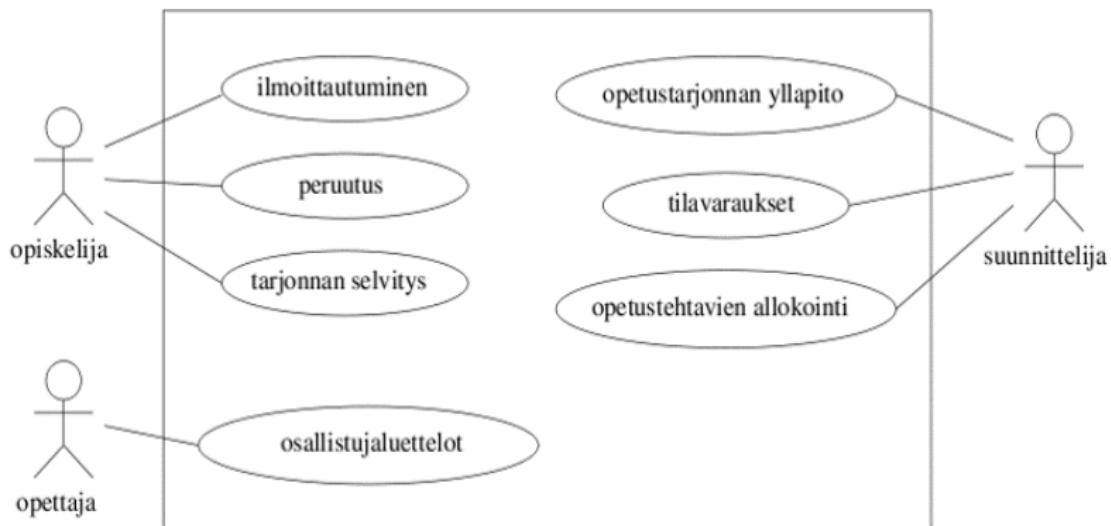
Käyttötapauksen kulku:

1. asiakas valitsee otto-toiminnon
2. automaatti kysyy nostettavaa summaa
3. asiakas syöttää haluamansa summan
4. pankilta tarkistetaan riittääkö asiakkaan saldo
5. summa veloitetaan asiakkaan tililtä
6. kuitti tulostetaan ja annetaan asiakkaalle
7. rahat annetaan asiakkaalle
8. pankkikortti palautetaan asiakkaalle

Käyttötapauksen kulku:

- 4a** asiakkaan tilillä ei tarpeeksi rahaa, palautetaan kortti

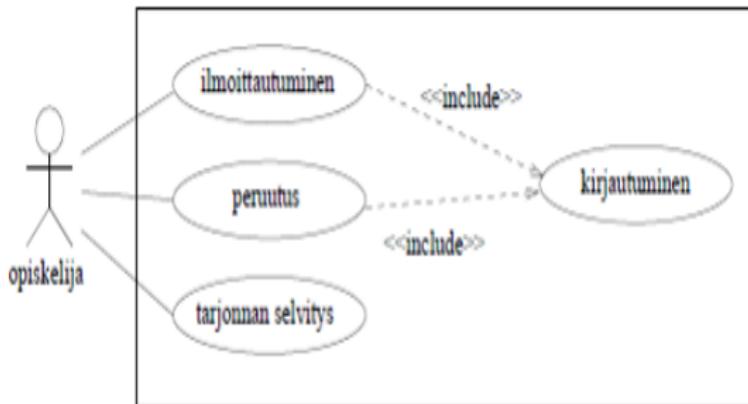
Palataan takaisin kurssi-ilmoittautumisjärjestelmään...



Käyttötapausmalli

Yhteiset osat

- ▶ Moneen käyttötapaukseen saattaa liittyä yhteinen osa
- ▶ Yhteisestä osasta voidaan tehdä "alikäyttötapaus", joka sisällytetään (include) pääkäyttötapaukseen
- ▶ Käyttötapauskaaviossa tätä varten merkintä «include»
 - ▶ katkoviivanuoli pääkäyttötapauksesta apukäyttötapaukseen
- ▶ Esim. käyttötapaus kirjautuminen suoritetaan aina kun tehdään ilmoittautuminen tai peruutus

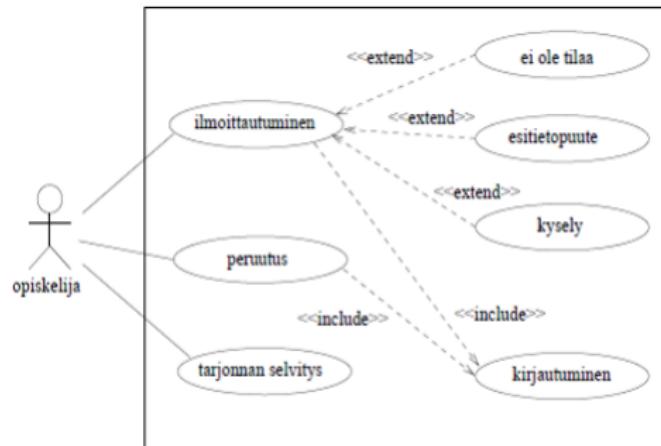


- *Käyttäjä*: opiskelija
- *Tavoite*: saada kurssipaikka
- *Laukaisija*: opiskelijan tarve
- *Esiehto*: opiskelija on ilmoittautunut kuluvalta lukukaudella läsnäolevaksi
- *Jälkiehto*: opiskelija on lisätty haluamansa ryhmän ilmoittautujien listalle
- *Käyttötapaksen kulku*:
 1. Opiskelija aloittaa kurssi-ilmoittautumistoiminnon
 2. Järjestelmä näyttää kurssitarjonnan
 3. Opiskelija tutkii kurssitarjontaa
 4. Opiskeljia valitsee ohjelmiston esittämästä tarjonnasta kurssin ja ryhmän
 5. **Suoritetaan käyttötapaus kirjautuminen**
 6. Järjestelmä ilmoittaa opiskelijalle ilmoittautumisen onnistumisesta.
- *Poikkeuksellinen toiminta*:

Käyttötapausmalli

Poikkeustilanteet ja laajennukset

- ▶ Sisällytettävä (eli **include**) käyttötapaus suoritetaan aina pääkäyttötapaksen suorituksen yhteydessä
- ▶ Myös tarvittaessa suoritettava laajennus tai poikkeustilanne voidaan kuvata apukäyttötapaksena, joka laajentaa (**extend**) pääkäyttötapausta
 - ▶ Laajennus suoritetaan siis vaan tarvittaessa
- ▶ Esim. Ilmoittautuessa saatetaan huomata esitetopuute, jonka käsiteily on oma käyttötapaaksensa



Käyttötapausmalli

Poikkeustilanteet ja laajennukset

Huomaa, että laajennuksessa nuolensuunta on apukäyttötapauksesta pääkäyttötapaukseen päin (toisin kuin sisällytyksessä)

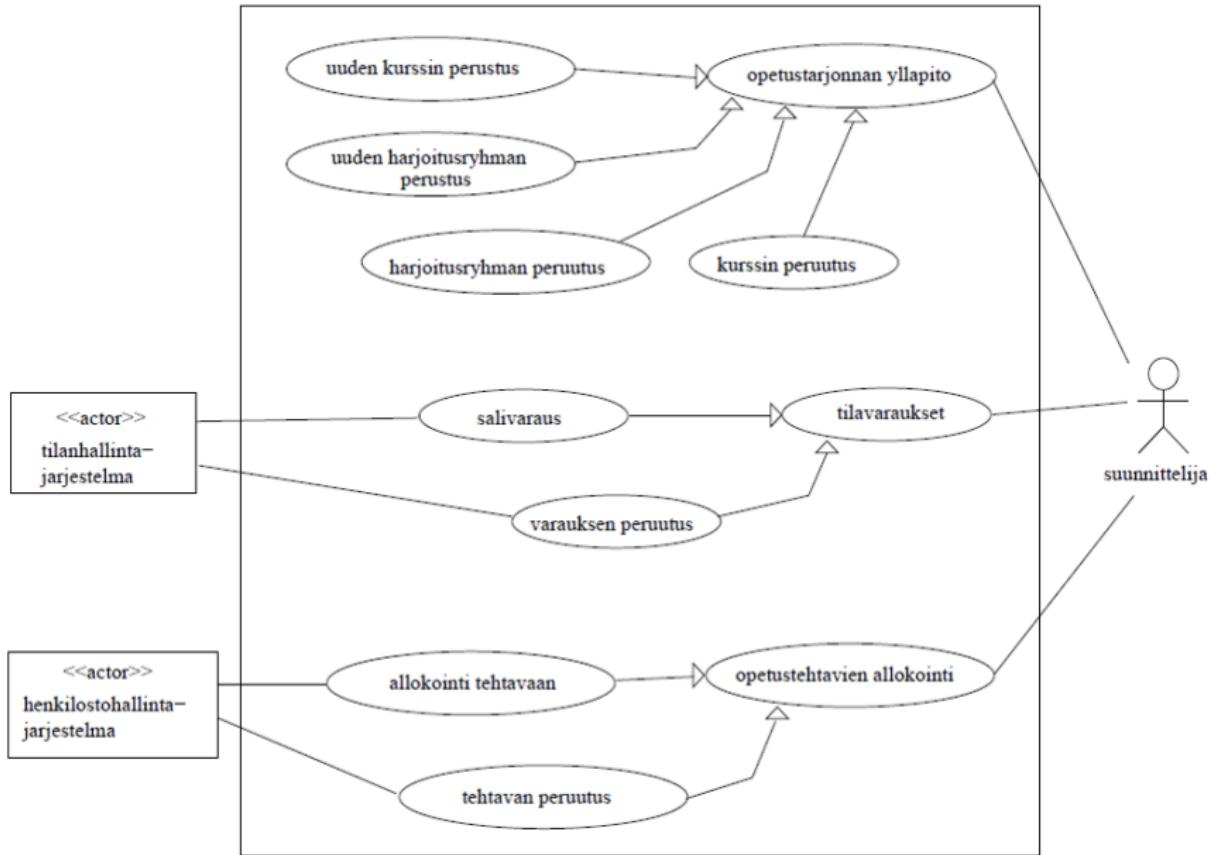
- ▶ Laajennus tulee ehdottomasti merkitä käyttötapaksen tekstualueen kuvaukseen
- ▶ Edellisen dian laajennusesimerkki ei ole erityisen onnistunut
 - ▶ Laajennuksienkin pitäisi olla kunnollisia käyttötapauksia (eli asioita joilla on selkeä tavoite), ei metodikutsumaisia kyselyjä tai ilmoituksia (kuten ei tilaa - tai esitietopuute-ilmoitus)
- ▶ Poikkeustilanteet on parempi kuvata tekstualisessa esityksessä ja jättää ne kokonaan pois käyttötapauskaavioista
- ▶ Koko laajennuskäsitteen tarve käyttötapauskaavioissa on hieman kyseenalainen

Käyttötapausmalli

Yleistetty ja erikoistettu käyttötapaus

- ▶ Suunnittelijan käyttötapauksista erityisesti opetustarjonnasta ylläpito on hyvin laaja tehtäväkokonaisuus
- ▶ Voi aankin ajatella, että kyseessä on *yleistetty käyttötapaus*, joka oikeasti pitääkin sisällään useita konkreettisia käyttötapauksia, kuten
- ▶ Esim. voidaan ajatella, että kurssin peruutus on osa yleisettyä käyttötapausta, jota voimme kutsua opetustarjonnasta ylläpidoksi
- ▶ *Erikoistettu käyttötapaus*, käyttötapaus, joka sisältää laajennuskohtia, joihin toinen käyttötapaus voidaan sijoittaa, merkitään nuolella, jossa on katkoviiva ja avainsana <<extends>>, mutta tästä harvemmin käytetään

Yleisettyt käyttötapaukset merkitään nuolella, jossa on nuolenkärki osoittamaan siihen suuntaan, johon käyttötapaus on yleistetty



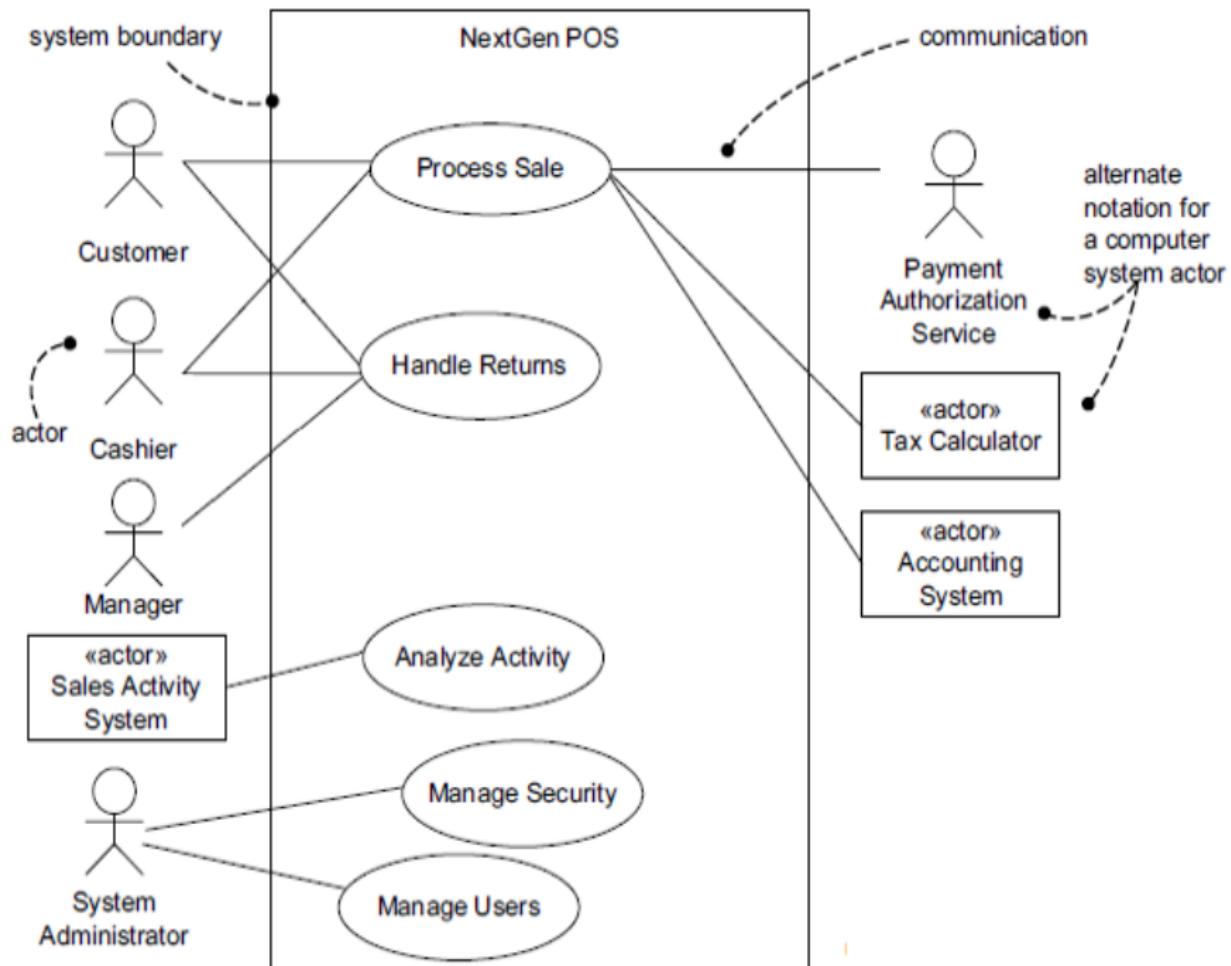
Käyttötapausmalli

Yleistetty ja erikoistettu käyttötapaus

- ▶ Craig Larmanin kirjasta *Applying UML and Patterns*²
- ▶ Aluksi etsitään järjestelmän käyttäjät
- ▶ Mietitään käyttäjien tavoitteita: mitä käyttäjä haluaa saada järjestelmällä tehtyä
- ▶ Käyttäjän tavoitteellisista toiminnoista (esim. käsitteli ostos) tulee tyypillisesti käyttötapauksia
- ▶ Samalla saatetaan löytää uusia käyttäjiä (erityisesti ulkoisia järjestelmiä joihin järjestelmä yhteydessä)
- ▶ Hahmotellaan alustava käyttötapausdiagrammi →

²Kirjan käyttötapausluku löytyy verkosta:

<http://www.craiglarman.com/wiki/index.php?title=Articles>



Käyttötapausmalli

Yleistetty ja erikoistettu käyttötapaus

- ▶ Otetaan aluksi tarkasteluun järjestelmän toiminnan kannalta kriittisimmät käyttötapaukset
- ▶ Ensin kannattanee tehdä vapaamuotoinen kuvaus käyttötapauksista ("brief use case")

Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

Kuva: POS = point of sales terminal eli kassapääte

- ▶ Tarkempi käyttötapaus kirjoitetaan projektin sopiman käyttötapauspohjan määräämässä muodossa

Use Case UC1: Process Sale

Primary Actor: Cashier

Preconditions: Cashier is identified and authenticated.

Success Guarantee (Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.

Main Success Scenario (or Basic Flow):

1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total.
Price calculated from a set of price rules.
Cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

***a. At any time, System fails:**

To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.

1. Cashier restarts System, logs in, and requests recovery of prior state.

2. System reconstructs prior state.

2a. System detects anomalies preventing recovery:

1. System signals error to the Cashier, records the error, and enters a clean state.

2. Cashier starts a new sale.

3a. Invalid identifier:

1. System signals error and rejects entry.

3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):

1. Cashier can enter item category identifier and the quantity.

3-6a: Customer asks Cashier to remove an item from the purchase:

1. Cashier enters item identifier for removal from sale.

2. System displays updated running total.

3-6b. Customer tells Cashier to cancel sale:

1. Cashier cancels sale on System.

3-6c. Cashier suspends the sale:

1. System records sale so that it is available for retrieval on any POS terminal.

7a. Paying by cash:

1. Cashier enters the cash amount tendered.
2. System presents the balance due, and releases the cash drawer.
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.

7b. Paying by credit:

1. Customer enters their credit account information.
2. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.
 - 2a. System detects failure to collaborate with external system:
 1. System signals error to Cashier.
 2. Cashier asks Customer for alternate payment.
 3. System receives payment approval and signals approval to Cashier.
 - 3a. System receives payment denial:
 1. System signals denial to Cashier.
 2. Cashier asks Customer for alternate payment.
 4. System records the credit payment, which includes the payment approval.
 5. System presents credit payment signature input mechanism.
 6. Cashier asks Customer for a credit payment signature. Customer enters signature.

Käyttötapausmalli

Tarkkaan kuvattu käyttötapaus

- ▶ Esimerkin mallin mukaan käyttötapauksen pääkulku kannattaa kuvata tiiviisti
 - ▶ Eri askeleiden sisältöä voi tarvittaessa tarkentaa (askel 7)
- ▶ Huomioi tapa, miten poikkeusten ja laajennusten sijainti pääkulussa merkitään
 - ▶ 7a → laajentaa/tarkentaa pääkulun kohtaa 7

Osa jossa laajennukset, tarkennukset ja poikkeukset dokumentoidaan, on usein paljon pidempi kuin normaali kulku

- ▶ Koska kyse vaatimusmäärittelystä, kuvaus toteutetaan abstraktilla tasolla eli...
 - ▶ Ei oteta kantaa toteutusyksityiskohtiin
 - ▶ eikä käyttöliittymään
 - ▶ Esim. tunnistetaanko ostos viivakoodin perusteella...

Käyttötapausmalli

Yhteenvetö

- ▶ Käyttötapaukset ovat yksi tapa kuvata ohjelmiston toiminnallisia vaatimuksia
- ▶ **Käyttötapauksen tekstuaalinen esitys oleellinen**
- ▶ Ohjelmistoprojektissa pitää sopia yhteinen tapa (*käyttötapauspohja*) käyttötapausten tekstuaaliseen esitykseen
- ▶ Käyttötapauskaavion merkitys lähinnä yleiskuvan antaja
- ▶ Jos huomaat käyttäväsi paljon aikaa ”oikeaoppisen” käyttötapauskaavion piirtämiseen, ryhdy välittömästi tekemään jotakin hyödyllisempää (esim. käyttötapausten tekstuaalisia esityksiä)

Ohjelmistotekniikan menetelmät

Luento 2

Testaus ja oliokaaviot

Testaus

Testauistasot

- ▶ **Yksikkötestaus**
 - ▶ Toimivatko yksittäiset metodit ja luokat kuten halutaan?
- ▶ **Integraatiotestaus**
 - ▶ Toimivatko yksittäiset *moduulit* yhdessä halutulla tavalla?
- ▶ **Järjestelmä/hyväksymistestaus**
 - ▶ Toimiiko kokonaisuus niin kuin vaatimusdokumentissa sanotaan?
- ▶ **Regressiotestaus**
 - ▶ *regressio* ≈ palautuminen, taantuminen, takautuminen
 - ▶ järjestelmälle muutosten ja bugikorjausten jälkeen ajettavia testejä jotka varmistavat että muutokset eivät riko mitään
 - ▶ Regressiotestit koostuvat yleensä yksikkö-, integraatio- ja järjestelmä/hyväksymätesteistä

Nykyinen trendi on tehdä testeistä automaattisesti suoritettavia

Testaus

Yksikkötestit

```
public class LyyraKorttiTest {  
  
    private LyyraKortti kortti;  
  
    @Before  
    public void setUp() {  
        kortti = new LyyraKortti(10);  
    }  
  
    @Test  
    public void konstruktoriAsettaaSaldonOikein() {  
        assertEquals("Kortilla on rahaa 10.0 euroa", kortti.toString());  
    }  
  
    @Test  
    public void syoEdullisestiVahentaaSaldoaOikein() {  
        kortti.syoEdullisesti();  
        assertEquals("Kortilla on rahaa 7.5 euroa", kortti.toString());  
    }  
}
```

Testaus

Yksikkötestit

- ▶ Yksikkötesteillä testataan yksittäistä "asiaa", esim. toimiiko metodi oikein tiettytyylisellä syötteellä
- ▶ Yhtä kokonaisuutta (esim. luokkaa) testaavat testitapaukset sijoitetaan yhden testiluokan sisälle
- ▶ Hyvät testit ovat kattavat eli testaavat kaiken koodin monipuolisesti
- ▶ Normaalien syötteiden lisäksi on testeissä erityisen tärkeää tutkia testattavien metodien toimintaa virheellisillä syötteillä ja erilaisilla raja-arvoilla

| | | |
|---------|------------|---------------------------|
| $x = y$ | $x > y$ | $x \text{ instead of } y$ |
| $x < y$ | $x \neq y$ | ... |

Testaus

JUnit

- ▶ Testaamisen harjoittelun käytetään kurssilla **JUnit** kirjastoa
- ▶ Testitapaukset (metodit) annotoidaan @Test avainasanalla
- ▶ Ennen jokaista testiä suoritetaan @Before -metodi
- ▶ Jokaisen testin jälkeen suoritetaan @After
- ▶ Ennen jokaista testiluokkaa suoritetaan @BeforeClass -metodi
- ▶ Testiluokan suorituksen jälkeen ajetaan @AfterClass -metodi

Testausta oppii parhaiten tekemällä, joten jätetään yksikkötestaus seuraaviin laskuharjoituksiin ja siirrytään eteenpäin!

Oliot ja luokat

Jokainen olio kuuluu johonkin luokkaan

Oliot ja luokat

Jokainen olio kuuluu johonkin luokkaan

Olio...

- ▶ on ohjelman tai sen sovellusalueen kannalta *mielenkiintoinen* asia tai käsite, tai ohjelman osa
- ▶ yleensä yhdistää tietoa ja toiminnallisuutta
- ▶ omaa identiteetin, eli erottuu muista olioista omaksi yksilökseen
- ▶ kuuluu johonkin luokkaan

Minkä tahansa järjestelmän katsotaan voivan muodostua oliosta, jotka yhteistyössä toimien ja toistensa palveluja hyödyntäen tuottavat järjestelmän tarjoamat palvelut

Oliot ja luokat

Suunnittelu ja toteutusvaiheen oliot ja luokat

Henkilö-luokka:

```
public class Henkilo {  
    private String nimi;  
    private int ika;  
  
    public Henkilo(String n) {  
        nimi = n;  
    }  
  
    public void vanhene() {  
        ika++;  
    }  
}
```

Henkilö-olioita:

```
Henkilo arto =  
new Henkilo("Arto");  
  
Henkilo heikki =  
new Henkilo("Heikki");  
  
arto.vanhene();  
heikki.vanhene();
```

Oliot ja luokat

Suunnittelu ja toteutusvaiheen oliot ja luokat

- ▶ Luokka kuvaaa minkälaisia siihen kuuluvat oliot ovat tietosisällön ja toiminnallisuuden suhteesta
- ▶ Oliota ja luokkia ajatellaan usein ohjelmointitason käsitleinä
 - ▶ Ohjelma muodostuu olioista
 - ▶ Oliot elävät koneen muistissa
 - ▶ Ohjelman toiminnallisuus muodostuu olioiden toiminnallisuudesta
- ▶ Ohjelmiston suunnitteluvaiheessa suunnitellaan mistä oliosta ohjelma koostuu ja miten oliot kommunikoivat
 - ▶ Nämä oliot sitten toteutetaan ohjelmointikielellä toteutusvaiheessa

Mistä suunnitteluvaiheen oliot tulevat? Miten ne keksitään?

Oliot ja luokat

Vaatinusanalyysivaiheen oliot ja luokat

- ▶ Vaatinusmäärittelyn yhteydessä tehdään usein *vaatinusanalyysi*
 - ▶ Kartoitetaan ohjelmiston sovellusalueen (eli sovelluksen kohdealueen) kannalta **tärkeitä käsitteitä ja niiden suhteita**
- ▶ Mietitään mitä tärkeitä asioita sovellusalueella on olemassa
 - ▶ Esim. kurssihallintojärjestelmän käsitteitä ovat...
 - ▶ Kurssi
 - ▶ Laskariryhmä
 - ▶ Ilmoittautuminen
 - ▶ Opettaja
 - ▶ Opiskelija
 - ▶ Sali
 - ▶ Salivaraus
 - ▶ Nämä käsitteet voidaan ajatella luokkina!

Oliot ja luokat

Vaatinusanalyysivaiheen oliot ja luokat

- ▶ **Vaatinusanalyysivaiheen luokat ovat vastineita reaalimaailman käsitteille**
- ▶ Kun edetään vaatimuksista ohjelmiston suunnitteluun, monet vaatinusanalyysivaiheen luokista saavat vastineensa "ohjelmostitason" luokkina, eli luokkina, jotka on tarkoitus ohjelmoida esim. Javalla
- ▶ Eli riippuen katsantokulmasta, luokka voi olla joko
 - ▶ reaalimaailman käsitteen vastine, tai
 - ▶ suunnittelu- ja ohjelmostitason "tekniinen" asia
- ▶ Tyypillisesti ohjelmostason olio on vastine jollekin todellisuudessa olevalle "oliolle" (*simuloidaan todellisuutta*)
- ▶ Ohjelmissa on myös paljon luokkia ja olioita, joille ei ole vastinetta todellisuudessa (Esim. käyttöliittymän oliot)

Oliot ja luokat

Oliomallinnus ja olioperustainen ohjelmistokehitys

- ▶ Olioperustainen ohjelmistokehitys etenee yleensä seuraavasti:
 1. Luodaan **määrittelyvaiheen oliomalli** sovelluksen käsitteistöstä
 - ▶ Mallin oliot ja luokat ovat rakennettavan sovelluksen kohdealueen käsitteiden vastineita
 2. Suunnitteluvaiheessa tarkennetaan edellisen vaiheen oliomalli **suunnitteluvaiheen oliomalliksi**
 - ▶ Oliot muuttuvat yleiskäsitteistä teknisen tason olioiksi
 - ▶ Mukaan tulee olioita, joilla ei suoraa vastinetta reaalimaailmassa
 - ▶ Osa olioista on luonteeltaan *pysyviä* ja niitä tulee vastaamaan jokin rakenne ohelman tietokannassa
 3. Toteutetaan suunnitteluvaiheen oliomalli jollakin **olio-ohjelointikielellä**
- ▶ Voidaankin ajatella, että malli tarkentuu muuttuen koko ajan ohjelointikieliläheisemmäksi/teknisemmäksi siirryttääessä määrittelystä suunnittelun ja toteutukseen

Luokka- ja oliokaaviot

Olioden ja luokkien kuvaus UML:ssä

Luokka- ja oliokaaviot

Olioden ja luokkien kuvaus UML:ssä

- ▶ Miten olioita ja luokkia voidaan kuvata UML:ssä³?
- ▶ Järjestelmän luokkarakennetta kuvaaa **luokkakaavio** (engl. *class diagram*)
 - ▶ Mitä luokkia olemassa
 - ▶ Minkälaisia luokat ovat
 - ▶ Luokkien ja niiden olioiden suhteet toisiinsa
- ▶ Luokkakaavio on UML:n eniten käytetty kaaviotyyppi
- ▶ Luokkakaavio kuvaaa ikäänkuin kaikkia mahdollisia olioita, joita järjestelmässä on mahdollista olla olemassa
- ▶ **Oliokaavio** (engl. *object diagram*) taas kuvaaa mitä olioita järjestelmässä on tiettyllä hetkellä

³Mikä on UML? Luennon 1 kalvoissa on kaikki mitä tarvitset.

Luokka- ja oliokaaviot

Olioden ja luokkien kuvaus UML:ssä

- ▶ Luokkaa kuvataan laatikolla, jonka sisällä on luokan nimi
- ▶ Luokasta luotuja olioita kuvataan myös laatikolla, erona on nimen merkintätapa
 - ▶ Nimi alleviivattuna, sisältäen mahdollisesti myös olion nimen
- ▶ Kuvassa Henkilö-luokka ja kolme Henkilö-olioa
- ▶ Luokkia ja olioita ei sotketa samaan kuvaan, kyseessä onkin kaksi kuvaaa: vasemmalla luokkakaavio ja oikealla oliokaavio
 - ▶ Oliokaavio kuvaa tietyn hetken tilanteen, olemassa 3 henkilöä, joista yksi on nimeton

Luokkakaavio

Eräs luokkakaaviota vastaava oliokaavio

Henkilö

arto : Henkilö

heikki : Henkilö

: Henkilö

Luokka- ja oliokaaviot

Attribuutit

- ▶ Luokan olioilla on attribuutteja eli oliomuuttujia ja operaatiota eli metodeja
- ▶ Nämä määritellään luokkamäärittelyn yhteydessä
 - ▶ Aivan kuten Javassa kirjoitettaessa class Henkilö ... määritellään kaikkien Henkilö:n attribuutit ja metodit luokan määrittelyn yhteydessä
- ▶ Luokkakaaviossa attribuutit määritellään luokan nimen alla omassa osassaan laatikkoa
 - ▶ Attribuutista on ilmaistu nimi ja tyyppi (voi myös puuttua)
- ▶ Oliokaaviossa voidaan ilmaista myös attribuutin arvo

| Henkilö |
|----------------------------|
| nimi : String ika : int |

| arto : Henkilö |
|---------------------------|
| nimi = "Arto" ika = 52 |

| heikki : Henkilö |
|-----------------------------|
| nimi = "Heikki" ika = 54 |

Luokka- ja oliokaaviot

Metodit

- ▶ Luokan olioiden metodit merkitään laatikon kolmanteen osaan
- ▶ Luokkiin on **pakko merkitä ainoastaan nimi**
 - ▶ Attribuutit ja metodit merkitään jos tarvetta
 - ▶ Usein metodeista merkitään ainoastaan nimi, joskus myös parametrien ja paluuarvon tyyppi
- ▶ Attribuuttien ja operaatioiden parametrien ja paluuarvon tyyppineinä voidaan käyttää valmiita tietotyyppejä (int, double, String, ...) tai rakenteisia tietotyyppejä (esim. taulukko, ArrayList).
- ▶ Tyyppi voi olla myös luokka, joko itse määritelty tai asiayhteydestä "itsestäänselvä"(alla Väri ja Piste)

| Henkilö |
|----------------------------|
| nimi : String ika : int |
| vanhene() menetoihin() |

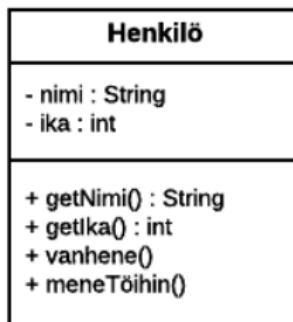
| Tiedosto |
|-------------------------------------|
| nimi kokoTavuina muokkausAika |
| tulosta() |

| Kuvio |
|--|
| vari : Vari sijainti : Piste |
| kierra(kulma : double) siirra(suunta) |

Luokka- ja oliokaaviot

Attribuuttien ja operaatioiden näkyvyys

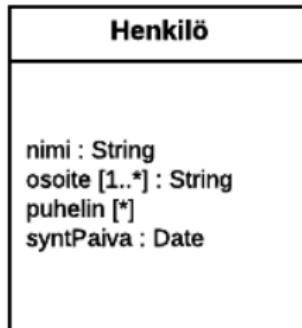
- ▶ Ohjelmointikielissä voidaan attribuuttien ja metodien näkyvyyttä muiden luokkien olioille säädellä
 - ▶ Javassa private, public, protected
- ▶ UML:ssa näkyvyys merkitään attribuutin tai metoden eteen: public +, private -, protected #, package ~
 - ▶ Jos näkyvyyttä ei ole merkitty, sitä ei ole määritelty (Kovin usein näkyvyyttä ei viitsitä merkitä)
- ▶ Esim. alla kaikki attribuutit ovat private eli eivät näy muiden luokkien olioille, metodit taas public eli kaikille julkisia



Luokka- ja oliokaaviot

Attribuutin moniarvoisuus

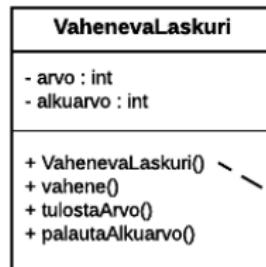
- ▶ Jos attribuutti on kokoelma samanlaisia arvoja (esim. taulukko), voidaan se merkitä luokkakaavioon
- ▶ Kirjoitetaan attribuutin perään hakasulkeissa kuinka monesta "asiasta" attribuutti toistuu (* = tuntematon)
- ▶ Henkilöllä on vähintään 1 osoite, mutta osoitteita voi olla useita
- ▶ Puhelinnumerona kohdalle on kirjoitettu [*], joka tarkoittaa, että numeroa ei välttämättä ole tai numeroita voi olla useita
- ▶ **Moniarvoisuus on asia, joka merkitään kaavioon harvoin!**



Luokka- ja oliokaaviot

VahenevaLaskuri esimerkki

```
public class VahenevaLaskuri {  
    private int arvo;  
    private int alkuarvo;  
  
    public VahenevaLaskuri(int arvo) {  
        this.arvo = arvo;  
        this.alkuarvo = arvo;  
    }  
  
    public void vahene() {  
        if ( this.arvo>0 ) {  
            this.arvo--;  
        }  
    }  
  
    public void tulostaArvo() {  
        System.out.println(this.arvo);  
    }  
  
    public void palautaAlkuarvo() {  
        this.arvo = this.alkuarvo;  
    }  
}
```

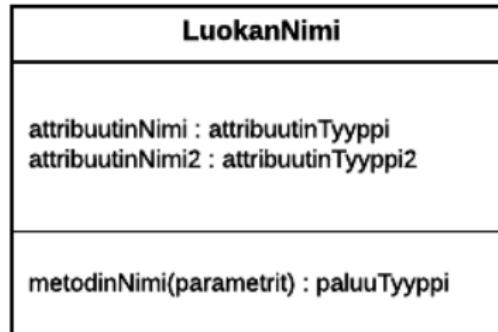


*Kuvassa mukana
UML-komenttisymboli!*

Luokka- ja oliokaaviot

Yhteenvetö

- ▶ Eli luokka on laatikko, jossa luokan nimi ja tarvittaessa attribuutit sekä metodit
- ▶ Attribuuttien ja metodien parametrien ja paluuarvon tyyppi ilmaistaan tarvittaessa
 - ▶ Näkyvyysmääreet ilmaistaan tarvittaessa
- ▶ Parametrin tyyppi voidaan myös merkitä "Javamaisesti", esim.
Date : aika tai int : ikä
 - ▶ myös kaksoispisteen voi jättää kokonaan pois...
- ▶ Jos esim. metodeja ei haluta näyttää, jätetään metodiosa pois, vastaavasti voidaan menetellä attribuuttien suhtein



Olioiden väliset yhteydet

Yleisesti yhteyksistä

- ▶ Olioiden välillä on yhteyksiä:
 - ▶ Työntekijä *työskentelee* Yrityksessä
 - ▶ Henkilö *omistaa* Auton
 - ▶ Henkilö *ajaa* Autolla
 - ▶ Auto *sisältää* Renkaat
 - ▶ Henkilö *asuu* Osoitteessa
 - ▶ Henkilö *omistaa* Osakkeita
 - ▶ Työntekijä *on* Johtajan alainen
 - ▶ Johtaja *johtaa* Työntekijötä
 - ▶ Johtaja *erottaa* Työntekijän
 - ▶ Opiskelija *on* ilmoittautunut Kurssille
 - ▶ Kello *sisältää* kolme Viisaria
- ▶ Olioiden välinen yhteys voi olla pysyvämpiluontoinen (rakenteinen) tai hetkellinen
 - ▶ Aluksi keskitytään pysyvämpiluontoisiin yhteyksiin

Olioiden väliset yhteydet

Yleisesti yhteyksistä

- ▶ Ohjelmakoodissa pysvä yhteys ilmenee yleensä luokassa olevana olioviitteenä, eli oliomuuttujana jonka tyyppinä on luokka
- ▶ Esimerkiksi **Henkilö** omistaa **Auton**:

```
public class Auto {  
    public void aja() {  
        System.out.println("liikkuu");  
    }  
  
    public class Henkilö {  
        private Auto omaAuto;  
  
        public Henkilö(Auto auto) {  
            omaAuto = auto  
        }  
  
        public void meneTöihin() {  
            omaAuto.aja();  
        }  
    }  
}
```

Olioiden väliset yhteydet

Assosiaatio

Olioiden väliset yhteydet

Assosiaatio

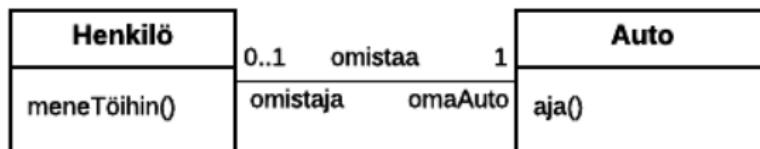
- ▶ Olioviite voitaisiin periaatteessa merkitä luokkakaavioon attribuuttina, kyseessähän on teknisessä mielessä attribuutti
- ▶ Parempi tapa on kuvata olioiden välinen yhteys luokkakaaviossa
 - ▶ Jos Henkilö- ja Auto-olion välillä voi olla yhteys, yhdistetään Henkilö- ja Auto-luokat viivalla
- ▶ Tilanne kuvattu alla
 - ▶ Yhteydelle on annettu nimi omistaa, eli Henkilö-olio omistaa Auto-olion



Olioiden väliset yhteydet

Kytkenräajoitteet ja roolit

- ▶ Ohjelmakoodissa jokaisella henkilöllä on täsmälleen yksi auto ja auto liittyy korkeintaan yhteen henkilöön
- ▶ Tämä kuvataan kytkenräajoitteina (engl. *multiplicity*)
 - ▶ Alla yhteyden oikeassa päässä on numero 1, joka tarkoittaa, että yhteen Henkilö-olioon liittyy täsmälleen yksi Auto-olio
 - ▶ Yhteyden vasemmassa päässä 0..1, joka tarkoittaa, että yhteen Auto-olioon liittyy 0 tai 1 Henkilö-oliaa
- ▶ Auton *rooli* yhteydessä on olla henkilön omaAuto, rooli on merkitty Auton viereen
 - ▶ Huom: roolin nimi on sama kun luokan Henkilö oliomuuttuja, jonka tyyppinä Auto
- ▶ Henkilön rooli yhteydessä on olla omistaja



Olioiden väliset yhteydet

Kytkentärajoitteet ja roolit

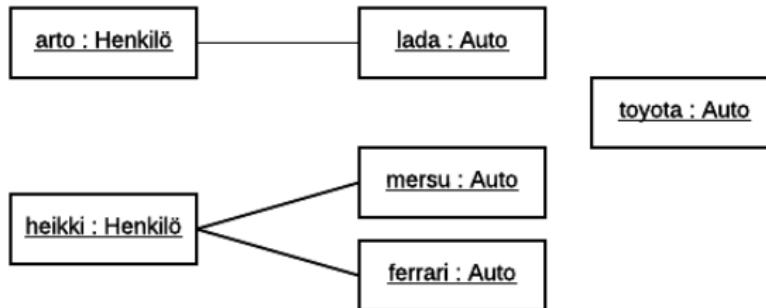
- ▶ Edellisessä esimerkissä yhdellä Henkilö-oliolla on yhteys täsmälleen yhteen Auto-olioon
 - ▶ Eli yhteyden Auto-päässä on kytkentärajoitteena 1
- ▶ Jos halutaan mallintaa tilanne, jossa kullakin Henkilö-oliolla voi olla mielivaltainen määrä autoja (nolla tai useampi), niin kytkentärajoitteeksi merkitään *

| | |
|------|--------------------------|
| 0 | Ei yhtään (harvinainen!) |
| 0..1 | Ei yhtään tai yksi |
| 1 | Tasan yksi |
| 0..* | Ei yhtään tai enemmän |
| * | Sama kuin 0..* |
| 1..* | Yksi tai enemmän |

Olioiden väliset yhteydet

Yhteydet oliokaaviossa

- ▶ Luokkakaavio kuvaaa luokkien olioiden kaikkia mahdollisia suhteita
 - ▶ Edellisessä sivulla sanotaan vaan, että tietyllä henkilöllä voi olla useita autoja ja tietyllä autolla on ehkä omistaja
- ▶ Jos halutaan ilmaista asioiden tila jollain ajankohdella, käytetään oliokaaviota
 - ▶ Mitä olentoja on tietyllä hetkellä olemassa ja miten ne yhdistyvät?
- ▶ Alla tilanne, jossa Artolla on 1 auto ja Heikillä 3 autoa, yhdellä autolla ei ole omistajaa



Olioiden väliset yhteydet

Yhden suhde moneen -yhteyden toteuttaminen Javassa

- ▶ Jos henkilöllä on korkeintaan yksi auto, on Henkilö-luokalla siis attribuutti, jonka tyyppi on Auto
 - ▶ private Auto omaAuto;
- ▶ Jos henkilöllä on monta autoa, on Javassa yleinen ratkaisu lisätä Henkilö-luokalle attribuutiksi listallinen (esim. ArrayList) autoja:
 - ▶ private ArrayList<Auto> omatAutot;
- ▶ ArrayLististä tarkemmin Ohjelmoinnin perusteiden materiaalissa

Olioiden väliset yhteydet

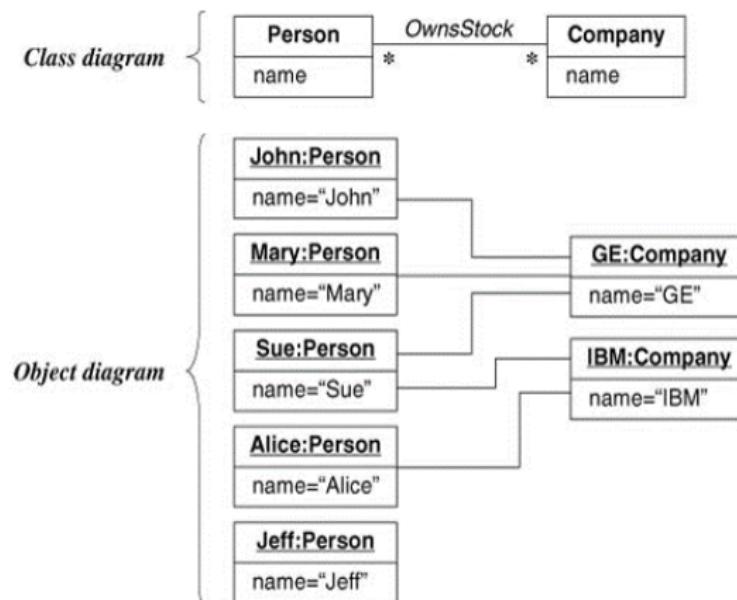
Helppoja yhteysesimerkkejä

- ▶ Kuten niin moni asia UML:ssä, on myös yhteyden nimen ja roolinimien merkintä vapaaehtoista
- ▶ Jos kytikentärajoite jätetään merkitsemättä, niin silloin yhteydessä olevien olioiden lukumäärä on määrittelemätön
- ▶ **Seuraavaksi joukko esimerkkejä →**

Olioiden väliset yhteydet

Helppoja yhteysesimerkkejä

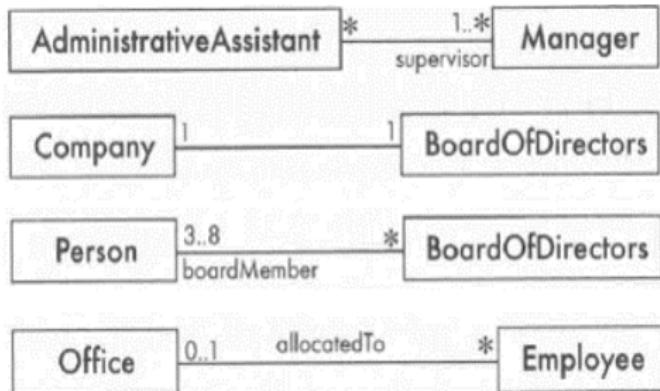
- ▶ Henkilö voi omistaa usean yhtiön osakkeita
- ▶ Yhtiöllä on monia osakkeenomistajia
- ▶ Eli yhteen Henkilö-olioon voi liittyä monta Yhtiö-oliaa
- ▶ Ja yhteen Yhtiö-olioon voi liittyä monta Henkilö-oliaa



Olioiden väliset yhteydet

Helppoja yhteysesimerkkejä

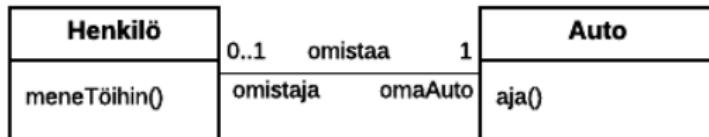
- ▶ Manageria kohti on useita assistenteja, assistentin johtajana (supervisor) toimii vähintään yksi manageri
- ▶ Yhtiöllä on yksi johtokunta, joka johtaa tasan yhtä yhtiötä
- ▶ Johtokuntaan kuuluu kolmesta kahdeksaan henkeä. Yksi henkilö voi kuulua useisiin johtokuntiin, muttei välttämättä yhteenkään.
- ▶ Toimistoon on sijoitettu (allocatedTo) useita työntekijöitä. Työntekijällä on paikka yhdessä toimisto tai ei missään



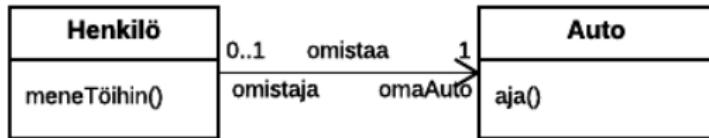
Olioiden väliset yhteydet

Yhteyden navigointisuunta

- ▶ Palautetaan mieleen Auto ja Henkilö -esimerkki⁴



- ▶ Auto-luokan koodista huomaamme, että auto-oliot eivät tunne omistajaansa
 - ▶ Henkilö-oliot taas tuntevat omistamansa autot Auto-tyyppisen attribuutin omaAuto ansiosta
- ▶ Yhteys siis on oikeastaan yksisuuntainen, henkilöstä autoon
- ▶ Asia voidaan ilmaista kaaviossa tekemällä yhteysviivasta nuoli
 - ▶ Nuolen kärki sinne suuntaan, johon on pääsy oliomuuttujan avulla (nyrkisääntö!)



⁴Katso Kytkentärajoitteet ja roolit kalvolta numero 91.

Olioiden väliset yhteydet

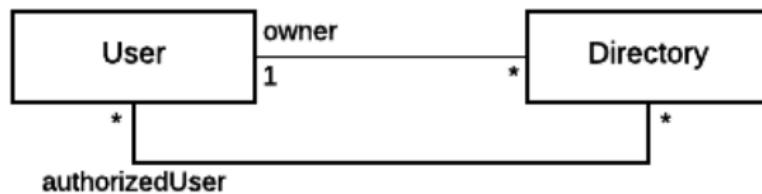
Yhteyden navigointisuunta

- ▶ Yhteyden navigointisuunnalla merkitystä lähinnä suunnittelua ja toteutustason kaavioissa
 - ▶ Merkitään vain jos suunta tärkeää tietää
 - ▶ Joskus kaksisuuntainen merkitään nuolella molempien suuntiin
 - ▶ Joskus taas nuoleton tarkoittaa kaksisuuntaista
- ▶ Määrittelytason luokkakaavioissa yhteyden suuntia ei yleensä merkitä ollenkaan
- ▶ Yhteyden suunnalla on aika suuri merkitys sillä, kuinka yhteys toteutetaan kooditasolla

Olioiden väliset yhteydet

Useampia yhteyksiä olioiden välillä

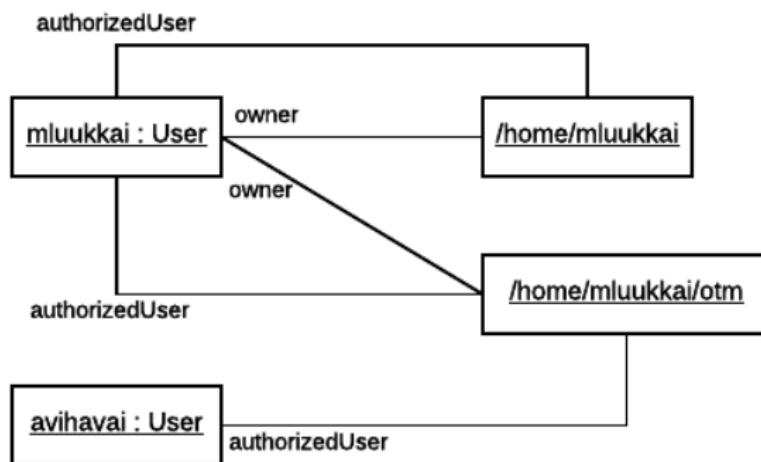
- ▶ Esim. Linuxissa jokaisella hakemistolla on tasan yksi omistaja
- ▶ Eli yhteen hakemisto-olioon liittyy roolissa owner tasan yksi käyttäjä-olio
- ▶ Jokaisella hakemistolla voi olla lisäksi useita käyttäjiä
 - ▶ Yhteen hakemistoon liittyy useita käyttäjiä roolissa authorizedUser
- ▶ Yksi käyttäjä voi omistaa useita hakemistoja
- ▶ Yhdellä käyttäjällä voi olla käyttöoikeus useisiin hakemistoihin
- ▶ Yhdellä käyttäjällä voi olla samaan hakemistoon sekä omistus- että käyttöoikeus



Olioiden väliset yhteydet

Useampia yhteyksiä olioiden välillä

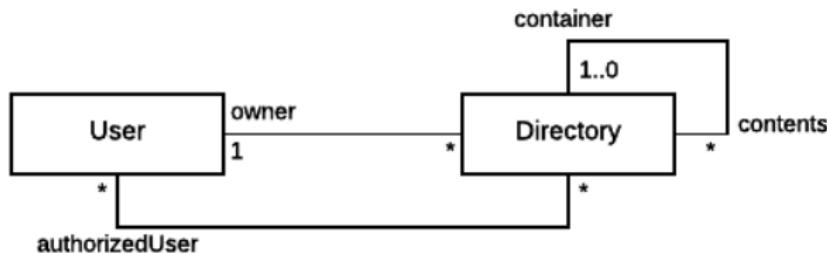
- ▶ Ohessa oliokaavio, joka kuvaa eräään edellisen luokkakaavion mukaisen tilanteen
 - ▶ Käyttäjät: mluukkai ja avihavai
 - ▶ mluukkai omistaa kaksi hakemistoa
 - ▶ mluukkai omistaa kaksi hakemistoa
 - ▶ Samojen olioiden välillä kaksi eri yhteyttä!
 - ▶ avihavai:lla käyttöoikeus hakemistoon /home/mluukkai/otm



Olioiden väliset yhteydet

Yhteys kahden saman luokan olion välillä

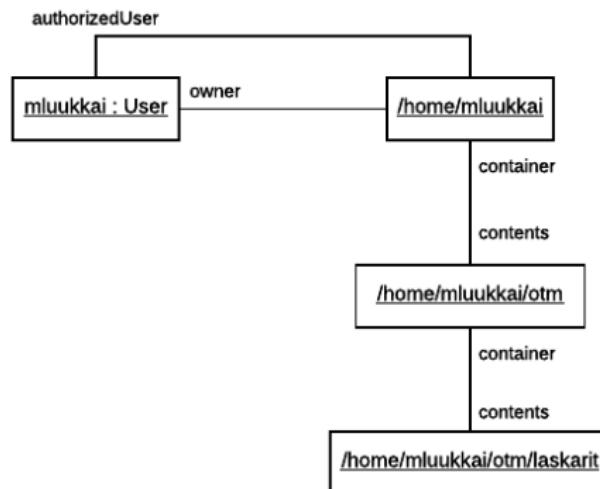
- ▶ Miten mallinnetaan se, että hakemistolla on alihakemistoja?
 - ▶ Hakemisto sisältää alihakemistoja
 - ▶ Hakemisto sisältyy johonkin toiseen hakemistoon
- ▶ Yhteen hakemisto-olioon voi liittyä 0 tai 1 hakemisto-oliaa roolissa *container* (=sisältäjä), eli hakemisto voi olla jonkin toisen hakemiston alla
- ▶ Yhteen hakemisto-olioon voi liittyä mielivaltainen määrä (*) hakemisto-olioita roolissa *contents* (=sisältö), eli hakemisto voi sisältää muita hakemistoja



Olioiden väliset yhteydet

Yhteys kahden saman luokan olion välillä

- ▶ Tilanne vaikuttaa sekavalta, selvennetään oliokaavion avulla
- ▶ /home/mluukkai hakemiston
 - ▶ Alihakemistojen rooli yhteydessä on *contents* eli sisältö
 - ▶ Päähakemiston rooli yhteydessä on *container* eli sisältäjä
- ▶ /home/mluukkai/otm on edellisen alihamiston, mutta sisältää itse alihamiston

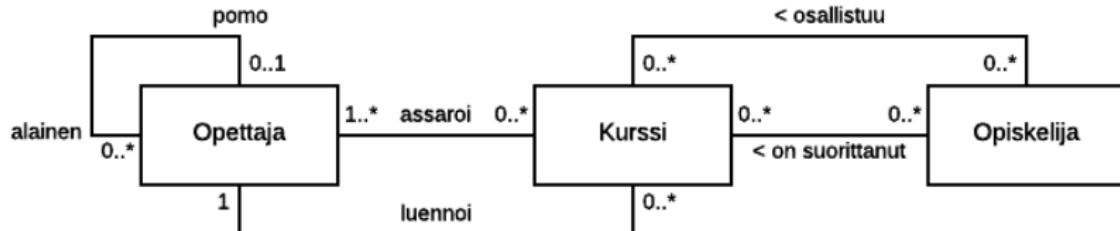


Olioiden väliset yhteydet

Monimutkaisempi esimerkki assosiaatioista

► Mallinnetaan seuraava tilanne:

- ▶ Kurssilla on luennoijana 1 opettaja ja assarina useita opettajia
- ▶ Opettaja voi olla useiden kurssien assarina ja luennoijana
- ▶ Opettajalla voi olla pomo ja useita alaisia
- ▶ Opettajat johtavat toisiaan
- ▶ Opiskelija voi osallistua useille kursseille
- ▶ Opiskelijalla voi olla suorituksia useista kursseista



Olioiden väliset yhteydet

Riippuvuus

Olioiden väliset yhteydet

Riippuvuus

- ▶ Muistellaan taas Auto-esimerkkiä:
- ▶ On olemassa henkilötä, jotka eivät omista autoa
- ▶ Autottomat henkilöt kuitenkin välillä lainaavat jonkin muun autoa
- ▶ Koodissa asia voitaisiin ilmaista seuraavasti:

```
public class AutotonHenkilo {  
    public void lainaaJaAja( Auto lainaAuto ) {  
        lainaAuto.aja();  
    }  
}
```

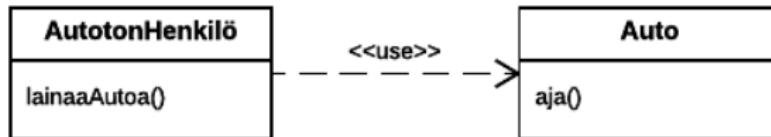
- ▶ Eli autottoman henkilön metodi lainaaJaAja saa parametrikseen auton (lainaAuto), jolla henkilö ajaa
- ▶ Auto on lainassa ainoastaan metodin suoritusajan
 - ▶ AutotonHenkilo ei siis omaa pysyvää suhdetta autoon

Olioiden väliset yhteydet

Riippuvuus

Riippuvuus on tavallaan myös yhteys, mutta "normaaliala" yhteyttä "heikompi" (koska se ei kestä yhtä kauaa)

- ▶ Koska kyseessä ei ole pysyvämpiluontoinen yhteys, on parempi käyttää luokkakaaviossa riippuvuussuhdetta (engl. dependency)
- ▶ Riippuvuus merkitään katkoviivanuolena, joka osoittaa siihen luokkaan josta ollaan riippuvaisia
- ▶ Alla on ilmaistu vielä riippuvuuden laatu
 - ▶ Tarkennin (eli *stereotyyppi*) «use» kertoo että kyseessä on käyttöriippuvuus, eli AutotonHenkilö kutsuu Auto:n metodia



Olioiden väliset yhteydet

Riippuvuus

- ▶ Joskus riippuvuus määritellään siten, että luokka A on riippuvainen luokasta B jos muutos B:hen saa aikaan mahdollisesti muutostarpeen A:ssa
 - ▶ Näin on edellisessä esimerkissä: jos Auto-luokka muuttuu (esim. metodi aja muuttuu siten että se tarvitsee parametrin), joudutaan AutotonHenkilö-luokkaa muuttamaan
- ▶ Riippuvuus on siis jotaain *heikompaa* kun tavallinen luokkien välinen yhteys
 - ▶ Jos luokkien välillä on yhteys, on niiden välillä myös riippuvuus, sitä ei vaan ole tapana merkitä (Henkilö joka omistaa Auton on riippuvainen autosta)

Huomaathan, että toisin kuin yhteyksiin, **riippuvuuksiin ei merkitä kytkentärajoitteita**

Olioiden väliset yhteydet

Kompositio

Olioiden väliset yhteydet

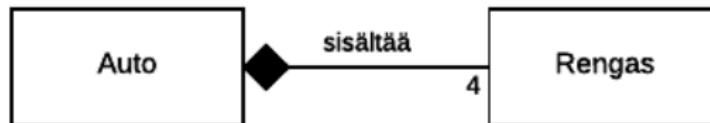
Kompositio

- ▶ Autossa on 4 pyörää, joten tilannehan voitaisiin mallintaa tekemällä autosta yhteys Rengas-olioon ja laittamalla kytkentärajoitteeksi 4
- ▶ Renkaat ovat kuitenkin siinä mielessä erityisessä asemassa, että voidaan ajatella, että ne ovat auton komponentteja
 - ▶ Renkaat sisältyvät autoon
- ▶ Kun auto luodaan, luodaan renkaat samalla
 - ▶ Koodissa auto luo renkaat
- ▶ Renkaat ovat private, eli niihin ei pääse ulkopuolelta käsiksi
- ▶ Kun roskienkerääjä tuhoaa auton, tuhoutuvat myös renkaat
- ▶ **Eli ohjelman renkaat sisältyvät autoon ja niiden elinkä on sidottu auton elinkään** (oikeat renkaat eivät tietenkään käytädy näin vaan ovat vaihdettavissa)

Olioiden väliset yhteydet

Kompositio

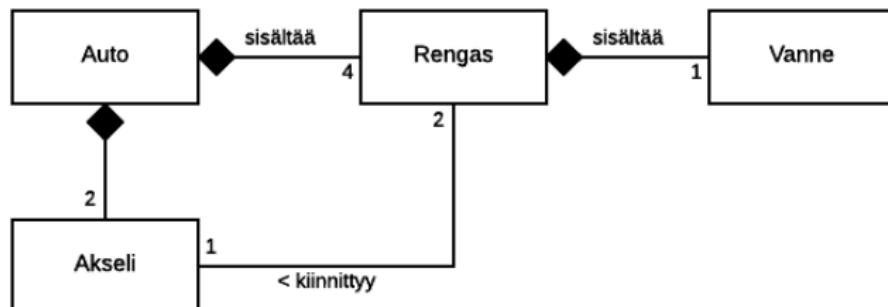
- ▶ Edellisen dian tilannetta nimitetään **kompositioksi** (engl. *composition*)
- ▶ Komposition symboli on "musta salmiakkikuvio", joka liitetään yhteyden siihen päähän, johon osat sisältyvät (nyrkisääntö!)
- ▶ Kompositiota käytetään kun seuraavat ehdot toteutuvat:
 1. Osat ovat olemassaoloriippuvaisia kokonaisuudesta
 2. Osa voi kuulua vaan yhteen kompositioon
 - ▶ Rengasta ei voi siirtää toiseen autoon
 3. Osa on koko elinaikansa kytketty samaan kompositioon
- ▶ Koska Rengas-olio voi liittyä nyt vain yhteen Auto-olioon, ei salmiakin puoleiseen päähän tarvita osallistumisrajoitetta koska se on joka tapauksessa 1



Olioiden väliset yhteydet

Monimutkaisempi esimerkki kompositiosta

- ▶ Tarkennettu Auto sisältää 4 rengasta ja 2 akselia
- ▶ Komposition osa voi myös sisältää oliota
 - ▶ Rengas sisältää vanteen
- ▶ Komposition osilla voi olla "normaaleja" yhteyksiä
 - ▶ Akseli kiinnittyy kahteen renkaaseen
 - ▶ Rengas on kiinnittynyt yhteen akseliin



Olioiden väliset yhteydet

Monimutkaisempi esimerkki kompositiosta

- ▶ Onko kompositiomerkkiä pakko käyttää?
 - ▶ Ei, mutta usein sen käyttö selkiytyy tilannetta
- ▶ Kompositiota ei kannata laittaa sinne minne se ei kuulu!
 - ▶ Kompositio on erittäin rajoittava suhde olioiden välillä, toisin kuin "normaali" yhteys, käytä kompositiota vasta kun olet tarkistanut onko kaikki sen ehdot täyttyneet
- ▶ **HUOM:** auton ja renkaat sisältävä esimerkki kuvaaa vaan esimerkkikoodi tilannetta mutta ei tietenkään ole realistinen kuvaamaan reaalimaailman autojen ja renkaiden suhdetta sillä normaalistaan renkaat eivät ole autoista olemassaolioriippuvaisia

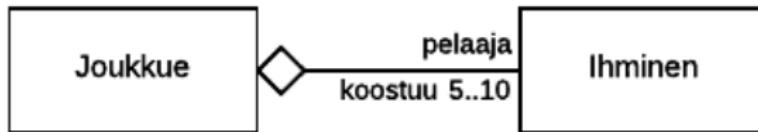
Olioiden väliset yhteydet

Kooste

Olioiden väliset yhteydet

Kooste

- ▶ Koosteella (engl. aggregation) tarkoitetaan koostumussuhdetta, joka ei ole yhtä komposition tapaan "ikuinen"
- ▶ Koosteella (engl. aggregation) tarkoitetaan koostumussuhdetta, joka ei ole yhtä komposition tapaan "ikuinen"
 - ▶ **HUOM:** suomenkiiset termit kooste ja kompositio ovat huonot ja jopa harhaanjohtavat
- ▶ Koostetta merkitään "valkoisella salmiakilla" joka tulee siihen päähän yhteyttä, johon osat kuuluvat
- ▶ Esimerkki: Joukkue koostuu pelaajista (jotka ovat ihmisiä)
 - ▶ Ihminen ei kuitenkaan kuulu joukkueeseen ikuisesti
 - ▶ Joukkue ei syynnytä eikä tapa pelaajaa
 - ▶ Ihminen voi kuulua yhtäaikaa useampaan joukkueeseen



Olioiden väliset yhteydet

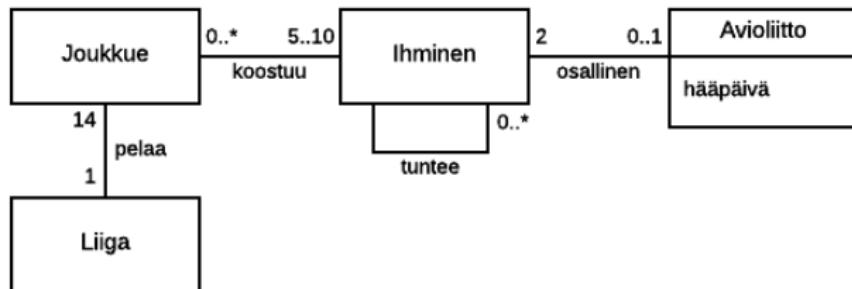
Kooste

- ▶ Komposition (eli mustan salmiakin) merkitys on selkeää, kyseessä on olemassaolioriippuvuus
- ▶ On sensijaan epäselvää millon koostetta (eli valkoista salmiakkia) tulisi käyttää normaalilin yhteyden sijaan
- ▶ Monet asiantuntevat oliomallintajat ovat sitä mieltä että koostetta ei edes tulisi käyttää
- ▶ Koostesuhde on poistunut UML:n standardista versiosta 2.0 lähtien
- ▶ Koostesuhde on kuitenkin edelleen erittäin paljon käytetty joten on hyvä tuntea symbooli passiivisesti
- ▶ **Tällä kurssilla koostetta ei käytetä eikä sitä tarvitse osata!**
- ▶ Joukkueen ja pelaajien välinen suhde voidaankin ilmaista normaalina yhteytenä

Olioiden väliset yhteydet

Monimutkaisempi esimerkki Koosteen kierrosta

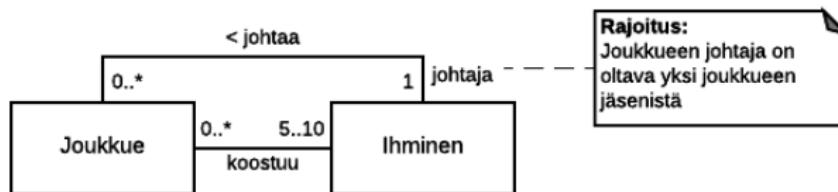
- ▶ Joukkue pelaa liigassa, jossa on 14 joukkuetta
- ▶ Ihminen voi kuulua mielivaltaisen moneen joukkueeseen
- ▶ Joukkueeseen kuuluu 5-10 ihmistä
- ▶ Ihminen tuntee useita ihmisiä
- ▶ Ihminen voi olla avioliitossa, mutta vain yhdessä avioliitossa kerrallaan
- ▶ Avoliitto koostuu kahdesta ihmisestä



Olioiden väliset yhteydet

Rajoitukset

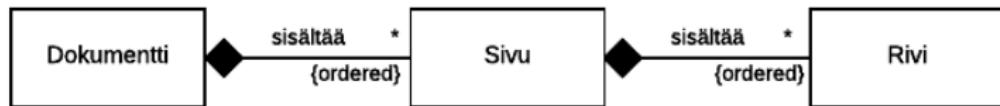
- ▶ Jos haluttaisiin mallintaa tilanne, että joku joukkueen jäsenistä on joukkueen johtaja, pelkkä luokkakaavio (siinä määrin kun tällä kurssilla UML:ää opitaan) ei riitä
- ▶ Tilanne voitaisiin mallintaa alla esitettyllä tavalla
 - ▶ Eli lisätään normaali yhteys *johtaa* joukkueen ja ihmisen välille
 - ▶ Määritellään kytkentärajoite: joukkueella on tasana 1 johtaja
 - ▶ Ilmaistaan UML-komenttina, että joukkueen johtajan on oltava joku joukkueen jäsenistä



Olioiden väliset yhteydet

Yhteydessä olevien luokkien järjestys

- ▶ Esim. edellisessä esimerkissä joukkue koostuu pelaajista
 - ▶ Pelaajilla ei ole kuitenkaan mitään erityistä järjestystä yhteyden kannalta
- ▶ Joskus osien järjestys voi olla tärkeää
- ▶ Esim. dokumentti sisältää sivuja ja kukin sivu sisältää tekstirivejä
 - ▶ Sivujen ja rivien on oltava tiettyssä järjestyksessä, muuten dokumentissa ei ole järkeää
- ▶ Se seikka, että osat ovat järjestyksessä voidaan ilmaista lisämääreenä {ordered}, joka laitetaan niiden olioiden päähän yhteyttä joilla on järjestys



Työkaluja piirtoon

Ohjelmia ja nettisivuja

Työkaluja piirtoon

Ohjelmia ja nettisivuja

Millä kaaviot kannattaa piirtää?

- ▶ Kynä ja paperia tai valkotaulu
- ▶ Ilmaisia työkaluja
 - ▶ Dia (win+linux)
 - ▶ Umbrello
 - ▶ ArgoUML
 - ▶ Openoffice
- ▶ Maksullisia työkaluja:
 - ▶ Visual Paradigm
 - ▶ Magic Draw
 - ▶ Microsoft Visio
 - ▶ Omnipage (Mac)

Työkaluja piirtoon

Ohjelmia ja nettisivuja

Millä kaaviot kannattaa piirtää?

- ▶ Mahdollisuksia myös netissä:
 - ▶ <http://yuml.me/> luokka- ja käyttötapauskaavioihin
 - ▶ <https://www.websequencediagrams.com/> Sekvenssikaavioihin
 - ▶ <https://www.draw.io/> Jokapaikan höylä

Ei siis ole olemassa selkeää vastausta mitä työkalua kannattaa käyttää. Tämän kurssin tarpeisiin kynä ja paperia riittää hyvin

Ohjelmistotekniikan menetelmät

Luento 3

Määrittelyvaiheen luokkakaavio, sekvenssikaavio ja
kommunikaatiokaavio

Määrittelyvaiheen luokkakaavion laatiminen

Alustavan luokkamallin muodostaminen

Määrittelyvaiheen luokkakaavion laatiminen

Alustavan luokkamallin muodostaminen

- ▶ Kuten jo muutamaan kertaan on mainittu, olioperustaisessa ohjelmistokehityksessä pyritään muodostamaan koko ajan tarkentuva luokkamalli, joka *simuloi* sovelluksen kohdealuetta
 - ▶ Ensin luodaan **määrittelyvaiheen luokkamalli** sovelluksen käsitteistöstä
 - ▶ Suunnitteluvaiheessa tarkennetaan edellisen vaiheen luokkamalli **suunnitteluvaiheen luokkamalliksi**
- ▶ Tarkastellaan seuraavaksi miten alustava, määrittelyvaiheen luokkamalli voidaan muodostaa
 - ▶ Tunnistetaan sovellusalueen käsitteet ja niiden väliset suhteet
 - ▶ Eli karkeasti ottaen tehtävänä on etsiä todellisuutta simuloiva luokkarakenne
- ▶ Menetelmästä käytetään nimitystä **käsiteanalyysi** (engl. *conceptual modeling*)
 - ▶ Järjestelmän sovellusalueen käsitteistöä kuvaavaa luokkamallia kutsutaan **kohdealueen luokkamalliksi** (engl. problem domain model)

Määrittelyvaiheen luokkakaavion laatiminen

Alustavan luokkamallin muodostaminen

- ▶ Menetelmän voi ajatella etenevän seuraavien vaiheiden kautta
 1. Kartoita luokkaehdokkaat
 2. Karsi luokkaehdokkaita
 3. Tunnista olioiden välistä yhteydet
 4. Lisää luokille attribuutteja
 5. Tarkenna yhteyksiä
 6. Etsi "rivien välissä" olevia luokkia
 7. Etsi yläkäsitteitä
 8. Toista vaiheita 1-7 riittävän kauan
- ▶ Yleensä aloitetaan vaiheella 1 ja sen jälkeen edetään sopivalta tuntuvassa järjestyksessä
- ▶ On harvinaista, että ensimiettimältä päädytään *lopulliseen* ratkaisuun

Määrittelyvaiheen luokkakaavion laatiminen

Luokkaehdokkaiden kartoitus

- ▶ (1) Laaditaan lista tarkasteltavan sovelluksen kannalta keskeisistä asioista, kohteista ja ilmiöistä, joita ovat esim:
 - ▶ Toimintaan osallistujat
 - ▶ Toiminnan kohteet
 - ▶ Toimintaan liittyvät tapahtumat, materiaalit ja tuotteet ja välituotteet
 - ▶ Toiminnalle edellytyksiä luovat asiat
- ▶ Kartoituksen pohjana voi käyttää esim.
 - ▶ kehitettävästä järjestelmästä tehtyä vapaamuotoista tekstuualista kuvausta tai
 - ▶ järjestelmän halutusta toiminnallisuudesta laadittuja käyttötapaauksia
- ▶ **Luokkaehdokkaat** ovat yleensä järjestelmän toiminnan kuvaussessa esiintyviä **substantiiveja**

Määrittelyvaiheen luokkakaavion laatiminen

Luokkaehdokkaiden kartoitus

Etsitään käytettävissä olevista kuvauskuvista kaikki substantiivit ja otetaan ne alustaviksi luokkaehdokkaiksi:

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

Määrittelyvaiheen luokkakaavion laatiminen

Luokkaehdokkaiden kartoitus

Löydettiin seuraavat substantiivit:

- ▶ elokuvalippu
- ▶ varaaminen
- ▶ lippu
- ▶ paikka
- ▶ näytös
- ▶ elokuva
- ▶ esittäminen
- ▶ teatteri
- ▶ aika
- ▶ asiakas
- ▶ varaus

Määrittelyvaiheen luokkakaavion laatiminen

Ehdokkaiden karsiminen

- ▶ Nämä pitkällä listalla ovat varmasti ylimääräisiä luokkakandidaatteja
- ▶ (2) Karsitaan sellaiset jotka eivät vaikuta potentiaalisilta luokilta
- ▶ Kysymyksiä, joita karsissa voi miettiä
 - ▶ Onko käsitteellä merkitystä järjestelmän kannalta
 - ▶ Onko kyseessä jonkin muun termin synonyymi
 - ▶ Onko kyseessä jonkin toisen käsitteen attribuutti
 - ▶ Liittyykö käsitteeseen tietosisältöä?
 - ▶ On tosin olemassa myös tietosisällötömiä luokkia...
 - ▶ Onko käsitteeseen liittyvä tieto sellaista, että järjestelmän on *muistettava* tieto pitkiä aikoja
- ▶ Huomattavaa on, että kaikki luokat eivät yleensä edes esiinny sanallisissa kuvaluksissa, vaan ne löytyvät vasta jossain myöhemmässä vaiheessa

Määrittelyvaiheen luokkakaavion laatiminen

Ehdokkaiden karsiminen

Löydettiin seuraavat substantiivit:

- ▶ elokuvalippu ← Termin lippu synonyymi
- ▶ ~~varaaminen~~ ← Tekemistä
- ▶ lippu
- ▶ paikka
- ▶ näytös
- ▶ elokuva
- ▶ esittaminen ← Tekemistä
- ▶ teatteri
- ▶ aika ← Attribuutti (näytöksen)
- ▶ asiakas
- ▶ varaus

Määrittelyvaiheen luokkakaavion laatiminen

Alustava yhteyksien tunnistaminen

- ▶ Kun luokat on alustavasti tunnistettu, kannattaa ottaa paperia ja kynä ja piirtää alustava luokkakaavio, joka koostuu vasta luokkalaatikoista
- ▶ (3) Tämän jälkeen voi ruveta miettimään minkä luokkien välillä on yhteyksiä
- ▶ Aluksi yhteydet voidaan piirtää esim. pelkkinä viivoina ilman yhteys- ja roolinimiä tai osallistumisrajoitteita
- ▶ Tekstuaalisessa kuvaussessä olevat **verbit ja genetiivit** viittaavat joskus olemassaolevaan **yhteyteen**
 - ▶ Lippu *oikeuttaa* paikkaan tietyssä näytöksessä
 - ▶ Näytöksellä *tarkoitetaan* elokuvan esittämistä tietyssä teatterissa tiettyyn aikaan
 - ▶ Samaa elokuvaan *voidaan* esittää useissa teattereissa useina aikoina.
 - ▶ Asiakas voi samassa varauksessa *varata* useita lippuja yhteen näytökseen

Määrittelyvaiheen luokkakaavion laatiminen

Alustava yhteyksien tunnistaminen

Huom: Kaikki verbit eivät ole yhteyksiä! Yhteydellä tarkoitetaan pysyvästä suhdetta, usein verbit ilmentävät ohimeneviä asioita

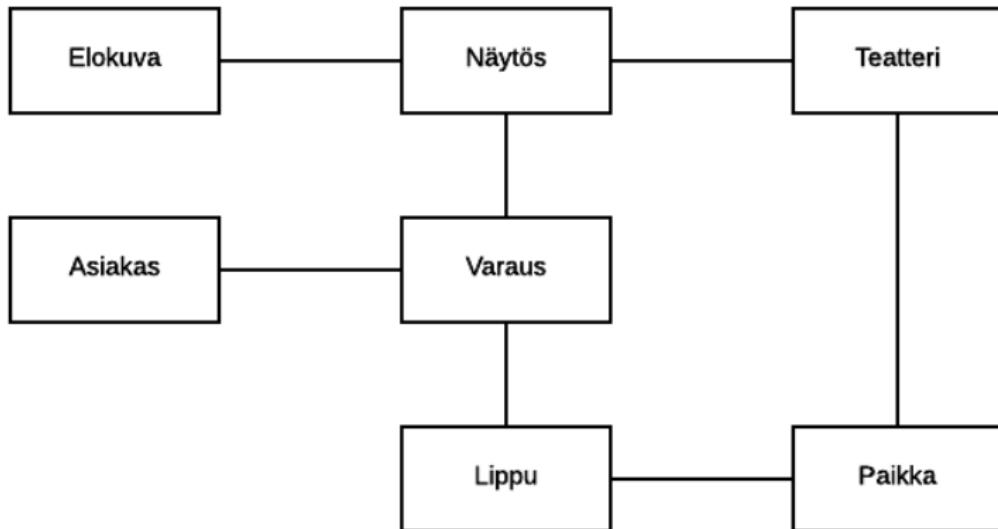
Löydettiin seuraavat yhteydet:

- ▶ Lippu – paikka
- ▶ Näytös – elokuva
- ▶ Näytös – teatteri
- ▶ Elokuvा – teatteri
- ▶ Asiakas – varaus
- ▶ Asiakas – lippu
- ▶ Lippu – varaus

Määrittelyvaiheen luokkakaavion laatiminen

Alustava yhteyksien tunnistaminen

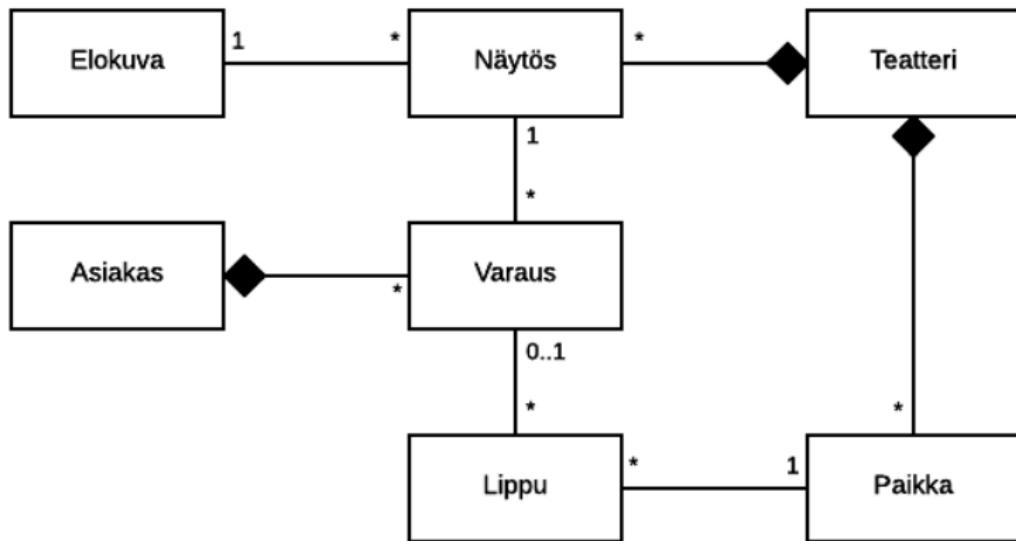
- ▶ Seuraavaksi jonkinlainen hahmotelma. Osa yhteyksistä siis edellisen listan ulkopuolelta, "maalaisjärellä" pääteltyjä:



Määrittelyvaiheen luokkakaavion laatiminen

Alustava yhteyksien tunnistaminen

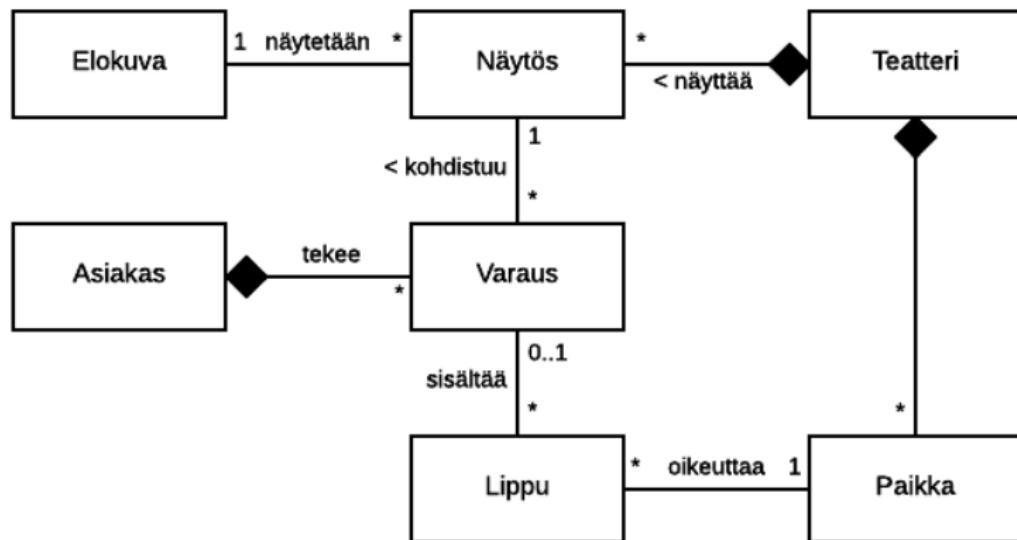
- ▶ Kun yhteys tunnistetaan ja vaikuttaa tarpeelliselta, tarkennetaan yhteyden laatua ja kytkentärajoitetta
- ▶ Ei ole olemassa oikeaa etenemisstrategiaa...



Määrittelyvaiheen luokkakaavion laatiminen

Yhteyksien tarkentaminen ja attribuuttien etsiminen

- ▶ (4) Attribuuttien löytäminen edellyttää yleensä lisätietoa, esim. asiakkaan haastatteluista
- ▶ Määrittelyvaiheen aikana tehtävää kohdealueen luokkamallia ei ole välttämättä tarkoitukseenmukaista tehdä kaikin osin tarkaksi
- ▶ (5) Tarkennetaan myös yhteyksiä



Määrittelyvaiheen luokkakaavion laatiminen

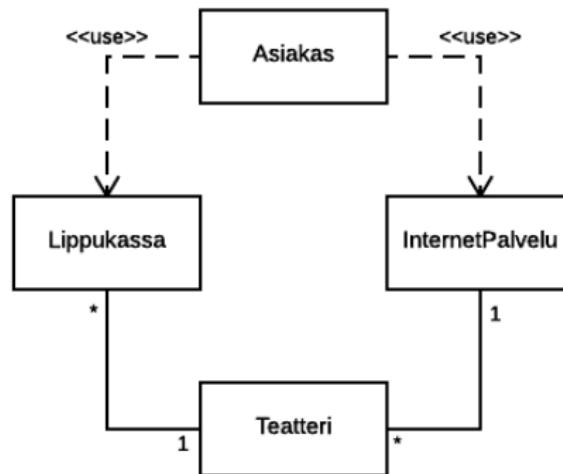
Mikä ei ole yhteys ja mikä ei?

- ▶ Oletetaan, että lipunvaraustapahtuman tekstuaaliseen kuvaukseen liittyisi myös seuraava:
 - ▶ Asiakas tekee lippuvarauksen elokuvateatterin internetpalvelun kautta. Elokuvateatterissa on useita lippukassoja. Asiakas lunastaa varauksensa lippukassalta viimeistään tuntia ennen esitystä.
- ▶ Tästä kuvauksesta löytyy kaksi uutta luokkakandidaattia:
 - ▶ internetpalvelu
 - ▶ lippukassa
- ▶ Tekstuaalisen kuvauksen perusteella teatterilla on yhteys internetpalveluun sekä lippukassoihin
- ▶ Ovatko nämä *rakenteellisia* yhteyksiä, jotka voidaan merkata myös luokkakaavioon?

Määrittelyvaiheen luokkakaavion laatiminen

Mikä ei ole yhteys ja mikä ei?

- ▶ Tekstuaalinen kuvaus antaa viitteen, että asiakalla on yhteys sekä internetpalveluun että lippukassaan
- ▶ **Näitä ei kuvata** luokkamallissa yhteytenä sillä kyse ei ole rakenteisesta, pysyvälaatuisesta yhteydestä
- ▶ Jos se, että asiakas käyttää lippukassan palvelua halutaan merkata luokkakaavioon, tulee yhteyden sijasta käyttää riippuvuutta



Määrittelyvaiheen luokkakaavion laatiminen

Mikä ei ole yhteys ja mikä ei?

- ▶ Edellisen dian kaltaisessa tilanteessa ei olisi mielekästä merkitä Asiakasta riippuvuutta Lippukassaan tai Internetpalveluun
- ▶ Luokka Asiakas on järjestelmässä oikean elokuvateatterin asiakkaan representaatio, joka ei kuitenkaan itsessään "suorita" mitään toimenpiteitä
- ▶ Hieman samaan tapaan yliopiston kuhunkin opiskelijaan liittyy OODI-järjestelmässä oma "olio", kuitenkaan tuo OODI:ssa oleva olio ei tee mitään, esim. käy luennoilla tai suorita kursseja
- ▶ Luokkamallin yhteyksissä siis ei tule kuvata toiminnallisutta, tyyliin Asiakas tekee varauksen Internetpalvelussa tai Opiskelija käy kurssin Luennolla vaan olioiden välisiä suhteita (jotka voivat olla toiminnan seuraus):

Suurin osa varsinaisen toiminnallisuden suorittavista olioista tulee vasta suunnittelutason luokkamalleihin.

Määrittelyvaiheen luokkakaavion laatiminen

Yläkäsiteiden etsiminen

- ▶ Seuraavaksi täytyy **(7)** etsiä yläkäsiteet
- ▶ Tämä liittyy periytymiseen ja palaamme asiaan seuraavalla luennolla
- ▶ **Lyhyesti:** jos tekisimme yleistä lippupalvelujärjestelmää, olisi lippu todennäköisesti yläkäsite, joka erikoistuu esim. elokuvalipuksi, konserttilipuksi, ym...
- ▶ Määrittelyvaiheen aikana tehtävään sovelluksen kohdealueen luokkamalliin ei vielä liitetä mitään metodeja
 - ▶ Metodien määrittäminen tapahtuu vasta ohjelman suunnitteluvaiheessa
 - ▶ Palaamme aiheeseen myöhemmin
 - ▶ Suunnitteluvaiheessa luokkamalli tarkentuu muutenkin monella tapaa

Määrittelyvaiheen luokkakaavion laatiminen

Mallinnuksen eteneminen

- ▶ Isoa ongelmaa kannattaa lähestyä pienin askelin, esim:
 - ▶ Yhteydet ensin karkealla tasolla, tai
 - ▶ Tehdään malli pala palalta, lisäten siihen muutama luokka yhteyksineen kerrallaan
- ▶ Mallinnus iteratiivisesti etenevässä ohjelmistokehityksessä
 - ▶ Ketterissä menetelmissä suositaan iteratiivista lähestymistapaa ohjelmistojen kehittämiseen
 - ▶ kerralla on määrittelyn, suunnittelun ja toteutuksen alla ainoastaan osa koko järjestelmän toiminnallisuudesta
 - ▶ Jos ohjelmiston kehittäminen tapahtuu ketterästi, kannattaa myös ohjelman luokkamallia rakentaa iteratiivisesti
 - ▶ Eli jos ensimmäisessä iteraatiossa toteutetaan ainoastaan muutaman käyttötapauksen kuvaama toiminnallisuus, esitetään iteraation luokkamallissa vain ne luokat, jotka ovat merkityksellisiä tarkastelun alla olevan toiminnallisuuden kannalta
 - ▶ Luokkamallia täydennetään myöhempien iteraatioiden aikana niiden mukana tuoman toiminnallisuuden osalta

Sekvenssikaavio

Olioiden yhteistyön mallintaminen

Sekvenssikaavio

Olioiden yhteistyön mallintaminen

- ▶ Luokkakaaviosta käy hyvin esille ohjelman *rakenne*
- ▶ Entä ohjelman toiminta?
 - ▶ Luokkakaaviossa voi olla metodien nimiä
 - ▶ Pelkät nimet eivät kuitenkaan kerro juuri mitään!
- ▶ Ohjelman toiminnan kuvaamiseen tarvitaan jotakin muuta
 - ▶ Tarve kuvata esim. skenaario "ostetaan 3 euroa sisältävällä lyyrakortilla edullinen lounas"

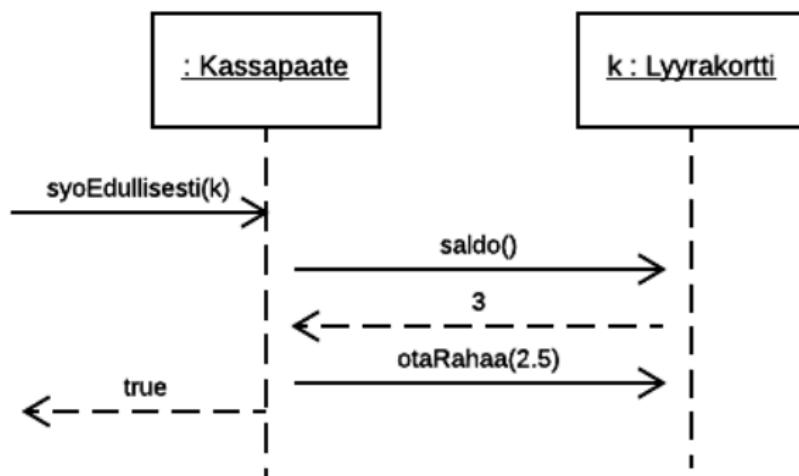
Koska järjestelmän toiminnan kulmakivenä on järjestelmän sisältämien olioiden yhteistyö, tarvitaan menetelmä yhteistyön kuvaamiseen

- ▶ UML tarjoaa kaksi menetelmää, joita kohta tarkastelemme:
 - ▶ sekvenssikaavio ja kommunikaatiokaavio

Sekvenssikaavio

Olioiden yhteistyön mallintaminen

- ▶ "Ostetaan 3 euroa sisältäväällä lyyrakortilla edullinen lounas"
 - ▶ Lukemalla koodia (ks. mallivastaus ohpe viikko 5) huomataan, että kassapääte kysyy ensin kortin saldon ja huomatessaan sen riittävän, vähentää kortilta edullisen lounaan hinnan
- ▶ Tilanteen kuvaava sekvenssikaavio:



Sekvenssikaavio

Olioiden yhteistyön mallintaminen

- ▶ Sekvenssikaaviossa kuvataan tarkasteltavan skenaarion aikana tapahtuva olioiden vuorovaikutus
- ▶ Oliot esitetään kuten oliokaaviossa, eli laatikkoina, joissa alleviivattuna olion nimi ja tyyppi
- ▶ Sekvenssikaaviossa oliot ovat (yleensä) ylhäällä rivissä
- ▶ Aika etenee kaaviossa alaspäin
- ▶ Jokaiseen olioon liittyy katkoviiva eli elämänviiva (engl. lifeline), joka kuvaaa sitä, että olio on olemassa
- ▶ Metodikutsu piirretään nuolena, joka lähtee kutsuvasta oliosta ja kohdistuu kutsuttavan olion elämänlankaan
- ▶ Tyypillisesti yksi sekvenssikaavio kuvaaa järjestelmän yksittäisen toimintaskenaarion

Sekvenssikaavio

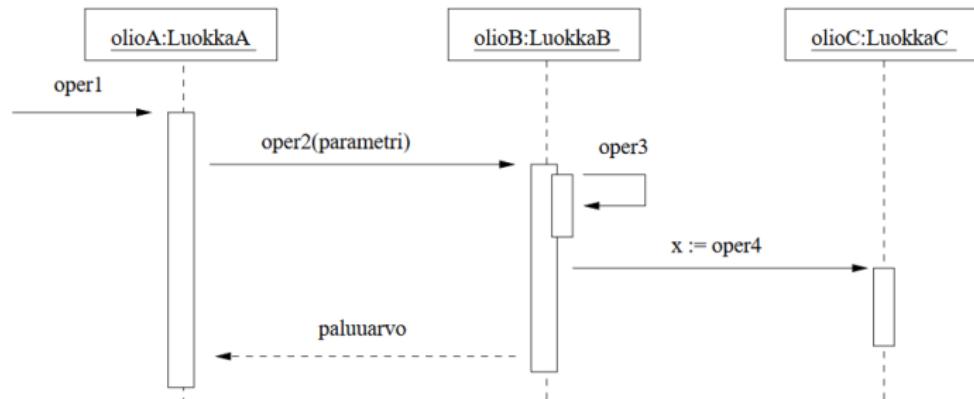
Olioiden yhteistyön mallintaminen

- ▶ Esimerkissä toiminta alkaa sillä, että joku (esim. pääohjelma, tässä tapauksessa nuoli on merkitty tulevan tyhjästä) kutsuu Kassapaate-olian metodia syoEdullisesti
- ▶ Metodikutsun seurauksena kassapääte kutsuu lyyrakortin metodia saldo, joka palauttaa kortilla olevan rahamäärään
 - ▶ Kortin palauttama saldo on merkitty katkoviivalla
 - ▶ Tämän jälkeen kassapääte kutsuu kortin metodia otaRaha, parametrilla 2.5 eli velottaa kortilta edullisen lounaan hinta
- ▶ Kun hinta on veloitettu, Kassapääte palauttaa operaation onnistumisen merkiksi true metoden syoEdullisesti kutsujalle
 - ▶ Metoden paluuarvo on jälleen merkitty katkoviivalla

Sekvenssikaavio

Olioiden yhteistyön mallintaminen

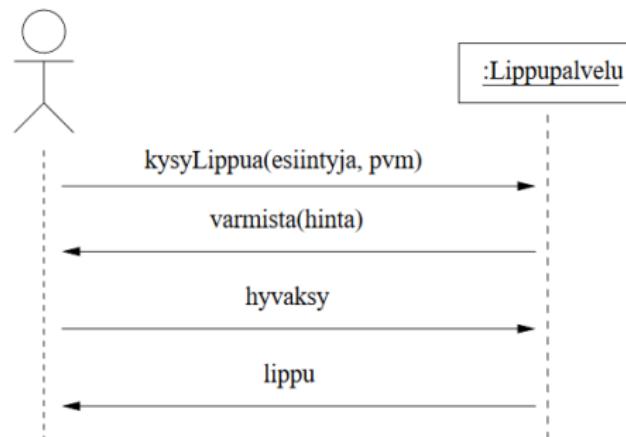
- ▶ Joskus on hyödyllistä piirtää **aktivaatiopalkki**, joka merkitsee ajan jolloin olio on aktiivisena, eli sen suoritus on kesken
- ▶ Aktivaatiopalkkia harvemmin jaksetaan piirtää paperille
- ▶ Merkinnällä $x := \text{oper4}$ tarkoitetaan, että metodia oper4() kutsutaan ja sen palautusarvo "napataan" muuttujaan x



Sekvenssikaavio

Lippuvaraus esimerkki

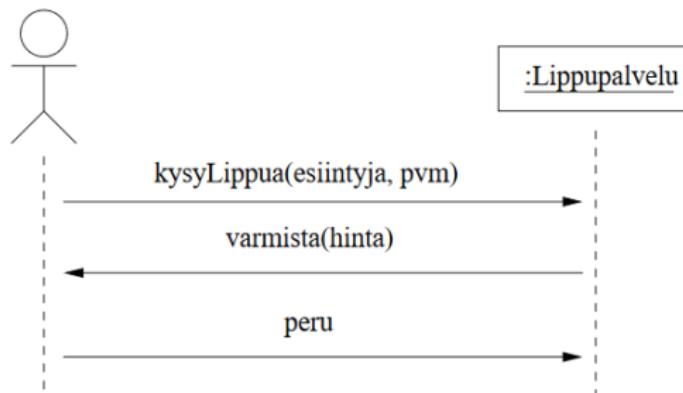
- ▶ Tarkastellaan alkeellista lippupalvelun tietojärjestelmää ja sen käyttötapausta
- ▶ Käyttötapaksen kulku:
 1. Käyttäjä kertoo tilaisuuden nimen ja päivämäärän
 2. Järjestelmä kertoo, minkä hintainen lippu on mahdollista ostaa
 3. Käyttäjä hyväksyy lipun
 4. Käyttäjälle annetaan tulostettu lippu



Sekvenssikaavio

Lippuvaraus esimerkki

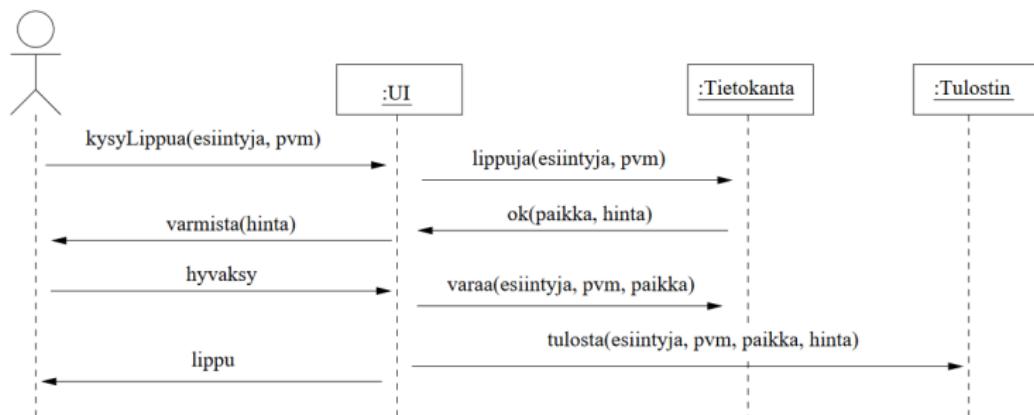
- ▶ Kuten näkyy voidaan sekvenssikaavioita käyttää myös oliosuunnittelussa!
 - ▶ Tästä voidaan saada helposti aikaan koodirunko
- ▶ Sekvenssikaaviot ovatkin usein käytössä oliosuunnittelun yhteydessä
- ▶ Jatketaan esimerkkiä, seuraavaksi järjestelmätason sekvenssikaaviona tilanne, jossa asiakas hylkää tarjotun lipun



Sekvenssikaavio

Lippuvaraus esimerkki

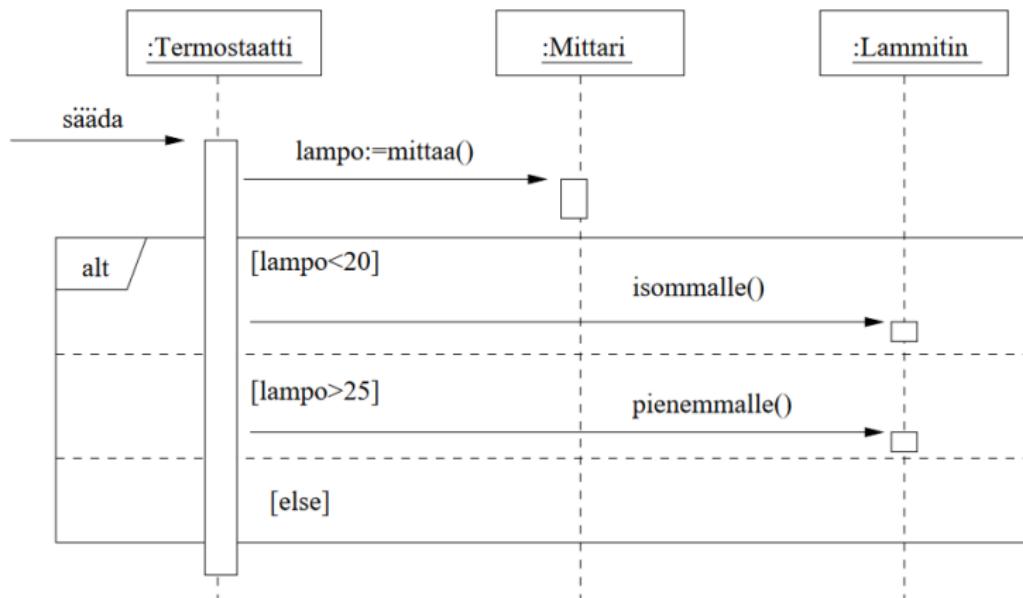
- ▶ Järjestelmätason sekvenssikaaviosta käy selkeästi ilmi käyttäjän ja järjestelmän interaktio
- ▶ Järjestelmän sisälle ei vielä katsota
- ▶ Seuraava askel on siirtyä suunnittelun ja tarkentaa miten käyttötapauksen skenaario toteutetaan suunniteltujen olioiden yhteistyönä



Sekvenssikaavio

Valinnaisuus

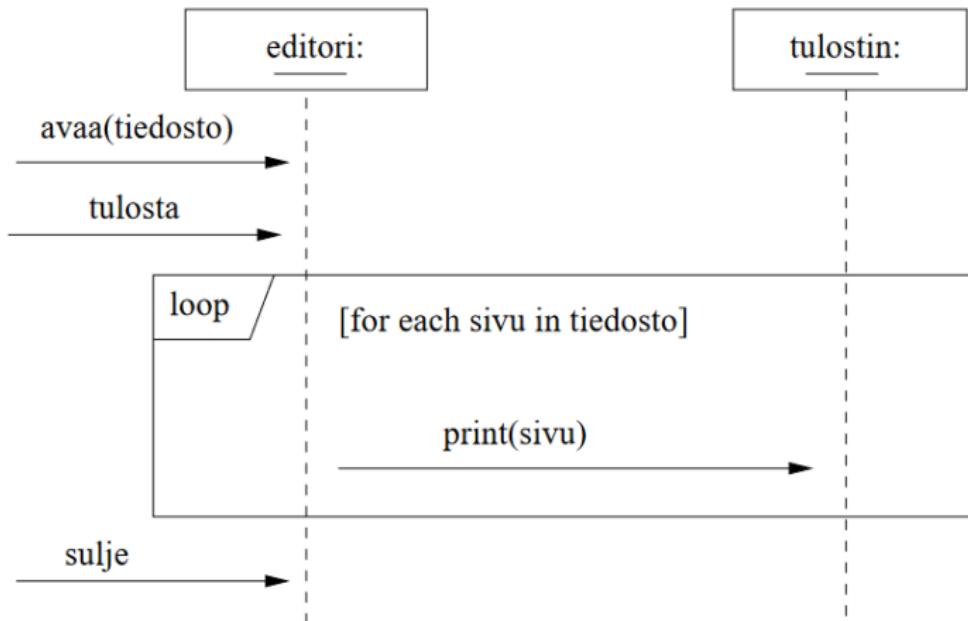
- ▶ Kaavioihin voidaan liittää lohko, jolla kuvataan valinnaisuutta
 - ▶ Vähän kuin if-else
 - ▶ Eli parametrina saadun arvon perusteella valitaan jokin kolmesta katkoviivan erottamasta alueesta



Sekvenssikaavio

Toisto

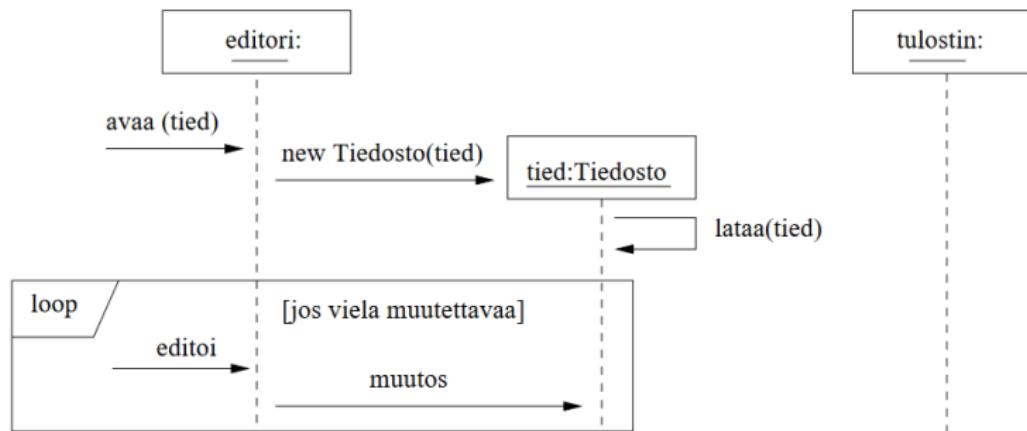
- ▶ Myös toistolohko mahdollinen (vrt. for tai while)
 - ▶ Huomaa miten toiston määrä on ilmaistu [ja] -merkkien sisällä
 - ▶ Voidaan käyttää myös vapaamuotoisempaa ilmausta, kuten "tulostetaan kaikki sivut erikseen"



Sekvenssikaavio

Olioiden luominen

- ▶ Esimerkki luentomonisteesta ⁵
- ▶ **Huom:** Uusi olio ei aloita ylhäältä vaan vasta siitä kohtaa milloin se luodaan

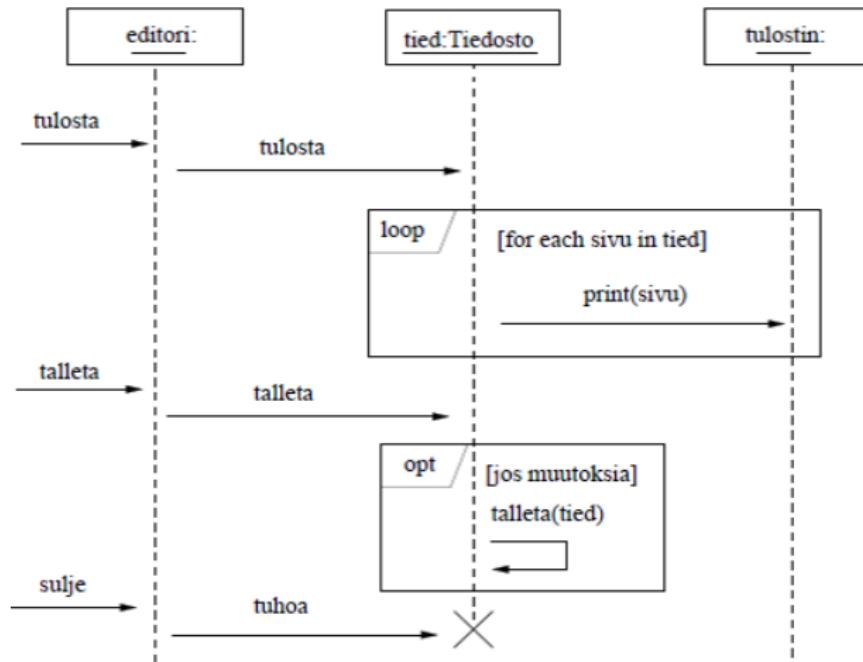


⁵Katso koko esimerkki sivulta 65

Sekvenssikaavio

Olioiden tuhoaminen

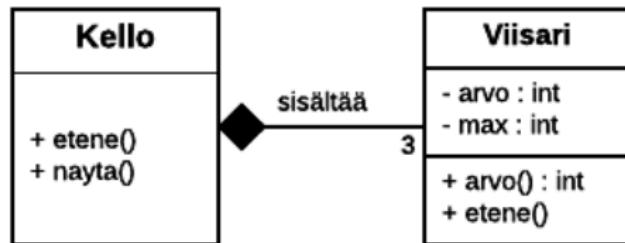
- ▶ Esimerkistä nähdään miten olion tuhoutuminen merkitään
- ▶ Mukana myös valinnainen (opt) lohko, joka suoritetaan jos ehto on tosi



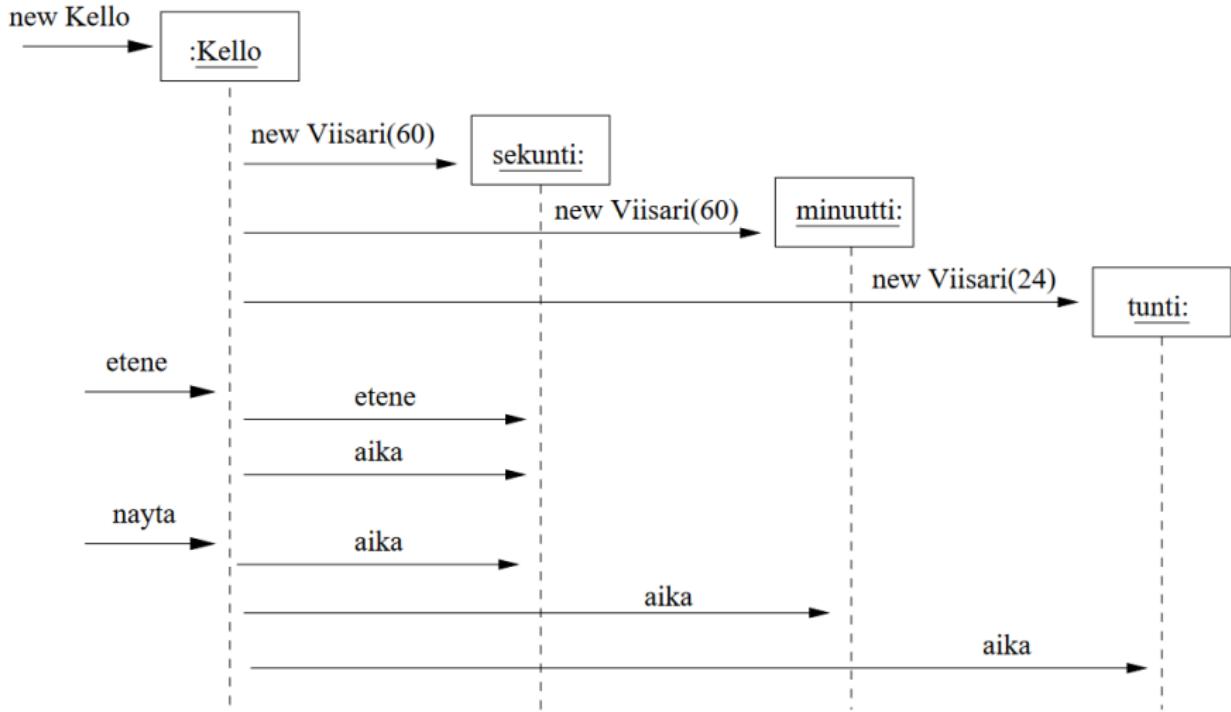
Sekvenssikaavio

Takaisinmallinnus

- ▶ Takaisinmallinnuksella (engl. *reverse engineering*) tarkoitetaan mallien tekemistä valmiina olevasta koodista
 - ▶ Erittäin hyödyllistä, jos esim. tarve ylläpitää huonosti dokumentoitua koodia
- ▶ Monisteesta löytyy Javalla toteutettu kello, joka nyt takaisinmallinnetaan
- ▶ **Luokkakaavio on helppo laatia:**



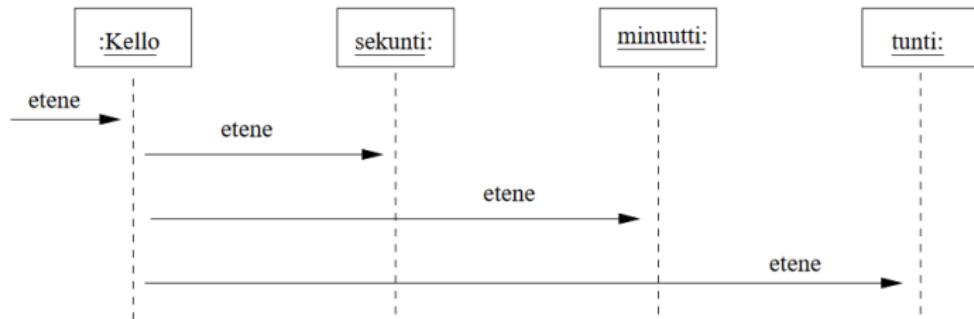
- ▶ Luokkakaaviosta ei vielä saa kuvaaa kellen toimintalogiikasta joten tarvitaan sekvenssikaavioita →



Sekvenssikaavio

Takaisinmallinnus

- ▶ Kaaviosta jätetty pois aika()-metodikutsut
- ▶ Samoin on jätetty pois Java-standardikirjaston out-oliolle suoritetut print()-metodikutsut
- ▶ **Eli jotta sekvenssikaavio ei kasvaisi liian suureksi, otetaan mukaan vain olennainen**
- ▶ Keskiyöllä kaikki viisarit pyörähtävät eli "etenevät" nollaan, tilannetta kuvaava sekvenssikaavio alla:



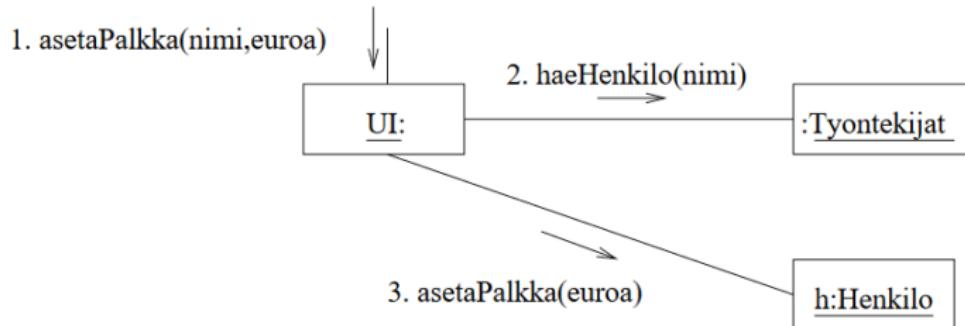
Kommunikaatiokaavio

Vaihtoehtoinen tapa kuvata yhteistoimintaa

Kommunikaatiokaavio

Vaihtoehtoinen tapa kuvata yhteistoimintaa

- ▶ Toinen tapa olioiden yhteistyön kuvaamiseen on kommunikaatiokaavio (engl. *communication diagram*)⁶
- ▶ Ohessa kalvon 150 Lippuvaraaus -esimerkki
- ▶ Metodien suoritusjärjestys ilmenee numeroinnista, sijoittelu on vapaa

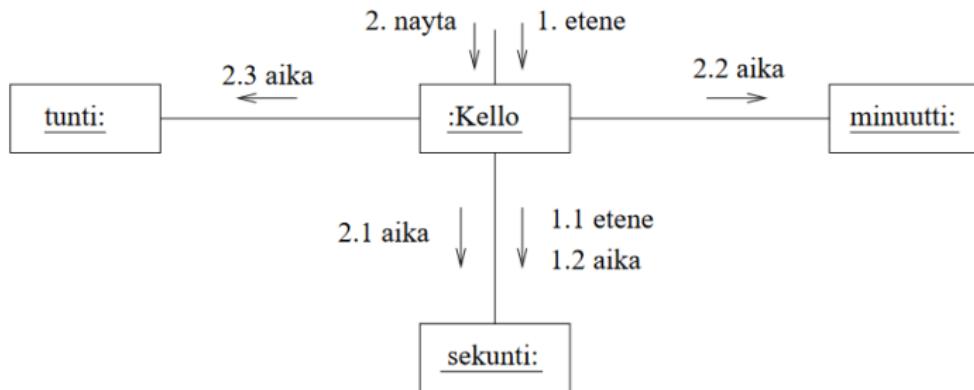


⁶Vanhalta nimitykseltä yhteistoimintakaavio (engl. *collaboration diagram*)

Kommunikaatiokaavio

Vaihtoehtoinen tapa kuvata yhteistoimintaa

- ▶ Viestien järjestyksen voi numeroida juoksevasti: 1, 2, 3, ...
- ▶ Tai allaolevan esimerkin⁷ tyyliin hierarkkisesti:
 - ▶ Kellole kutsutaan metodia etene(), tällä numero 1
 - ▶ Etenemisen aiheuttaa sekuntiviiasarille suoritetut metodikutsut etene() ja näytä(), nämä numeroitu 1.1 ja 1.2
 - ▶ Seuraavaksi kellole kutsutaan metodia näytä(), numero 2
 - ▶ Sen aiheuttamat metodikutsut numeroitu 2.1, 2.2, 2.3, ...



⁷Kello ja viisarit esimerkki

Kommunikaatiokaavio

Yhteenveto olioiden yhteistoiminnan kuvaamisesta

- ▶ Sekvenssikaavioita käytetään useammin kuin kommunikaatiokaavioita
 - ▶ Sekvenssikavio lienee luokkakaavioiden jälkeen eniten käytetty UML-kaaviotyyppi
- ▶ Sekä sekvenssi- että kommunikaatiokaavioilla tärkeä asema oliosuunnittelussa
- ▶ Kaaviot kannattaa pitää melko pieninä ja niitä ei kannata tehdä kuin järjestelmän tärkeimpien toiminnallisuksien osalta
 - ▶ Kommunikaatiokaaviot ovat yleensä hieman pienempiä, mutta toisaalta metodikutsujen ajallinen järjestys ei käy niistä yhtä hyvin ilmi kuin sekvenssikaavioista
- ▶ On epäselvää missä määrin sekvenssikaavioiden valinnaisuutta ja toistoa kannattaa käyttää
- ▶ Sekvenssikaaviot on alunperin kehitetty tietoliikeneprotokollien kuvaamista varten

Ohjelmistotekniikan menetelmät

Luento 4

Yhteyteen liittyvät tiedot, perintä, rajapinnat ja
oliosuunnittelun periaatteet

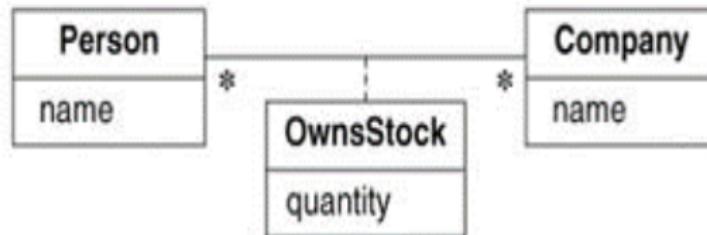
Yhteyteen liittyvät tiedot

Yhteyden tietojen mallinnus

Yhteyteen liittyvät tiedot

Yhteyden tietojen mallinnus

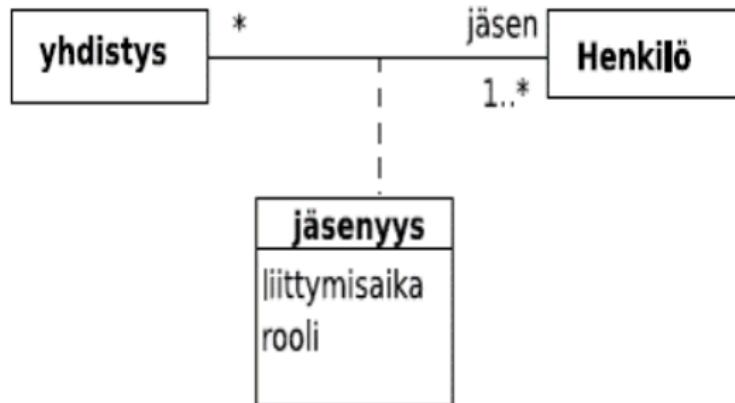
- ▶ Yhteyteen voi joskus liittyä myös tietoa
- ▶ Esim. tilanne missä henkilö voi olla (usean) yhtiön osakkeenomistaja
 - ▶ Osakkeenomistuksen kannalta tärkeä asia on omistettujen osakkeiden määrä
- ▶ Yksi tapa mallintaa tilanne on käyttää yhteysluokkaa (engl. association class), eli yhteyteen liittyvää luokkaa, joka sisältää esim. yhteyteen liittyviä tietoja
- ▶ Alla yhteysluokka sisältää omistettujen osakkeiden määrän



Yhteyteen liittyvät tiedot

Yhteyden tietojen mallinnus

- ▶ Luentomonisteessa mallinnetaan tilanne, jossa henkilö voi olla jäsenenä useassa yhdistyksessä
 - ▶ yhdistyksessä on vähintään 1 jäsen
- ▶ Jäsenyys kuvataan yhteytenä, johon liittyy yhteysluokka
 - ▶ jäsenyyden alkaminen (liittymisaika) sekä jäsenyyden tyyppi (rooli, eli onko rivijäsen, puheenjohtaja tms...) kuvataan yhteysluokan avulla



Yhteyteen liittyvät tiedot

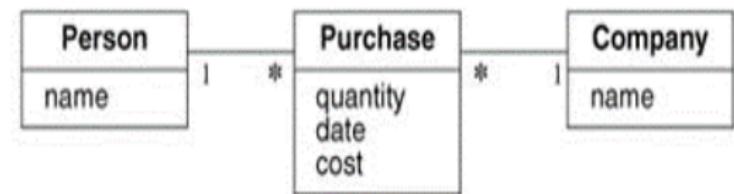
Kannattaako yhteysluokkia käyttää?

- ▶ Kannattaako yhteysluokkia käyttää?
 - ▶ Korkean tason abstrakteissa malleissa ehkä
 - ▶ Suunnittelutason malleissa todennäköisesti ei, sillä ei ole selvää, mitä yhteysluokka tarkoittaa toteutuksen tasolla
- ▶ Yhteysluokan voi aina muuttaa tavalliseksi luokaksi
- ▶ Yhteysluokka joudutaankin käytännössä aina ohjelmoidessa toteuttamaan omana luokkanaan, joka yhdistää alkuperäiset luokat joiden välillä yhteys on
 - ▶ Tämän takia yhteysluokkia ei välttämättä kannata käyttää alunperinkään

Yhteyteen liittyvät tiedot

Yhteysluokasta normaaliksi luokaksi

- ▶ Alla muutaman dian takainen osake-esimerkki
 - ▶ Nyt ilman yhteysluokkaa
- ▶ Henkilöllä on useita ostoksia (purchase)
- ▶ Ostokseen liittyy määrä (kuinka monesta osakkeesta kyse), päiväys ja hinta
- ▶ Yksi ostos taas liittyy tasan yhteen yhtiöön ja tasan yhteen henkilöön
- ▶ Henkilö ei ole enää suorassa yhteydessä firmaan
 - ▶ Henkilö "tuntee" kuitenkin omistamansa firmat ostosolioiden kautta



Yhteyteen liittyvät tiedot

Yhteysluokasta normaaliksi luokaksi

- ▶ Henkilön jäsenyys yhdistyksessä on myös luonteva kuvata yhteysluokan sijaan omana luokkanaan:

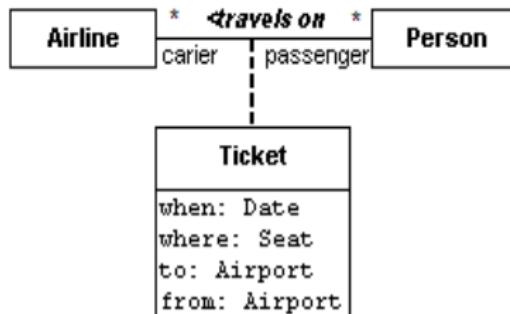


- ▶ Jäsenyyden olemassaoloriippuvuus on nyt merkitty henkilöön
- ▶ Miksi näin? Miksei yhdistykseen tai peräti molempien?
- ▶ Periaatteessa loogisinta olisi tehdä jäsenyydestä olemassaoloriippuvainen sekä yhdistyksestä että henkilöstä, mutta UML ei salli tätä (Sääntö 2: Osa voi kuulua vain yhteen kompositioon)
- ▶ Olemassaoloriippuvuus saa siis olla vain yhteen olioon ja hetken mietinnän jälkeen on päättetty valita Henkilö jäsenyyden "omistavaksi" osapuoleksi, periaatteessa Yhdistys olisi ollut yhtä hyvä valinta

Yhteyteen liittyvät tiedot

Kaksi olioja ja yhteyksien lukumäärä

- ▶ Kurssin kotisivulle linkitetyssä Holubin UML quick referenssessä on alla oleva esimerkki
- ▶ Eli henkilöllä voi olla travels on -yhteyksiä useiden lentoyhtiöiden kanssa
 - ▶ Yhteysluokkana Ticket on kerrottu matkan tiedot
- ▶ **Tähän malliin sisältyy ongelma:**
 - ▶ Henkilö voi olla travels on -yhteydessä moniin eri lentoyhtiöolioihin
 - ▶ Saman lentoyhtiöölön (esim. Finnair) kanssa ei kuitenkaan voi olla useampaa yhteyttä



Yhteyteen liittyvät tiedot

Kaksi olioja ja yhteyksien lukumäärä

Siis: jos luokkakaaviossa kahden luokan välillä on yhteys, voi kaksi luokkien olioja olla vain yhdessä yhteydessä kerrallaan!

- ▶ Olioiden välillä voi olla vain yksi yhteys, eli esim. Arto voi lentää Finnairilla vain kerran
- ▶ Tämä siitä huolimatta, että kytkentärajoitus on *
- ▶ Voi olla useita lippuja, mutta jokainen täytyy olla eri lentoyhtiöltä!
- ▶ Ratkaisu ongelmaan on kuvata yhteys omana luokkanaan
 - ▶ Henkilöllä voi olla useita lippuja
 - ▶ Lippu liittyy tiettyyn lentoyhtiöön ja tiettyyn henkilöön
 - ▶ Lentoyhtiön liittyy useita myytyjä lippuja



Perintä

Yleistys-erikoistus ja periminen

Perintä

Yleistys-erikoistus ja periminen

- ▶ Tähän mennessä tekemissämme luokkakaaviossa kaksi luokkaa ovat voineet liittyä toisiinsa muutamalla tapaa
- ▶ Yhteys ja kompositio liittyyvät tilanteeseen, missä luokkien olioilla on rakenteellinen (= jollain lailla pysyvä) suhde, esim.:
 - ▶ Henkilö omistaa Auton (yhteys: normaali viiva)
 - ▶ Huoneet sijaitsevat Talossa (kompositio: musta salmiakki)
 - ▶ = olemassaoloriippuuus, eli salmiakin toisen pää olemassaolo riippuu salmiakkipäässä olevasta
 - ▶ Jos talo hajotetaan, myös huoneet häviävät, huoneita ei voi siirtää toiseen taloon
- ▶ Löyhempi suhde on taas riippuuus, liittyy ohimenevämpiiin suhteisiin, kuten tilapäiseen käyttösuhteeseen, esim:
 - ▶ Autoton Henkilö käyttää Autoa (katkoviivanuoli)
- ▶ Tänään tutustumme yhteen hieman erilaiseen luokkien väliseen suhteeseen, eli **yleistys-erikostussuhteeseen**, jonka vastine ohjelmoinnissa on periminen

Perintä

Yleistys-erikoistus ja periminen

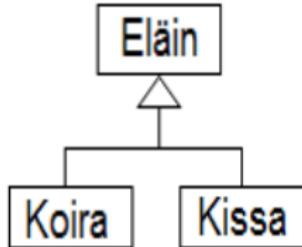
- ▶ Ajatellaan luokkia Eläin, Kissa ja Koira
- ▶ Kaikki Koira-luokan oliot ovat selvästi myös Eläin-luokan oliota, samoin kaikki
- ▶ Kissa-luokan oliot ovat Eläin-luokan olioita
- ▶ Koira-oliot ja Kissa-oliot ovat taas täysin eriäviä, eli mikään koira ei ole kissa ja päinvastoin
- ▶ Voidaan sanoa, että luokkien Eläin ja Koira sekä Eläin ja Kissa välillä vallitsee yleistys-erikoistussuhde:
 - ▶ Eläin on **yliluokka** (engl. *Superclass*)
 - ▶ Kissa ja Koira ovat eläimen **aliluokkia** (engl. *Subclass*)
- ▶ Yliluokka Eläin siis määrittelee mitä tarkoittaa olla eläin
 - ▶ Kaikkien mahdollisten eläinten yhteiset ominaisuudet ja toiminnallisuudet
- ▶ Aliluokassa, esim. Koira tarkennetaan mitä muita ominaisuuksia ja toiminnallisuutta luokan olioilla eli Koirilla on kuin yliluokassa Eläin on määritelty

Perintä

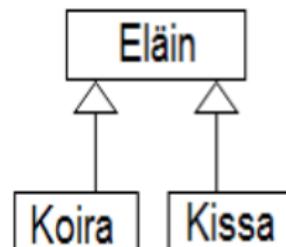
Yleistys-erikoistus ja periminen

Aliluokat siis perivät (engl. *inherit*) yliluokan ominaisuudet ja toiminnallisuuden

- ▶ Luokkakaaviossa yleistyssuhde merkitään siten, että aliluokasta piirretään yliluokkaan kohdistuva nuoli, jonka päässä on iso "valkoinen" kolmio
- ▶ Jos aliluokkia on useita, voivat ne jakaa saman nuolenpään tai molemmat omata oman nuolensa:



tai:



Perintä

Yleistys-erikoistus ja periminen

- ▶ Tarkkamuistisimmat huomaavat ehkä, että olemme jo törmänneet kurssilla yleistys-erikoisstussuhteeseen käyttötapausten yhteydessä
 - ▶ Sama valkoinen kolmiosymboli oli käytössä myös käyttötapausten yleistyksen yhteydessä
 - ▶ Yleistetty käyttötapaus⁸ opetustarjonnan ylläpito erikoistui kurssin perustamiseen, laskariryhmän perustamiseen ym..
 - ▶ Erikoistus myös mainittu edellisen yhteydessä

Luokkien välinen yleistys-erikoistussuhde eli yli- ja aliluokat toteutetaan ohjelmointikielissä siten, että aliluokka perii yliluokan

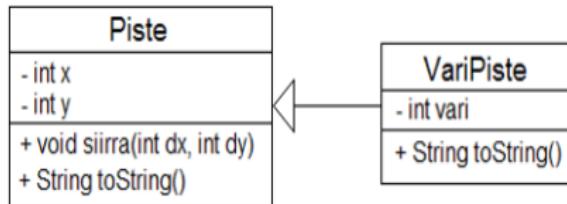
⁸Katso dia numero 55 luennolta 1.

Perintä

Yleistys-erikoistus ja periminen

```
public class Piste{  
    private int x, y;  
  
    public void siirra(int dx, int dy) {  
        x+=dx;  
        y+=dy;  
    }  
  
    public String toString(){  
        return "(" + x + ")"  
    }  
}
```

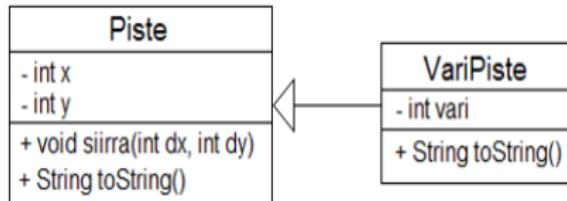
```
public class VariPiste extends Piste {  
    private int vari;  
  
    public String toString(){  
        return super.toString() + "c: " + vari;  
    }  
}
```



Perintä

Yleistys-erikoistus ja periminen

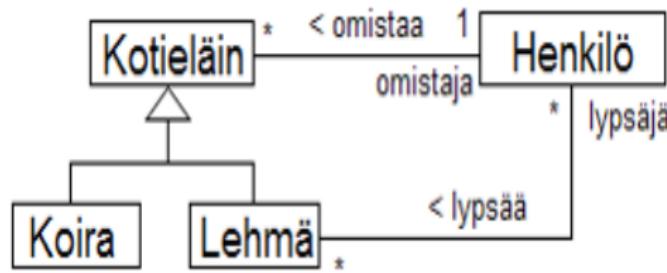
- ▶ Yliluokan Piste attribuutit x ja y sekä metodi siirra() siis periytyvät aliluokkaan VariPiste
 - ▶ Periytyviä attribuutteja metodeja ei merkitä aliluokan kohdalle
- ▶ Jos ollaan tarkkoja, Piste-luokan metodi toString periytyy myös VariPiste-luokalle, joka syrjäyttää (engl. *override*) perimänsä omalla toteutuksella
 - ▶ Korvaava toString()-metodi merkitään aliluokkaan VariPiste
- ▶ Eli kuviosta on pääteltäväissä, että VariPisteellä on:
 - ▶ Attribuutit x ja y sekä metodi siirra perittynä
 - ▶ Attribuutti vari, jonka se määrittelee itse
 - ▶ Määritelty metodi toString joka syrjäyttää yliluokalta perityn
 - ▶ Koodista nähdään, että korvaava metodi käyttää yliluokassa määriteltyä metodia



Perintä

Mitä kaikkea periytyy?

- ▶ Luokat Koira ja Lehmä ovat molemmat luokan Kotieläin aliluokkia
- ▶ Jokaisella kotieläimellä on omistajana joku Henkilö-olio
- ▶ Koska omistaja liittyy kaikkiin kotieläimiin, merkitään yhteys Kotieläin- ja Henkilö-luokkien välille
 - ▶ **Yhteydet periytyvät aina aliluokille**, eli Koira-olioilla ja Lehmä-olioilla on omistajana yksi Henkilö-olio
- ▶ Ainoastaan Lehmä-olioilla on lypsääjiä
 - ▶ Yhteys lypsää tuleekin Lehmän ja Henkilön välille

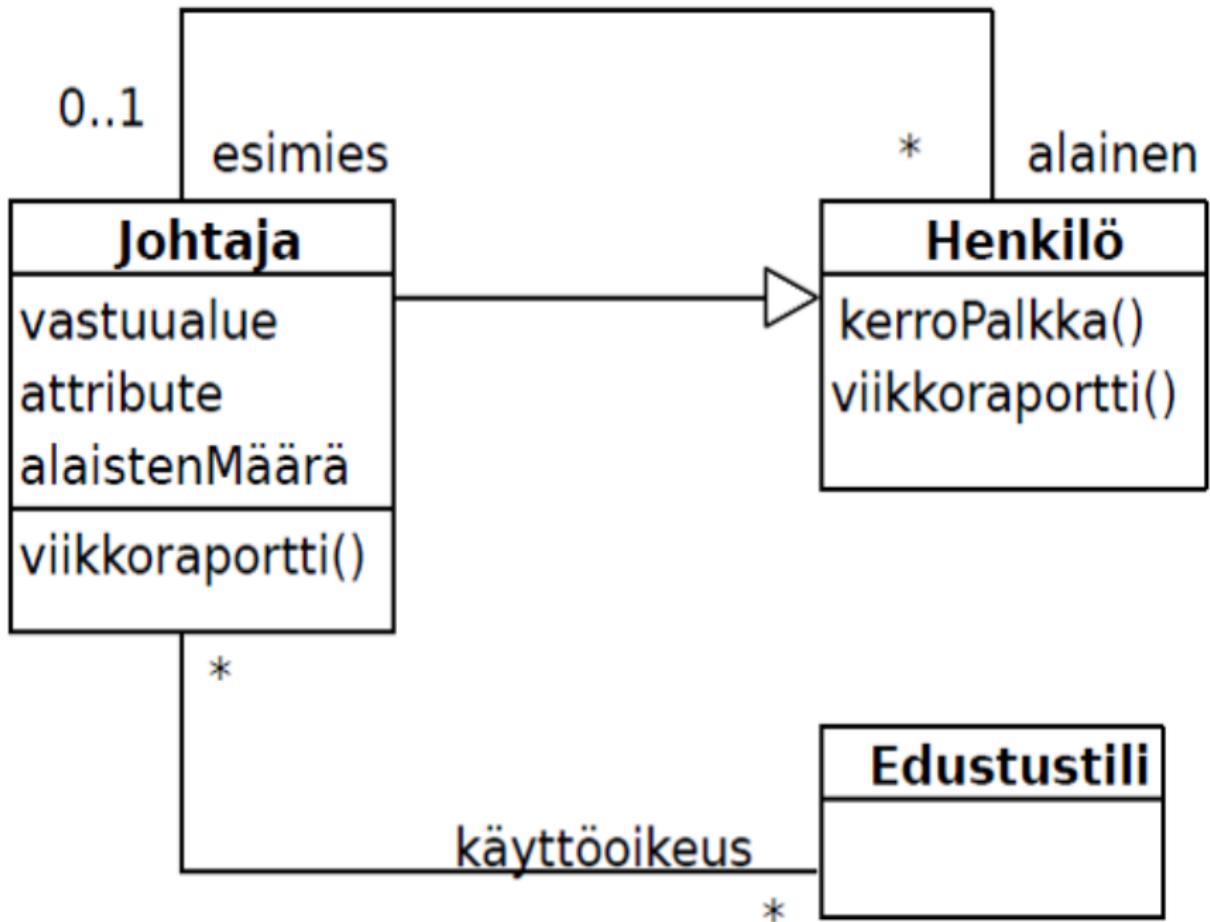


Perintä

Aliluokan ja yliluokan välinen yhteys

Otetaan taas esimerkki:

- ▶ Yrityksen työntekijää kuvaaa luokka Henkilö Henkilöllä on metodit kerroPalkka() ja viikkoraportti()
 - ▶ Henkilöllä on metodit kerroPalkka() ja viikkoraportti()
- ▶ Johtaja on Henkilön aliluokka
 - ▶ Johtajalla on alaisena useita henkilöitä
 - ▶ Henkilöllä on korkeintaan yksi johtaja esimiehenä
 - ▶ Johtajalla voi olla käyttöoikeuksia Edustustileihin
 - ▶ Edustustilillä on useita käyttöoikeuden omaavia johtajia
 - ▶ Johtajan viikkoraportti on erilainen kuin normaalien työntekijän viikkoraportti
- ▶ Tilannetta kuvaava luokkakaavio seuraavalla sivulla



Perintä

Aliluokan ja yliluokan välinen yhteys

- ▶ Johtaja siis perii **kaiken** Henkilöltä
 - ▶ Henkilö on alainen-roolissa yhteydessä nollaan tai yhteen Johtajaan
 - ▶ Tästä seuraa, että myös Johtaja-olioilla on sama yhteys, eli myös johtajilla voi olla johtaja!
- ▶ Metodi **viikkoraportti()** on erilainen johtajalla kuin muilla henkilöillä, siispä Johtaja-luokka korvaa Henkilö-luokan metodin omallaan
- ▶ Esim. Henkilö-luokan metodi **viikkoraportti()**:
 - ▶ Kerro ajankäyttö työtehtäviin
- ▶ Johtaja-luokan korvaama metodi **viikkoraportti()**:
 - ▶ Kerro ajankäyttö työtehtäviin
 - ▶ Laadi yhteenvetö alaisten viikkoraporteista
 - ▶ Raportoi edustustilin käytöstä
- ▶ Yhteys käyttöoikeus Edustustileihin voi siis ainoastaan olla niillä henkilöillä, jotka ovat johtajia

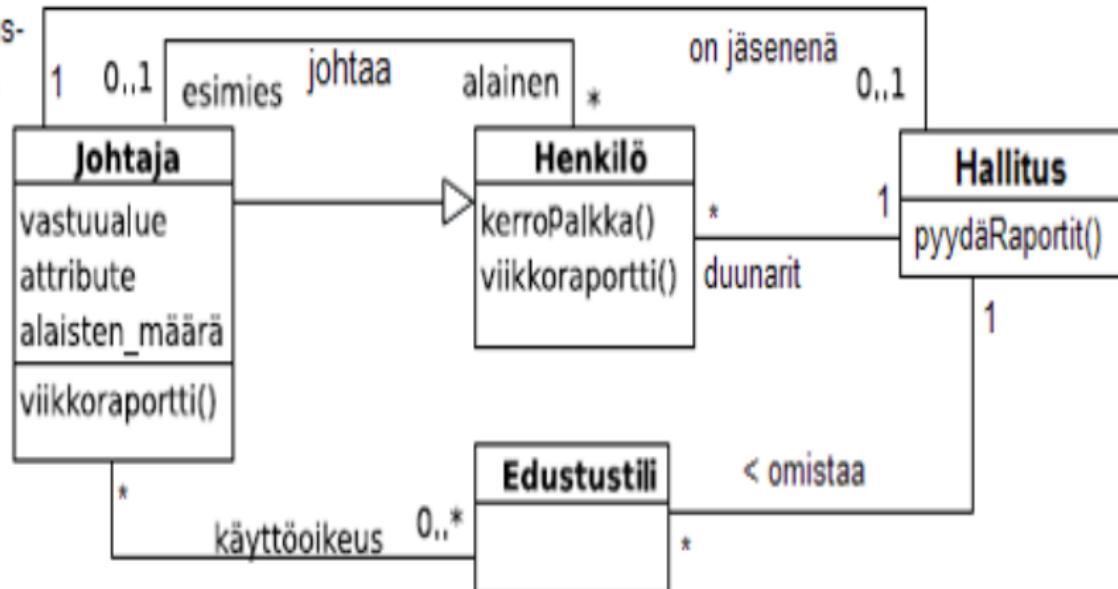
Perintä

Aliluokan ja yliluokan välinen yhteys

Laajennetaan mallia:

- ▶ Yrityksen hallitus koostuu ulkopuolisista henkilöistä (joita ei sisällytetä malliin) ja yrityksen toimitusjohtajasta, joka siis kuuluu henkilöstöön
 - ▶ Hallitus on edustustilien omistaja
 - ▶ Hallitus "tuntee" toimitusjohtajaa lukuunottamatta kaikki työntekijät, myös normaalit johtajat ainoastaan Henkilö-olioina
- ▶ Hallitus pyytää työntekijöiltä viikkoraportteja
 - ▶ Viikkoraportin tekevät kaikki paitsi toimitusjohtaja

toimitus-
johtaja



Perintä

Olio tietää luokkansa

- ▶ Hallitus siis tuntee kaikki työntekijänsä, mutta ei erittele ovatko he normaaleja työntekijöitä vai johtajia
 - ▶ Hallituksen koodissa kaikkia työntekijöitä pidetään Henkilö-oliosta koostuvassa listalla duunarit. Johtajathan ovat myös henkilötä!
- ▶ Hallituksen ei siis ole tarvetta tuntea kuka on johtaja ja kuka ei
- ▶ Pyytäessään viikkoraporttia, hallitus käsittelee kaikkia samoin:
- ▶ Jokainen duunari tuntee "oikean" luokkansa
- ▶ Kun hallitus kutsuu duunarille metodia viikkoraportti(), jos kyseessä on normaali henkilö, suoritetaan henkilön viikkoraportointi, jos taas kyseessä on johtaja, suoritetaan johtajan viikkoraportti (polymorfismia!)

```
public class Hallitus{
    private ArrayList< Henkilo > duunarit;
    private Johtaja toimitusjohtaja;

    public void pyydaRaportit(){
        for ( d : duunarit ) {
            if ( d != toimitusjohtaja ) {
                d.viikkoraportti()
            }
        }
    }
}
```

Abstraktit luokat ja rajapinnat

Abstraktit luokat

Abstraktit luokat ja rajapinnat

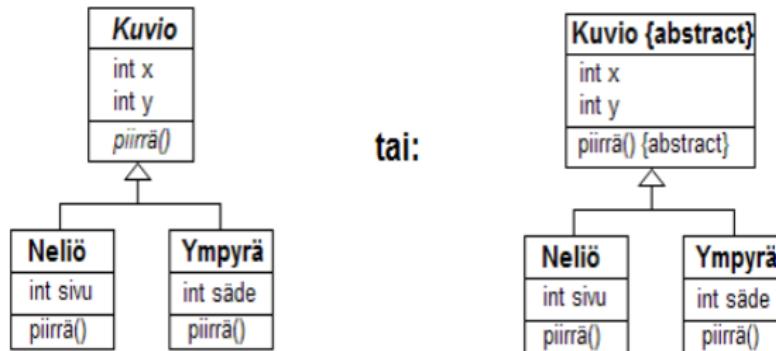
Abstraktit luokat

- ▶ Yliluokalla Kuvio on sijainti, joka ilmaistaan x- ja y-kordinaatteina sekä metodi piirrä()
- ▶ Kuvion aliluokkia ovat Neliö ja Ympyrä
 - ▶ Neliöllä on sivun pituus ja Ympyrällä säde
- ▶ Kuvio on nyt pelkkä abstrakti käsite, Neliö ja Ympyrä ovat konkreettisia kuvioita jotka voidaan piirtää ruudulle kutsumalla sopivia grafiikkakirjaston metodeja
- ▶ Kuvio onkin järkevä määritellä **abstraktiksi luokaksi**, eli luokaksi josta ei voi luoda instansseja, joka ainoastaan toimii sopivana yliluokkana konkreettisille kuvioille
- ▶ Kuviolla on attribuutit x ja y, mutta metodi piirrä() on abstrakti metodi, eli Kuvio ainoastaan määrittelee metodin nimen ja parametrien sekä paluuarvon tyypit, mutta metodille ei anneta mitään toteutusta
- ▶ Kuvion perivät luokat Neliö ja Ympyrä antavat toteutuksen abstraktille metodille

Abstraktit luokat ja rajapinnat

Abstraktit luokat

- ▶ Luokkakaaviossa on kaksi tapaa merkitä abstraktius
 - ▶ Abstraktin luokan/metodin nimi kursiivilla, tai
 - ▶ liitetään abstraktin luokan/metodin nimeen tarkenne abstract



- ▶ `public abstract class Kuvio{`
- ▶ `public abstract void piirrä();`

Abstraktit luokat ja rajapinnat

Rajapinnat

Abstraktit luokat ja rajapinnat

Rajapinnat

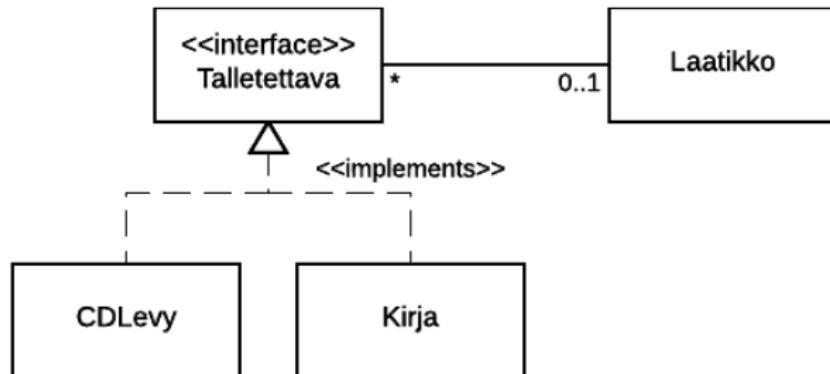
- ▶ Javan rajapinta (engl. *interface*) on ikäänsuin abstrakti luokka, joka ei sisällä attribuutteja ja jossa (useimmiten) kaikki metodit ovat abstrakteja
- ▶ Rajapintaluokka siis (yleensä) listaa ainoastaan joukon metodien nimiä
 - ▶ Java 8 on tuonut tähän sen poikkeuksen, että rajapintojen metodeilla voi olla oletustoteutuksia
- ▶ Yksi luokka voi toteuttaa useita rajapintoja
 - ▶ Ja sen lisäksi vielä periä yhden luokan
- ▶ Perimällä luokka saa yliluokasta attribuutteja ja metodeja
- ▶ Rajapinnan toteuttaminen on pikemminkin velvollisuus
 - ▶ Jos luokka toteuttaa rajapinnan, sen täytyy toteuttaa kaikki rajapinnan määrittelemät metodit (paitsi ne joilla on oletustoteutus)

Rajapinta on sopimus, jonka toteuttaja lupaa toteuttaa ainakin rajapinnan määrittelemät metodit.

Abstraktit luokat ja rajapinnat

Esimerkkejä rajapinnoista

- ▶ Mallinnetaan Ohjelmoinnin jatkokurssin toisen viikon tehtävä Tavaroida ja Laatikoita ⁹
- ▶ Rajapintaluokka kuvataan luokkana, johon liitetään tarkenne <<interface>>
- ▶ Rajapinnan toteuttaminen merkitään kuten periminen, mutta katkoviivana
 - ▶ Voidaan tarkentaa tarkenteella «implements»



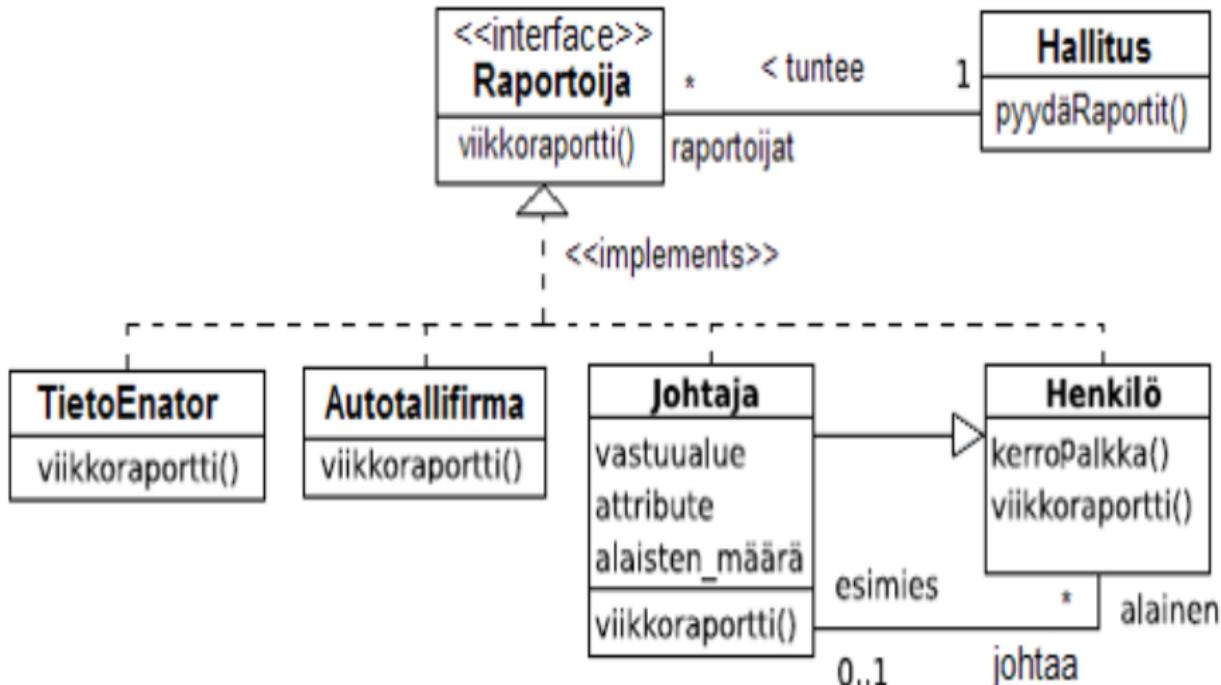
⁹Tehtävä numero 136

Abstraktit luokat ja rajapinnat

Esimerkkejä rajapinnoista

- ▶ Palataan muutaman dian takaiseen yritysesimerkkiin ¹⁰
- ▶ Tilanne on nyt se, että yritys on ulkoistanut osan toiminnoistaan
- ▶ Hallitus on edelleen kiinnostunut viikkoraporteista
 - ▶ Hallitusta ei kuitenkaan kiinnosta se, tuleeko viikkoraportti omalta henkilöstöltä vai alihankkijalta
- ▶ Muuttuneessa tilanteessa hallitus tunteekin ainoastaan joukon raportointiin kykeneviä olioita
 - ▶ Jotka voivat olla Henkilötä, Johtajia tai alihankkijoita
 - ▶ Kukin näistä toteuttaa metodin `viikkoraportti()` omalla tavallaan
- ▶ Tilanne kannattaa hoitaa määrittelemällä rajapinta ja vaatia, että kaikki hallituksen tuntemat tahot toteuttavat rajapinnan
 - ▶ `public interface` Raportoija
- ▶ Hallitukselle riittääkin, että se tuntee joukon Raportoijia (eli rajapinnan toteuttajia)

¹⁰Katso dia numero 181

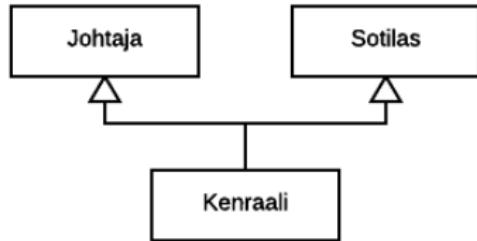


Moniperintä
Lisää perintää, monesti!

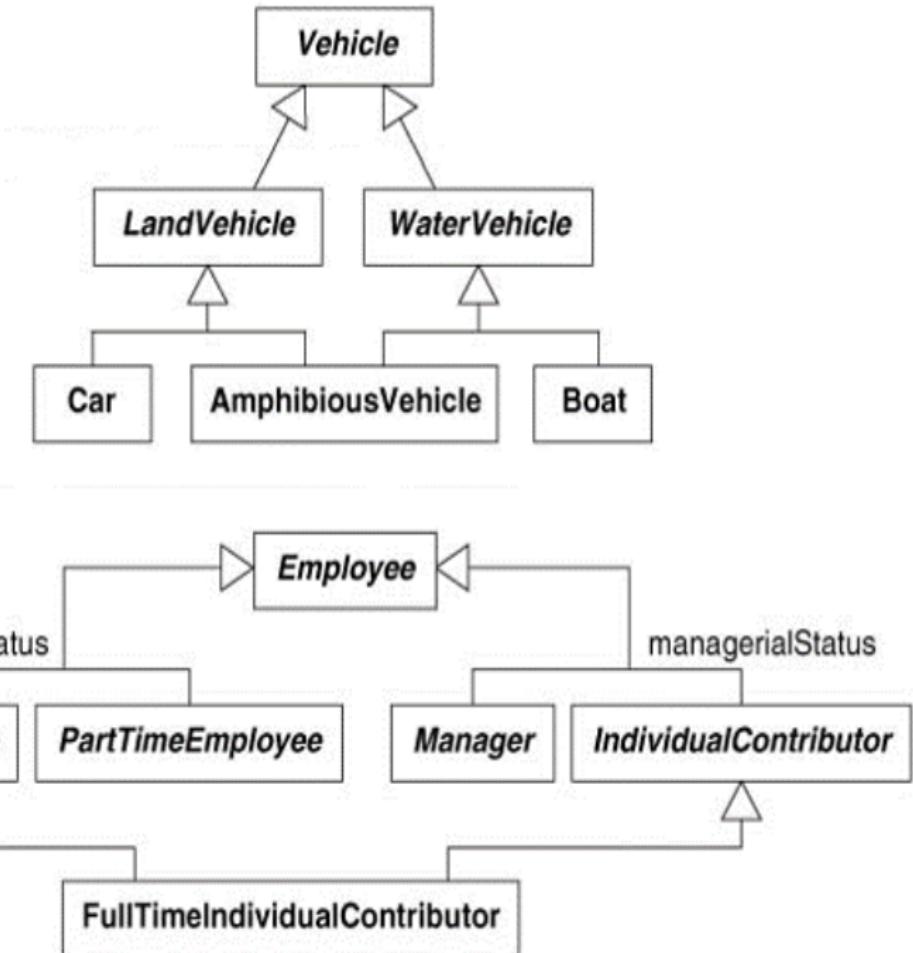
Moniperintä

Lisää perintää, monesti!

- ▶ Joskus tulee esiin tilanteita, joissa yhdellä luokalla voisi kuvitella olevan useita yliluokkia
- ▶ Esim. kenraalilla on sekä sotilaan, että johtajan ominaisuudet



- ▶ Kyseessä moniperintä (engl. *multiple inheritance*)
- ▶ Esim. Kulkuneuvo jakautuu maa- ja merikulkuneuvoksi - Auto on maakulkuneuvo, vene merikulkuneuvo, amfibio sekä maa- että merikulkuneuvo
- ▶ Esim. Työntekijät voi jaotella kahdella tavalla: Pää- ja sivutoimiset ja Johtajat ja normaalit (Yksittäinen työntekijä voi sitten olla esim. päätoiminisen johtaja)



Moniperintä

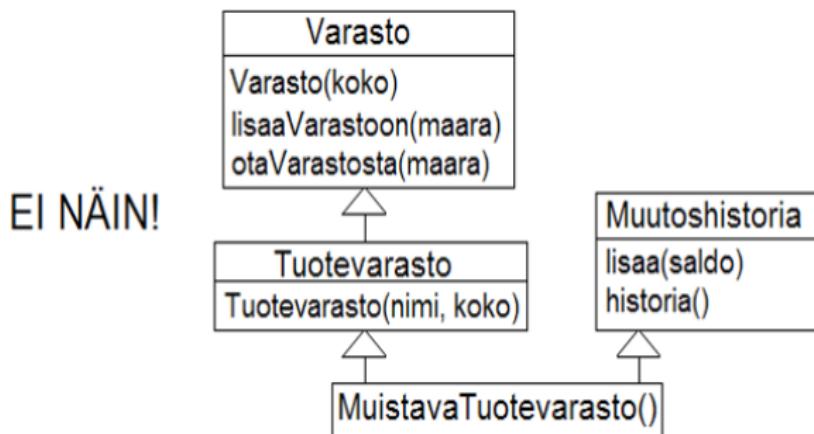
Moniperinnän ongelmat

- ▶ Moniperintä on monella tapaa ongelmallinen asia ja useat kielet, **kuten Java eivät salli moniperintää**
 - ▶ C++ sallii moniperinnän
 - ▶ "moderneissa" kielissä kuten Python, Ruby ja Scala on olemassa ns. mixin-mekanismi, joka mahdollistaa "hyvinkäytätyvän" moniperintää vastavan mekanismin
- ▶ Monissa tilanteissa onkin viisasta olla käyttämättä moniperintää ja yrittää hoitaa asiat muin keinoin
- ▶ Näitä muita keinoja (jos unohdetaan mixin-mekanismi) ovat:
 - ▶ Moniperiytyksen korvaaminen yhteydellä, eli käytännössä "liittämällä" olioon toinen olio, joka laajentaa alkuperäisen olion toiminnallisuutta
 - ▶ Javan rajapinnan toimivat joissain tapauksessa moniperinnän korvikkeena varsinkin kun rajapinnat tukevat Java 8:n ilmestymisen jälkeen metodien oletusarvoisia toteutuksia

Moniperintä

Muistava tuotevarasto esimerkki

- ▶ Ohjelmoinnin jatkokurssin viikon 4 (tai viikon 10) laskareissa ¹¹
- ▶ Toteutetaan Muistava Tuotevarasto, joka on luokka, johon lisätään toiminnallisuutta perimisen sijaan liittämällä siihen toinen olio
- ▶ Joku C++-ohjelmoija voisi soveltaa tilanteessa moniperintää:

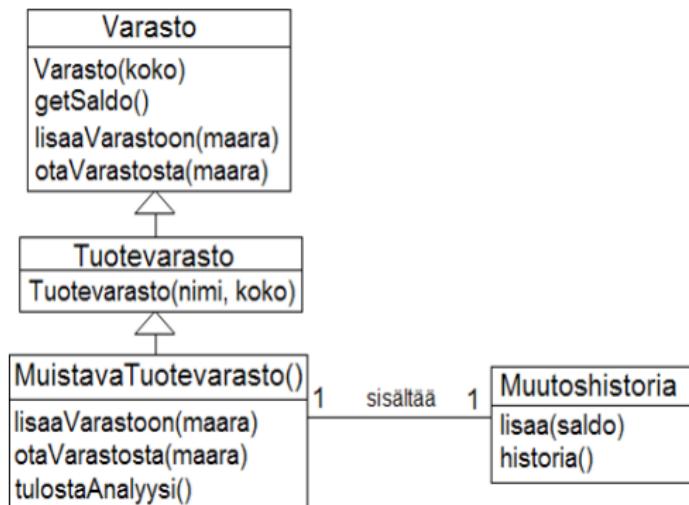


¹¹Tehtävä numero 155

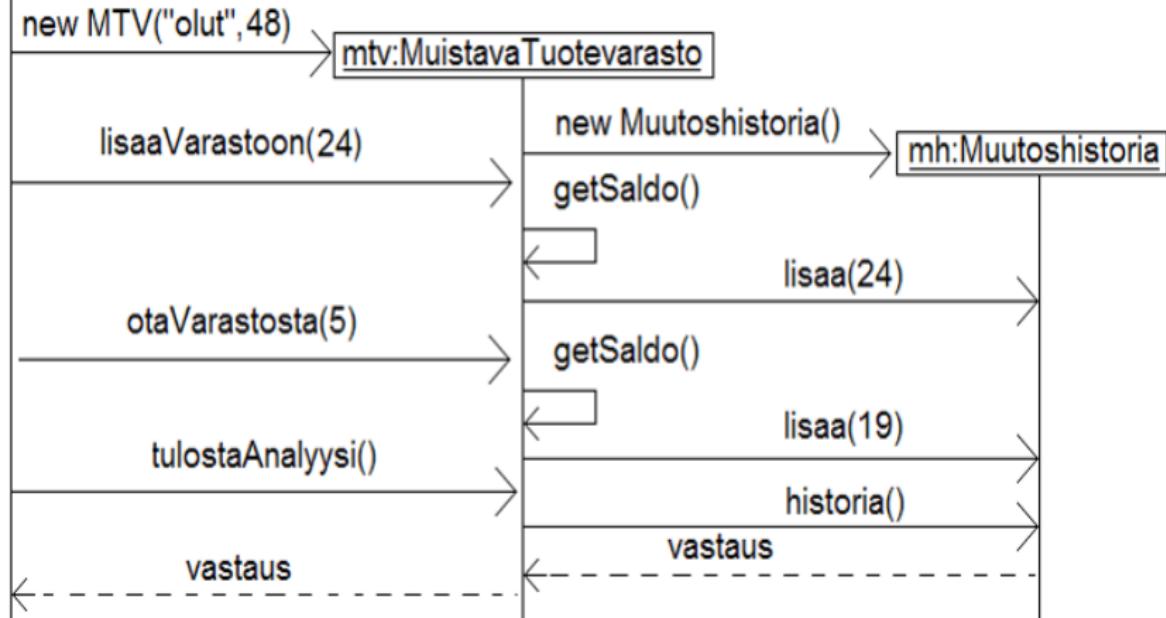
Moniperintä

Muistava tuotevarasto esimerkki

- ▶ Javassa ei moniperintää ole, ja vaikka olisikin, on parempi liittää "muistamistoiminto" muistavaan tuotevarastoon erillisenä oliona:
- ▶ Aina kun muistavan tuotevaraston saldo päivitettyy (metodien lisaaVarastoon ja otaVarastosta yhteydessä), samalla laitetaan uusi saldo muutoshistoriaan



sovellus:



```
public MuistavaTuotevarasto extends Tuotevarasto {  
    private Muutoshistoria varastotilanteet;  
  
    public MuistavaTuotevarasto(String tuote, double koko){  
        super(tuote, koko);  
        varastotilanteet = new Muutoshistoria();  
    }  
  
    public void lisaaVarastoon(double maara){  
        super.lisaaVarastoon(maara);  
        double saldo = getSaldo();  
        varastotilanteet.lisaa(saldo);  
    }  
  
    public String tulostaAnalyysi() {  
        return varastotilanteet.historia();  
    }  
}
```

Oliosuunnittelun periaatteita

Miten olioita tulisi käyttää?

Oliosuunnittelun periaatteita

Miten olioita tulisi käyttää?

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

Oliosuunnittelun periaatteita

Miten olioita tulisi käyttää?

- ▶ Aikojen saatossa on huomattu, että sisäiseltä laadultaan hyvissä ohjelmissa on tiettyjä samankaltaisia piirteitä, ja näitä tutkimalla on päädytty joukkoon hyvän oliosuunnittelun periaatteita
- ▶ **Periaatteita on useita, tarkastellaan tänään näitä neljää:**
 - ▶ Single responsibility principle
 - ▶ Favour composition over inheritance
 - ▶ Program to an interface, not to an Implementation
 - ▶ Riippuvuuksien minimointi

Oliosuunnittelun periaatteita

Single responsibility principle

- ▶ **Single responsibility** tarkoittaa karkeasti ottaen, että **oliolla tulee olla vain yksi vastuu** eli yksi asiakokonaisuus, mihin liittyvästä toiminnasta luokka itse huolehtii
- ▶ Robert C. Martin:
"A class should have only one reason to change."
- ▶ Äskeinen esimerkkimme MuuttuvaTuotevarasto toteuttaa periaatetta, sillä sen vastuulla on vain varaston nykyisen tilanteen ylläpito
- ▶ Se delegoi vastuun aikaisempien varastosaldojen muistamisesta Muutoshistoria-oliolle

Oliosuunnittelun periaatteita

Favour composition over inheritance

- ▶ **Favour composition over inheritance** eli suosi yhteistoiminnassa toimivia olioita perinnän sijaan
- ▶ Perinnällä on paikkansa, mutta sitä tulee käyttää harkiten!
- ▶ Muistava Tuotevarasto käyttää myös perintää järkevästi
 - ▶ Jos olisi moniperitty Tuotevarasto ja Muutoshistoria, olisi muodostettu luokka, joka rikkoo single responsibility – eli yhden vastuun periaatteen!

Oliosuunnittelun periaatteita

Program to an interface, not to an Implementation

- ▶ "Program to an interface, not to an Implementation", eli... ohjelmoi käyttämällä rajapintoja äläkä konkreettisia implementaatioita
- ▶ Laajennettavuuden kannalta ei ole hyvä idea olla riippuvainen konkreettisista luokista, sillä ne saattavat muuttua
- ▶ Parempi on tuntea vain rajapintoja (tai abstrakteja luokkia) ja olla tietämätön siitä mitä rajapinnan takana on
- ▶ Tämä mahdollistaa myös rajapinnan takana olevan luokan korvaamisen kokonaan uudella luokalla
- ▶ Esim. aluksi tuetaan XML, mutta myöhemmin halutaan tuke JSON ja CSV

Oliosuunnittelun periaatteita

Riippuvuuksien minimointi

- ▶ **Minimoi riippuvuudet**, eli älä tee *spagettikoodia*, jossa kaikki olioit tuntevat toisensa
- ▶ Pyri elimonoinaan riippuvuudet siten, että luokat tuntevat mahdollisimman vähän muita luokkia, ja mielellään nekin vain rajapintojen kautta (muista edellinen dia!)
- ▶ Tässä ylenesä auttaa kun noudattaa suunnittelumalleja ¹² (engl. *design patterns*)
- ▶ Piirtämällä malleja voidaan tunnistaa "ongelmakohtia"

¹²Esimerkkejä Java-maailmasta:

<https://github.com/iluwatar/java-design-patterns>

Oliosuunnittelun periaatteita

Kannattaako periaatteita noudattaa?

- ▶ Onko näissä periaatteissa järkeä? Kyllä, sillä niiden noudattaminen lisäävät ohjelmien ylläpidettävyyttä
- ▶ Kannattaako periaatteita noudattaa: useimmiten
 - ▶ joskus kuitenkin voi olla jonkun muun periaatteen nojalla viisasta rikkoa jotain toista periaatetta...
 - ▶ Jos kyseessä "kertakäyttökoodi", ei luonnollisesti kannata panostaa ylläpidettävyyteen
- ▶ "ikiaikaisia periaatteita", motivaationa ohelman muokattavuuden, uusiokäytettävyyden ja testattavuuden parantaminen
- ▶ Huonoa oliosuunnittelua on verrattu velan (engl. *technical debt*) ottamiseen
- ▶ Piittaamattomalla ja laiskalla ohjelmoinnilla/suunnittelulla saadaan ehkä nopeasti aikaan jotain, mutta hätäinen ratkaisu tullaan maksamaan korkoineen takaisin myöhemmin jos ohjelmaa on tarkoitustaan laajentaa tai muuttaa
- ▶ Joissain tilanteissa tosin velan ottaminen voi kannattaa

Oliosuunnittelun periaatteita

Oliosuunnittelun tärkeitä nimiä



Erich Gamma



Robert Martin



MY REFACTORINGS
LET ME SHOW YOU THEM

Martin Fowler



Kent Beck

Ohjelmistotekniikan menetelmät

Luento 5

Koodihajut, refaktoriointi, TDD, Järjestelmätason sekvenssikaaviot ja Ohjelmiston arkkitehtuuri

Koodihajut

Koodi haisee: merkki huonosta suunnittelusta

Koodihajut

Koodi haisee: merkki huonosta suunnittelusta

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

Koodihajut

Eriaisia koodihajuja

- ▶ **Koodihajuja on hyvin monenlaisia ja monentasoisia**
- ▶ Aloittelijankin on hyvä oppia tunnistamaan ja välttämään tavanomaisimpia
- ▶ Muutamia esimerkkejä hajuista:
 - ▶ Duplicated code (Copy&Paste -koodia)
 - ▶ Methods too big
 - ▶ Classes with too many instance variables
 - ▶ Classes with too much code
 - ▶ Uncommunicative name
 - ▶ Comments
- ▶ Internetistä löytyy paljon hajulistoja, esim:
 - ▶ <https://sourcemaking.com/refactoring/smells>
 - ▶ <http://c2.com/xp/CodeSmell.html>
 - ▶ <https://blog.codinghorror.com/code-smells/>

Koodin refaktorointi

Lääke koodihajuihin

Koodin refaktorointi

Lääke koodihajuihin

- ▶ Lääke koodihajuun on **refaktorointi** eli muutos koodin rakenteeseen, joka kuitenkin pitää koodin toiminnan ennallaan
- ▶ Eriksia koodin rakennetta parantavia refaktorointeja on lukuisia ¹³
- ▶ Muutama hyvin käytökeloinen ja nykyaikaisessa kehitysympäristössä (esim NetBeans, Eclipse, IntelliJ) automatisoitu refaktorointi:
 - ▶ **Rename method** (rename variable, rename class, CTRL+R)
 - ▶ Eli uudelleennimetään huonosti nimetty asia
 - ▶ **Extract method**
 - ▶ Jaetaan liian pitkä metodi erottamalla siitä omia apumetodejaan
 - ▶ **Extract interface**
 - ▶ Luodaan automaattisesti rajapinta perustuen jonkin luokan metodeihin ja korvataan suora riippuvuus luokkaan riippuvuudella luotuun rajapintaan

¹³Katso esimerkiksi:

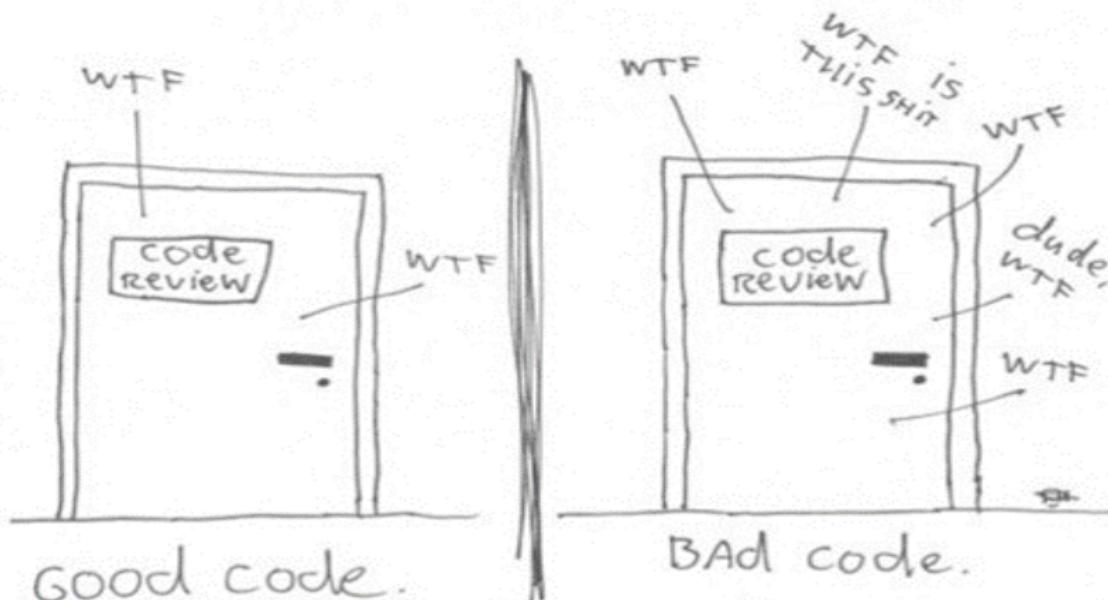
<http://sourcemaking.com/refactoring>

Koodin refaktorointi

Miten refaktorointi kannattaa tehdä

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

The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE



Test Driven Development

Test Drive it!

Test Driven Development

Test Drive it!

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



Test Driven Development

TDD eli Testivetoisen kehitys

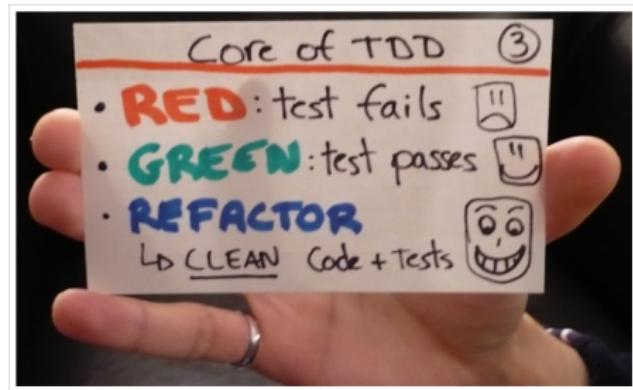
- ▶ TDD:ssä ohjelmoija (eikä siis erillinen testaaja) kirjoittaa testikoodin
- ▶ Testit laaditaan ennen koodattavan luokan toteutusta, yleensä jo ennen lopullista suunnittelua
- ▶ Sovelluskoodi kirjoitetaan täyttämään testien asettamat vaatimukset
 - ▶ Testit määrittelevät miten ohjelmoitavan luokan tulisi toimia
 - ▶ Testit toimivatkin osin koodin dokumentaationa, sillä testit myös näyttävät miten testattavaa koodia käytetään
- ▶ Testien on ennen toteutuksen valmistumista epäonnistuttava!
 - ▶ Näin pyritään varmistamaan, että testit todella testaavat haluttua asiaa

Oikeastaan TDD ei ole testausmenetelmä vaan ohjelmiston kehitysmenetelmä, joka tuottaa sivutuotteenaan automaattisesti ajettavat testit

Test Driven Development

TDD-sykli

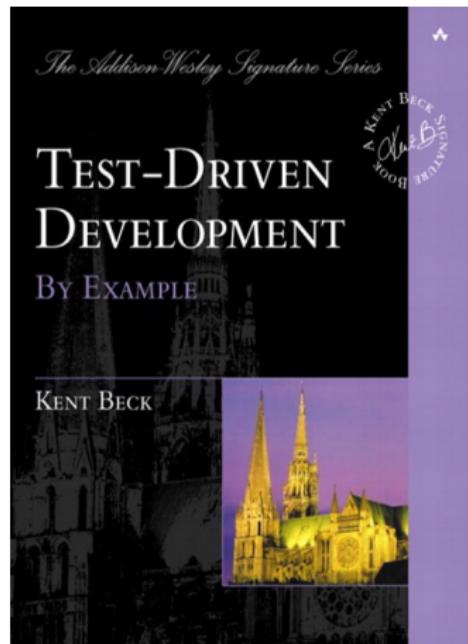
1. Tehdään yksi testitapaus
 - ▶ testitapaus testaa ainoastaan yhden "pienin" asian
2. Tehdään koodi joka läpäisee testitapauksen
3. Refaktoroidaan koodia, eli parannellaan koodin laattua ja struktuuria
 - ▶ Testit varmistavat koko ajan ettei mitään mene rikki
4. Kun koodin rakenne on kunnossa, palataan vaiheeseen (1) ↑



Test Driven Development

TDD-sykli

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



Järjestelmätason sekvenssikaaviot

Järjestelmä yhtenä oliona

Järjestelmätason sekvenssikaaviot

Järjestelmä yhtenä oliona

- ▶ Toimiakseen halutulla tavalla, on järjestelmän tarjottava ne toiminnot tai operaatiot, jotka käyttötapausten läpiviemiseen vaaditaan
- ▶ Joissain tilanteissa on hyödyllistä dokumentoida tarkasti, mitä yksittäisiä operaatioita käyttötapauksen toiminnallisuuden toteuttamiseksi järjestelmältä vaaditaan
- ▶ Dokumentointiin sopivat *järjestelmätason sekvenssikaaviot*, eli sekvenssikaaviot, joissa koko järjestelmä ajatellaan yhtenä oliona



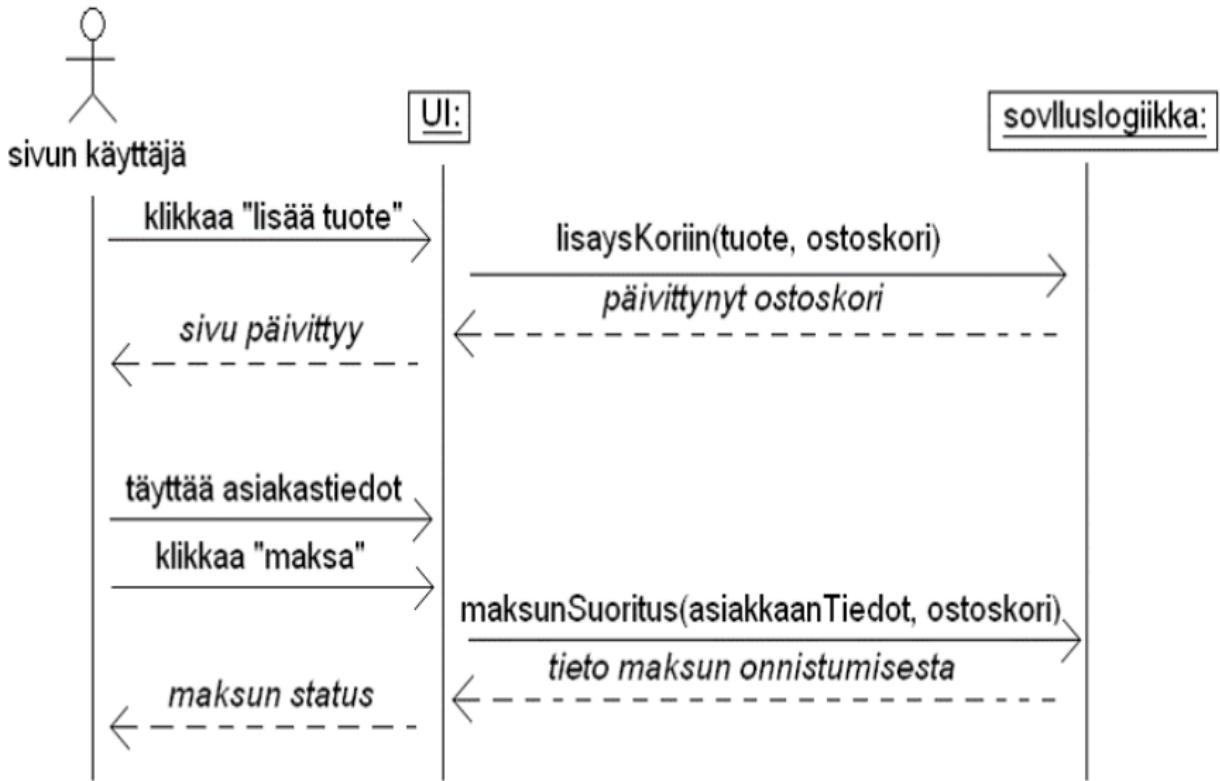
Järjestelmätason sekvenssikaaviot

Käyttöliittymän ja sovelluslogiikan eriyttäminen, osa 1

Järjestelmätason sekvenssikaaviot

Käyttöliittymän ja sovelluslogiikan eriyttäminen, osa 1

- ▶ Edellisen dian järjestelmätason sekvenssikaavio keskittyy käyttäjän ja järjestelmän väliseen interaktioon, eli siihen miten käyttäjä kommunikoi järjestelmän käyttöliittymän kanssa
- ▶ Käyttöliittymä ja varsinainen sovelluslogiikka kannattaa monestakin syystä eriyttää toisistaan, ja näin tehdään myös Biershopin toteutuksessa
- ▶ Tehdäänkin hieman tarkempi järjestelmätason sekvenssikaaviositys, jossa järjestelmä kuvataan kahtena "oliona", käyttöliittymänä ja sovelluslogiikkana:
 - ▶ Käyttöliittymä ottaa vastaan vastaan käyttäjän interaktion (esim. näppäinten painallukset ja syötteen antamisen) ja tulkitsee ne järjestelmän toiminnoiksi
 - ▶ Sovelluslogiikka vastaa varsinaisesta toiminnasta
- ▶ Nämä kaksi olioja eivät siis ole lopullisen sovelluksen oikeita olioita, ne ainoastaan kuvaavat järjestelmän rakenteen jakautumista erillisiin komponentteihin (joiden sisällä on "oikeita" ohjelointikielellä toteutettuja olioita)



Järjestelmätason sekvenssikaaviot

Käyttöliittymän ja sovelluslogiikan eriyttäminen, osa 1

- ▶ Käyttötapausten suorittaminen aiheuttaa käyttäjälle suoraan näkyviä toimenpiteitä, esim:
 - ▶ Käyttöliittymänäkymän päivittyminen
 - ▶ Siirtyminen toiseen näkymään
- ▶ Käyttötapausten suorittaminen aiheuttaa myös järjestelmän sisäisiä asioita
 - ▶ luodaan, päivitetään, poistetaan järjestelmän kohdealueen olioita ja niiden välisiä suhteita (kohdealueen olioita ovat siis määrittelyvaiheen luokkakaavion oliot, kuten esim tuote, ostos, ostoskori...)
- ▶ Käyttötapausten suorittaminen saattaa myös edellyttää muiden järjestelmien kanssa tapahtuvaa kommunikointia
 - ▶ Esim. luottokortin veloitus luottokunnan verkkorajapinnan avulla

Järjestelmätason sekvenssikaaviot

Käyttöliittymän ja sovelluslogiikan eriyttäminen, osa 1

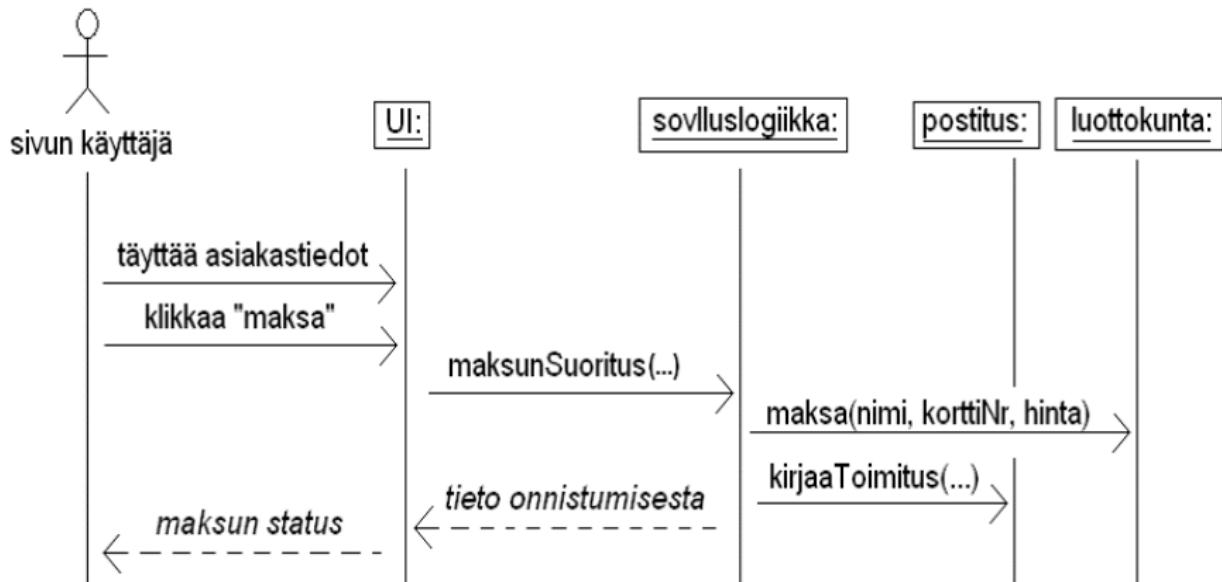
Ohessa tarkastellaan: käyttötapausten lisää ostoskoriin ja suorita maksu suorittamisen aiheuttamia järjestelmän sisäisiä asioita ja kommunikointia ulkoisten järjestelmien kanssa

► lisää ostos koriin

- ▶ Ostoskori-olioon lisättävä ostettavaa tuotetta vastaava uusi ostos-olio
- ▶ tai jos samaa tuotetta on jo korissa, päivitettyä korissa jo olevaa ostos-oliaa

► suorita maksu

- ▶ Suoritetaan maksu Luottokunnan rajapinnan avulla
- ▶ Jos maksu onnistuu
 - ▶ Ilmoitetaan ostoksen tiedot ja toimitusosoite postitusjärjestelmän verkkorajapinnalle
 - ▶ Tyhjennetään ostoskori
 - ▶ Luodaan ostotapahtuma



Järjestelmätason sekvenssikaaviot

Määrittelystä suunnittelun

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

Ohjelmiston arkkitehtuuri

Ohjelmiston suunnittelu

Ohjelmiston arkkitehtuuri

Ohjelmiston suunnittelu

Suunnitteluvaiheessa tarkoituksena on löytää sellaiset oliot, jotka pystyvät yhteistoiminnallaan toteuttamaan järjestelmältä vaaditavat operaatiot

- ▶ Suunnittelu jakautuu karkeasti ottaen kahteen vaiheeseen:
 1. Arkkitehtuurisuunnittelu
 2. Oliosuunnittelu
- ▶ Ensimmäinen vaihe on **arkkitehtuurisuunnittelu**, jonka aikana hahmotellaan järjestelmän rakenne karkeammalla tasolla
- ▶ Tämän jälkeen suoritetaan **oliostuunnittelu**, eli suunnitellaan oliot, jotka ottavat vastuulleen järjestelmältä vaaditun toiminnallisuuden toteuttamisen
 - ▶ Yksittäiset oliot eivät yleensä pysty toteuttamaan kovin paljoa järjestelmän toiminnallisuudesta
 - ▶ Erityisesti oliosuunnitteluvaiheessa tärkeäksi seikaksi nouseekin olioiden välinen yhteistyö, eli se vuorovaikutus, jolla oliot saavat aikaan halutun toiminnallisuuden

Ohjelmiston arkkitehtuuri

Määrittely- ja suunnittelutason luokkien yhteys

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

Ohjelmiston arkkitehtuuri

Määrittely- ja suunnittelutason luokkien yhteys

- ▶ Ohjelmiston arkkitehtuurilla (engl. *software architecture*) tarkoitetaan ohjelmiston korkean tason rakennetta
 - ▶ jakautumista erillisiin komponentteihin
 - ▶ komponenttien välisiä suhteita
- ▶ *Komponentilla* tarkoitetaan yleensä kokoelmaa toisiinsa liittyviä olioita/luokkia, jotka suorittavat ohjelmassa joitain tehtäväkokonaisuutta
 - ▶ esim. käyttöliittymän voitaisiin ajatella olevan yksi komponentti
- ▶ Iso komponentti voi muodostua useista alikomponenteista
 - ▶ Biershopin sovelluslogiikkakomponetti voisi sisältää komponentin, joka huolehtii sovelluksen alustamisesta ja yhden komponentin kutakin järjestelmän toimintokokonaisuutta varten
 - ▶ tai Biershopissa voisi olla omat komponentit ostosten tekoa ja varastotietojen ylläpitoa varten

Ohjelmiston arkkitehtuuri

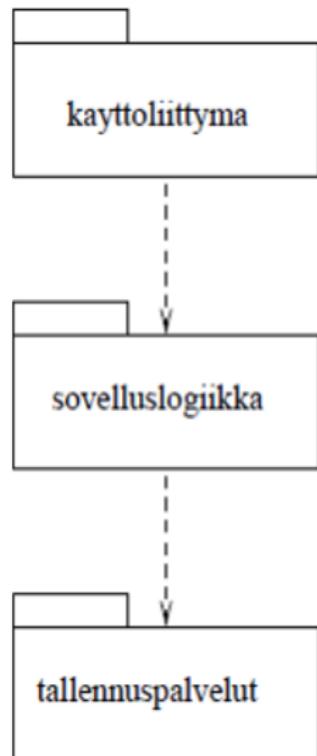
Määrittely- ja suunnittelutason luokkien yhteyks

- ▶ Jos ajatellaan pelkkää ohjelman jakautumista komponentteiksi, puhutaan oikeastaan *loogisesta arkkitehtuurista*
- ▶ Looginen arkkitehtuuri ei ota kantaa siihen miten eri komponentit sijoitellaan, eli toimiiko esim. käyttöliittymä samassa koneessa kuin sovelluksen käyttämä tietokanta
- ▶ Ohjelmistoarkkitehtuurit on TODELLA laaja aihe jota käsitellään nyt vain pintapuolisesti
 - ▶ Aiheesta on olemassa noin 4. vuotena suoritettava syventävien opintojen kurssi *Ohjelmistoarkkitehtuurit*
 - ▶ Asiaa käsitellään myös 2. vuoden kurssilla *Ohjelmistotuotanto*
- ▶ UML:ssa on muutama kaaviotyyppi jotka sopivat arkkitehtuurin kuvaamiseen
 - ▶ Komponenttikaaviota ja sijoittelukaaviota emme nyt käsittele
 - ▶ Komponenttikaavio on erittäin käytökeloinen kaaviotyyppi, mutta silti jätämme sen myöhemmille kursseille

Ohjelmiston arkkitehtuuri

Pakkauskaavio

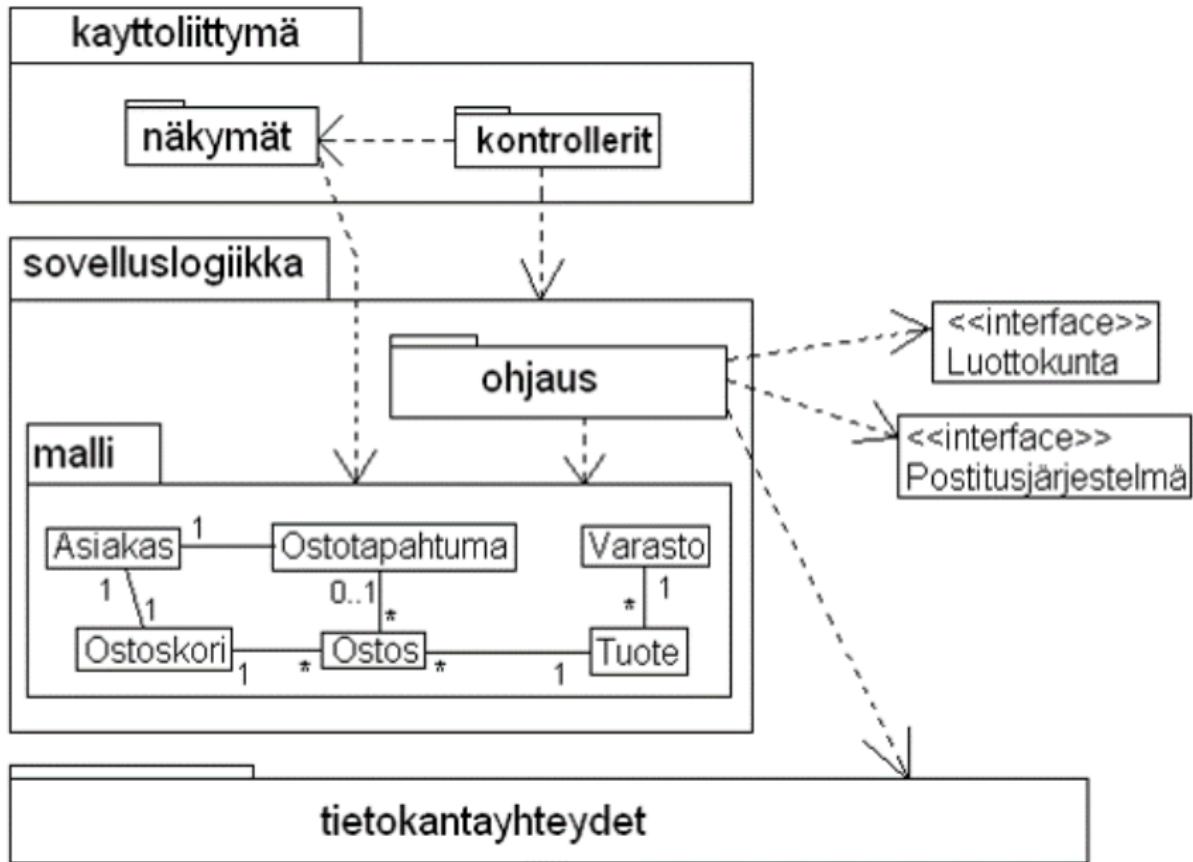
- ▶ Ohessa karkea hahmotelma Biershopin arkkitehtuurista
- ▶ Näemme, että järjestelmä on jakautunut kolmeen pää-komponenttiin
 - ▶ Käyttöliittymä, Sovelluslogiikka ja Tallennuspalvelut
- ▶ Jokainen komponentti on kuvattu omana pakkauksena, eli isona laatikkona, jonka vasempaan ylälaitaan liittyy pieni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



Ohjelmiston arkkitehtuuri

Pakkauskaavio

- ▶ Pakkauskaaviossa yksi komponentti kuvataan pakkaussymbolilla
 - ▶ Pakkauksen nimi on joko keskellä symbolia tai yläneurkan laatikossa
- ▶ Pakkausten välillä olevat riippuvuudet ilmaistaan katkoviivanuolena, joka suuntautuu pakkaukseen, johon riippuvuus kohdistuu
- ▶ Riippuvuus tarkoittaa käytännössä sitä, että *käyttöliittymän olioit kutsuvat sovelluslogiikan olioiden metodeja*
- ▶ Pakkauksen sisältö on mahdollista piirtää pakkaussymbolin sisään
 - ▶ Pakkauksen sisällä voi olla alipakkauksia tai luokkia
- ▶ Riippuvuudet voivat olla myös alipakkausten välsiä



Ohjelmiston arkkitehtuuri

Kerrosarkkitehtuuri

Ohjelmiston arkkitehtuuri

Kerrosarkkitehtuuri

- ▶ Kerrosarkkitehtuuri (engl. *layered architecture*) yksi hyvin tunnettu arkkitehtuurimalli (engl. *architecture pattern*), eli periaate, jonka mukaan tietynlaisia ohjelmia kannattaa pyrkiä rakentamaan
- ▶ Kerros on kokoelma toisiinsa liittyviä olioita tai ali komponentteja, jotka muodostavat esim. toiminnallisuuden suhteen loogisen kokonaisuuden ohjelmistosta

Kerrosarkkitehtuurissa on pyrkimyksenä järjestellä komponentit sitten, että ylempänä oleva kerros käyttää ainoastaan alempana olevien kerroksien tarjoamia palveluita

Ohjelmiston arkkitehtuuri

Kerrosarkkitehtuurin etuja

- ▶ Kerroksittaisuus helpottaa ylläpitoa
 - ▶ Kerroksen sisältöä voi muuttaa vapaasti jos sen palvelurajapinta eli muille kerroksille näkyvät osat säilyvät muuttumattomina
 - ▶ Sama pätee tietysti mihin tahansa komponenttiin
- ▶ Jos kerroksen palvelurajapintaan tehdään muutoksia, aiheuttavat muutokset ylläpitotoimenpiteitä ainoastaan ylemmän kerroksen riippuvuuksia omaavin osiin
- ▶ Sovelluslogiikan riippumattomuus käyttöliittymästä helpottaa ohjelman siirtämistä uusille alustoille
- ▶ Alimpien kerroksien palveluja, kuten esim. tallennuspalvelukerrosta voidaan ehkä uusiokäyttää myös muissa sovelluksissa
- ▶ Ylemmät kerrokset voivat toimia korkeammalla abstraktiotasolla
 - ▶ Kaikkien ohjelmoijien ei tarvitse ymmärtää kaikkia detaljeja, osa voi keskittyä tietokantaan, osa käyttöliittymiin, osa sovelluslogiikkaan

Ohjelmiston arkkitehtuuri

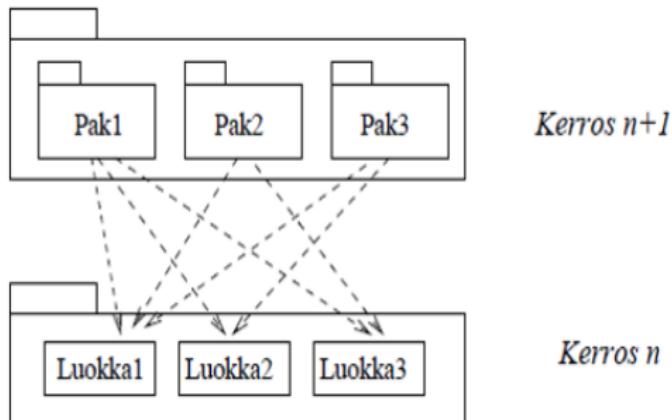
Ei pelkkiä kerroksia...

- ▶ Myös kerrostien sisällä ohjelman logisesti toisiinsa liittyvät komponentit kannattaa ryhmitellä omiksi kokonaisuuksiksi, joka voidaan UML:ssa kuvata pakkauksena
- ▶ Yksittäisistä komponenteista kannattaa tehdä mahdollisimman *yhtenäisiä* toiminnallisuudeltaan
 - ▶ eli sellaisia, joiden osat kytkeytyvät tiiviisti toisiinsa ja palvelevat ainoastaan yhtä selkeästi eroteltua tehtäväkokonaisutta
- ▶ Samalla pyrkimyksenä on, että erilliset komponentit ovat mahdollisimman löyhästi kytettyjä (engl. *loosely coupled*) toisiinsa
 - ▶ komponenttien välisiä riippuvuuksia pyritään minimoimaan
- ▶ **Selkeä jakautuminen komponentteihin myös helpottaa:** työn jakamista suunnittelu- ja ohjelmointivaiheessa sekä testausta, laajennusta että ylläpitoa!

Ohjelmiston arkkitehtuuri

Kerroksellisuus ei riitä

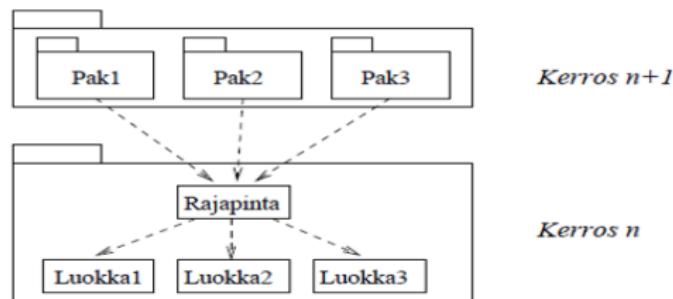
- ▶ Pelkkä kerroksittaisuus ei tee ohjelman arkkitehtuurista automaattisesti hyvää.
- ▶ Alla tilanne, missä kerroksen $n+1$ kolmella alipaketilla on kullakin paljon riippuvuuksia kerroksen n sisäisiin komponenttiin
- ▶ Esim. muutos kerroksen n luokkaan 1 aiheuttaa nyt muutoksen hyvin moneen ylemmän kerroksen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



Ohjelmiston arkkitehtuuri

Kerroksellisuus ei riitä

- ▶ **Kerrosten** välille kannattaa määritellä selkeä **rajapinta**
- ▶ Yksi tapa toteuttaa rajapinta on luoda kerroksen sisälle erillinen rajapintaolio, jonka kautta ulkoiset yhteydet tapahtuvat
 - ▶ Tätä periaatetta sanotaan fasaadiksi (engl. facade pattern)
 - ▶ Huom: rajapinnalla ei nyt tarkoiteta pelkästään Javan rajapintaa, kyseessä voi olla myös usean rajapinnan kokonaisuus jonka ylempi kerros näkee
- ▶ Alla luotu rajapintaolio kerrokselle n. Kommunikointi kerroksen kanssa tapahtuu rajapintaolian kautta:



Ohjelmiston arkkitehtuuri

Käyttöliittymän ja sovelluslogiikan erottaminen, osa 2

Ohjelmiston arkkitehtuuri

Käyttöliittymän ja sovelluslogiikan erottaminen, osa 2

- ▶ Kerrosarkkitehtuurin ylimpänä kerroksena on yleensä käyttöliittymä
- ▶ Pidetään järkeväänä, että **ohjelman sovelluslogiikka on täysin erotettu käyttöliittymästä**
 - ▶ Sovelluslogiikan erottaminen lisää koodin määrää, joten jos kyseessä "kertakäyttösovellus", ei ylimääräinen vaiva kannata
- ▶ Käytännössä tämä tarkoittaa kahta asiaa:
 - ▶ **Sovelluksen palveluja toteuttavilla olioilla ei ole suoraan yhteyttä käyttöliittymän olioihin** (esim Swing componentit tai HTML-koodia generoivat luokat)
 - ▶ **Käyttöliittymän toteuttavat oliot eivät sisällä ollenkaan ohjelman sovelluslogiikkaa**
 - ▶ Käyttöliittymäolot ainoastaan piirtävät käyttöliittymäkomponentit ruudulle, välittävät käyttäjän komennot eteenpäin sovelluslogiikalle ja heijastavat sovellusolioiden tilaa käyttäjille

Ohjelmiston arkkitehtuuri

Käyttöliittymän ja sovelluslogiikan erottaminen, osa 2

- ▶ Käytännössä erottelu tehdään liittämällä käyttöliittymän ja sovellusalueen olioiden väliin erillisiä komponentteja, jotka koordinoivat käyttäjän komentojen aiheuttamien toimenpiteiden suoritusta sovelluslogiikassa
- ▶ Erottelun pohjana on Ivar Jacobsonin kehittämä idea oliotyyppejä jaoitteluista kolmeen osaan, rajapintaolioihin (boundary), ohjausolioihin (control) ja sisältöolioihin (entity)
- ▶ Käyttöliittymän (eli rajapintaoloiden) ja sovelluslogiikan (eli sisältöoloiden) yhdistävät **ohjausoliot**¹⁴
 - ▶ Joissain yhteyksissä, esim. Java Spring -sovelluskehyskseen yhteydessä samasta asiasta käytetään nimitystä palvelu (engl. *service*)
- ▶ Käyttöliittymä ei siis itse tee mitään sovelluslogiikan kannalta oleellisia toimintoja, vaan ainoastaan välittää käyttäjien komentoja ohjausolioille, jotka huolehtivat sovelluslogiikan olioiden manipuloimisesta

¹⁴idea ohjausolioista on hiukan sukua ns. MVC-periaatteelle, tarkalleen ottaen kyse ei kuitenkaan ole täysin samasta asiasta

Ohjelmiston arkkitehtuuri

Käyttöliittymän ja sovelluslogiikan erottaminen, osa 2

- ▶ Kuten äsknen mainittu: yksi tapa tehdä käyttöliittymän ja sovelluslogiikan erottelu on lisätä käyttöliittymän ja varsinaisten sovellusolioiden väliin **ohjausolioita**, jotka ottavat vastaan käyttöliittymästä tulevat operaatiokutsut ja kutsuvat edelleen sovelluslogiikan olioiden metodeja
- ▶ Ohjausolio voi olla ohjelman **kaikkien operaatioiden yhteinen** ohjausolio tai vaihtoehtoisesti voidaan käyttää **käyttötapauskohtaisia** ohjausolioita
 - ▶ Välimuodotkin ovat mahdollisia eli joillain käyttötapauksella voi olla oma ohjausolio ja jotkut taas käyttävät yhteistä ohjausolia
- ▶ Liian monesta asiasta huolehtivat ohjausoliot kuitenkin rikkovat single responsibility -periaatetta ja vaikeuttavat ohjelman ylläpidettävyyttä ja testattavuutta!

Ohjelmiston arkkitehtuuri

Käyttöliittymän ja sovelluslogiikan erottaminen, osa 2

- ▶ Todettakoon edellisiin kalvoihin liittyen, että sopivien teknisten apuluokkien keksiminen on äärimmäisen haastavaa
- ▶ Monissa suunnittelumalleista (engl. *design patterns*) on kyse juuri sopivien teknisten ratkaisujen ja abstraktioiden mukaantuomisesta
 - ▶ Esim. fasaadi on suunnitelumalli!
- ▶ Valitettavasti suunnittelumalleja ja muuta haasteellista ja mielenkiintoisia oliosuunnittelun liittyvää ei tällä kurssilla juuri ehditä käsittelemään
 - ▶ Asiaan tutustutaan tarkemmin toisen vuoden kurssilla Ohjelmistotuotanto
 - ▶ Aiheesta löytyy runsaasti kirjallisuutta
 - ▶ Robert Martin: Agile and iterative development
 - ▶ Larman: Applying UML and Patterns
 - ▶ Freeman et. All.: Head first design patterns
 - ▶ Gamma et all.: Design patterns

Ohjelmistotekniikan menetelmät

Luento 6

Järjestelmätestaus ja kertausta

Kertausta ja uutta asiaa testauksesta

Ohjelmistotuotantoprosessin vaiheita

- ▶ Ohjelmistotuotanto on yhteisnimitys niille työnteron ja työnjohdon menetelmille, joita käytetään, kun tuotetaan tietokoneohjelmia sekä monista tietokoneohjelmista koostuvia tietokoneohjelmistoja.
- ▶ Ohjelmistotuotantoprosessiin liittyy erilaisia vaiheita:
 1. Vaatimusmäärittely
 2. Suunnittelu
 3. Toteutus
 4. Testaus
 5. Ylläpito
- ▶ Perinteisesti vaiheet tehdään peräkkäin (*vesiputoosmalli*)
- ▶ Vaihtoehtoinen tapa on työskennellä *iteratiivisesti*, eli toistaa vaiheita useaan kertaan peräkkäin ja limittäin (*Ketterä ohjelmistotuotanto*)

Kertausta ja uutta asiaa testauksesta

Kertausta testauksesta

► Yksikkötestaus

- Toimivatko yksittäiset metodit ja luokat kuten halutaan?

► Integraatiotestaus

- Toimivatko yksittäiset *moduulit* yhdessä halutulla tavalla?

► Järjestelmä/hyväksymistestaus

- Toimiiko kokonaisuus niin kuin vaatimusdokumentissa sanotaan?

► Regressiotestaus

- *regressio* \approx palautuminen, taantuminen, takautuminen
- järjestelmälle muutosten ja bugikorjausten jälkeen ajettavia testejä jotka varmistavat että muutokset eivät riko mitään
- Regressiotestit koostuvat yleensä yksikkö-, integraatio- ja järjestelmä/hyväksymätesteistä

Lähes kaikki kurssilla tähän mennessä kirjoitetut testit ovat olleet yksikköttestejä, jotka on automatisoitu JUnitin avulla

Kertausta ja uutta asiaa testauksesta

Testien laatu

- ▶ Normaalien syötteiden lisäksi on testeissä erityisen tärkeää tutkia testattavien metodien toimintaa virheellisillä syötteillä ja erilaisilla raja-arvoilla
- ▶ Testien laataa on kurssin aikana mitattu kolmella tavalla:
 - ▶ **Rivikattavuus** mittaa miten montaa prosenttia koodiriveistä testit suorittavat
 - ▶ **Haarautumakattavuus** taas mittaa miten monta prosenttia koodin haarautumakohdista (if:in ja toistolauseiden ehdot) on suoritettu
 - ▶ **Mutaatiotestauksella** tarkasteltiin huomaavatko testit koodiin asetettuja bugeja
 - ▶ Mutaatiotestauskirjasto tekee koodiin pieniä muutoksia, eli mutantteja, esim. muuttaa ehdossa <:n <=:ksi
 - ▶ Jos testit eivät mene mutanttien takia rikki, on epäilys että testit eivät huomaisi koodissa olevaa virhettä ja testejä tulee parantaa
- ▶ Kun käytimme Maven-käänösystävällistöä, oli testien laatumittareiden käyttö helppoa...

Kertausta ja uutta asiaa testauksesta

Testaus ja riippuvuuksien eliminointi

- ▶ Testaus muuttuu joskus tarpeettoman hankalaksi, jos ohjelmissa on luokkien välillä tarpeettomia riippuvuuksia
 - ▶ Eräs viime viikon laskareissa kokeillun Test Driven Development (TDD) -menetelmän hyviä puolia on se, että koodista tulee "automaatisesti" hyvin testattavissa olevaa ja turhia riippuvuuksia ei koodissa yleensä esiinny
- ▶ Seuraavalla dialla Ohjelmoinnin perusteiden viikon 3 tehtävän *Lottoarvonta ratkaisu*
 - ▶ Lottorivi-olion konstruktori luo Random-olion, jonka avulla lottonumerot arvotaan
 - ▶ Luokan testaaminen on nyt erittäin vaikeaa, testeissähän ei pystytä kontrolloimaan Random-olian arpomia lukuja
 - ▶ Lottorivistä tulisi testata ainakin seuraavat asiat
 - ▶ lottorivi ei voi sisältää samaa arvoa useampaan kertaan
 - ▶ lottorivi voi saada arvoja väliltä 1-39

Kertausta ja uutta asiaa testauksesta

Lottorivi ja riippuvuuden injekointi

```
public class Lottorivi {  
    private ArrayList<integer> numerot;  
    private Random random;  
  
    public Lottorivi() {  
        random = new Random();  
        arvoNumerot();  
    }  
  
    public ArrayList<integer> numerot() {  
        return numerot;  
    }  
  
    public void arvoNumerot() {  
        numerot = new ArrayList<>();  
        while (numerot.size() < 7) {  
            int numero = random.nextInt(39) + 1;  
            if (!numerot.contains(numero)) {  
                numerot.add(numero);  
            }  
        }  
    }  
}
```

Kertausta ja uutta asiaa testauksesta

Lottorivi ja riippuvuuden injektointi

- ▶ Ongelmana Lottorivin testaamisessa ei sinänsä ole riippuvuus Random-olioon, vaan se, että lottorivin ulkopuolelta ei ole tapaa päästää käsiksi Lottorivin luomaan Random-olioon
- ▶ Muutetaan lottorivin konstruktoria seuraavasti:
 - ▶ public Lottorivi(Random random)
- ▶ Eli lottorivi muodostetaan nyt luomalla Random-olio, joka annetaan lottoriville konstruktorin parametrina:
- ▶ Tekniikasta käytetään nimitystä **riippuvuuksien injektointi** (engl. *dependency injection*)
 - ▶ riippuvuutena olevaa oliaa ei luoda konstruktorissa tai luokan sisällä (ellei haluta: `this(new Random());`)
 - ▶ konstruktorille annetaan valmiiksi luotu riippuvuus, konstruktori laittaa riippuvuutena olevan olion talteen oliomuuttujaan

Kertausta ja uutta asiaa testauksesta

Lottorivi ja riippuvuuden injekointi

- ▶ Koska lottorivi saa käyttämänsä Random-olian konstruktorin parametrina, voimme helposti luoda omia versioitamme Randomista ja käyttää niitä apuna testaamisessa
- ▶ Luodaan luokka RandomStub, joka "arpoo" sille konstruktorin parametrina annetut luvut:

```
public class RandomStub extends Random {  
    private List<integer> numbers;  
    public RandomStub(Integer ... luvut) {  
        numbers = new ArrayList<>();  
        numbers.addAll(Arrays.asList(luvut));  
    }  
  
    @Override // korvataan peritty toiminnallisuus  
    public int nextInt(int bound) {  
        return numbers.remove(0);  
    }  
}
```

- ▶ Koska RandomStub *perii* luokan Random, voidaan sitä käyttää kaikkialla randomin paikalla

Integraatiotestaus

Integraatiotestaus ja "hankalat" riippuvuudet

Integraatiotestaus

Integraatiotestaus ja "hankalat" riippuvuudet

- ▶ Yksikkötestaus on käsitteenä suhteellisen hyvin ymmärretty
- ▶ Järjestelmä/hyväksymätestauksessa taas testataan ohjelmiston tarjoamaa toiminnallisuutta käyttöliittymän läpi samaan tapaan kuin loppukäyttäjä tulee järjestelmää käyttämään
- ▶ Kaikki näiden väliin jäävät testauksen muodot ovat *integraatiotestausta*
- ▶ 5. laskareissa olleet testit ovat oikeastaan integraatiotestejä!
- ▶ Vaikka testit kohdistuvatkin luokan Palkanlaskenta metodeihin, ne kuitenkin oleellisesti testaavat palkanlaskennasta, henkilöstä ja muutamasta muusta luokasta muodostuvan kokonaisuuden toimintaa
- ▶ Myös 5. luennolla käsitellyn Biershopin ohjausoloiden testit ovat integraatiotestejä, sillä ne testaavat useamman luokan yhteistoimintaa

Integraatiotestaus

Integraatiotestaus ja "hankalat" riippuvuudet

- ▶ Testien kannalta on hieman ikävää, jos testattavan järjestelmän jokin komponentti keskustelee tietokannan tai jonkin verkossa olevan komponentin kanssa
 - ▶ Testien suoritus hidastuu
 - ▶ Testien tulos saattaa olla riippuvainen tietokannan sisällöstä
 - ▶ Verkon yhteysongelmat voivat vaikuttaa testien toimivuuteen
- ▶ Näissä tilanteissa on järkevä toteuttaa hankalista luokista testausta varten valekomponentti eli *stub* ja injektoida se testattavalle järjestelmälle hankalan riippuvuuden sijaan
 - ▶ Stubin tulee toimia testattavan järjestelmän kannalta kuten oikea komponentti
- ▶ Tämä tietysti edellyttää, että riippuvuudet pystytään injektoimaan testattaville luokille kalvojen 7-9 tapaan
- ▶ Stub-olio on helppo luoda perimällä alkuperäinen riippuvuus ja korvata kokonaan siinä testien kannalta merkityksellisten metodien toiminnallisuus

Integraatiotestaus

Biershop integraatiotesti esimerkki

- ▶ Biershopin testeissä on testauksen kannalta ikävä, tietokantaa käyttävä TuoteDAO eli tietokantayhteydestä huolehtiva olio korvattu testien suorituksen aikana keskusmuistissa toimivalla tietokantayteysoliolla:

```
@Test
public void yhdenTuotteenLisaaminenKoriin() {
    Varasto varasto = new Varasto(new TuoteDAOForTest());
    Tuote tuote = varasto.etsiTuote(1);
    int saldoAlussa = tuote.getSaldo();
    Ostoskori kori = new Ostoskori();
    Lisaakoriin komento = new Lisaakoriin(tuote.getId(), kori);
    komento.suorita();

    assertEquals(saldoAlussa-1, tuote.getSaldo());
    assertEquals( 1, kori.tuotteitaKorissa() );
    assertEquals( tuote.hinta(), kori.hinta() );
}
```

- ▶ Testi varmistaa, että ohjausolio Lisaakoriin toimii halutulla tavalla, eli vähentää tuotteen varastosalhoa, vie tuotteen koriin (eli koriin ilmestyy yksi oikean hintainen tuote)

Järjestelmätestaus

Miten käyttöliittymää voi testata?

Järjestelmätestaus

Miten käyttöliittymää voi testata?

Järjestelmätestauksessa tulee varmistaa toimiiko ohjelmisto sille vaatimusmäärittelyssä asetettujen vaatimusten mukaan

- ▶ Testauksen tulee tapahtua saman rajapinnan läpi, miten loppukäyttäjä järjestelmää käyttää, eli sovelluksesta riippuen joko komentoriviltä, graafisesta käyttöliittymästä tai webselaimesta
- ▶ Järjestelmätestit kannattaa muodostaa ohjelmiston käyttötapausten mukaan
- ▶ Tarkastellaan Ohjelmoinnin jatkokurssin ensimmäisen viikon tehtävää Laskin¹⁵
- ▶ Käyttäjän syötteiden simuloiminen on mahdollista korvaamalla ohjelman käyttämä System.in syötevirta omalla oliolla
- ▶ Vastaavasti voimme korvata System.out tulostusvirran

¹⁵ Tehtävä numero 128

Järjestelmätestaus

Laskimen testi, joka korvaa syöte- ja tulostevirran

- ▶ Järjestelmätestauksen voi hoitaa myös JUnit-kirjaston avulla.
 - ▶ JUnit ei ole tarkoitukseen optimaalinen, mutta "riittävän hyvä", ja Javalle ei kauheasti parempiakaan vaihtoehtoja ole tarjolla

```
@Test
public void yhdenTuotteenLisaaminenKoriin() {
    Varasto varasto = new Varasto(new TuoteDAOForTest());
    Tuote tuote = varasto.etsiTuote(1);
    int saldoAlussa = tuote.getSaldo();
    Ostoskori kori = new Ostoskori();
    LisaaKoriin komento = new LisaaKoriin(tuote.getId(), kori);
    komento.suorita();

    assertEquals(saldoAlussa-1, tuote.getSaldo());
    assertEquals( 1, kori.tuotteitaKorissa() );
    assertEquals( tuote.hinta(), kori.hinta() );
}
```

Järjestelmätestaus

Laskimen testi, joka korvaa syöte- ja tulostevirran

- ▶ Testin simuloitu syöte on merkkijono, jossa yksittäiset syöterivit päättyyvät rivinvaihtoon
- ▶ Merkkijonosta luodaan bittivirta, joka asetetaan *ohjelman syötevirran* arvoksi
- ▶ `setUp`-metodissa testi asettaa uuden `ByteArrayOutputStream`-oliota ohjelman `tulosvirraksi` metodin `System.set` avulla
- ▶ Laskimen suorituksen jälkeen `tulosvirtaolio` muutetaan `toString`-metodilla merkkijonoksi ja tarkastetaan, että ohjelman tulostus on odotetun kaltainen
- ▶ Testit olettavat, että ohjelmassa luodaan ainoastaan yksi `Scanner`-olio
- ▶ Vaihtoehtoinen ratkaisu syötevirran korvaamiselle olisi ollut muuttaa laskinta siten, että lukija-olio oltaisiin injektoitu konstruktorin parametrina

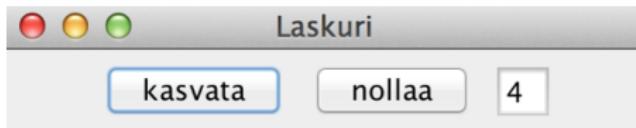
Graafisen käyttöliittymän testaus

Java-Swing käyttöliittymän testausta

Graafisen käyttöliittymän testaus

Java-Swing käyttöliittymän testausta

- ▶ Tarkastellaan yksinkertaista graafista ohjelmaa, jonka avulla laskurin arvoa voidaan kasvattaa tai laskuri voidaan nollata



- ▶ Javan Swing-kirjastolla tehtyjen graafisten sovellusten testaaminen ei loppujenlopuksi ole kovin vaikeaa
- ▶ Asetetaan komponenteille nimet:

```
private void luoKomponentit(Container container) {  
    add = new JButton("kasvata");  
    add.setName("kasvata");  
    reset = new JButton("nollaa");  
    reset.setName("nollaa");  
    reset.setEnabled(false);  
    show = new JTextField(" " + value);  
    show.setName("tulos");  
}
```

Graafisen käyttöliittymän testaus

Java-Swing käyttöliittymän testausta

- ▶ AssertJ¹⁶ on johtava työkalu Swing-sovellusten testaamiseen:

```
public class LaskuriTest extends AssertJSwingJUnitTestCase {  
    private FrameFixture window;  
  
    @Test  
    public void laskuriAlussaNolla() {  
        window.textBox("tulos").requireText("0");  
    }  
  
    @Test  
    public void laskuriKasvaa() {  
        window.button("kasvata").click();  
        window.textBox("tulos").requireText("1");  
        window.button("kasvata").click();  
        window.button("kasvata").click();  
        window.textBox("tulos").requireText("3");  
    }  
}
```

- ▶ window-olio tarjoaa pääsyn eri komponentteihin. Normaalisti käytettävien assert-lauseiden sijaan testien odotettu tulos määritellään require-määreiden avulla

¹⁶<http://joel-costigliola.github.io/assertj/>

Kurssikoe

Tietoa kurssikokeesta

Kurssikoe

Tietoa kurssikokeesta

- ▶ Kokeen voi suorittaa seuraavina päivinä:
 - ▶ Ke 24.8.2016 klo 17-20
 - ▶ Ma 19.9.2016 klo 17-20
 - ▶ Ke 19.10.2016 klo 17-20
- ▶ Nippelitason detaljien sijasta kokeessa arvostetaan enemmän sovellusosaamista
 - ▶ Tosin erääät detaljit (mainitaan myöhemmin) ovat tärkeitä
- ▶ Kokeeseen saa tuoda mukanaan 2-puoleisen, **itse tehdyn, käsin kirjoitetun** yhden A4-arkin lunttilapun.
- ▶ Lunttilapun teossa siis ei saa käyttää kopiokonetta, tietokonetta tai printteriä
 - ▶ Yksikään kokeen kysymys ei tosin tule olemaan sellainen että kukaan onnistuisi kirjoittamaan vastauksen etukäteen lunttilapulle

Kurssikoe

Mikä on tärkeintä kurssilla/kokeessa?

- ▶ Kokonaiskuva: *Mitä? Miksi? Milloin? Missä? Miten?*
- ▶ **Ohjelmistotuotantoprosessin** vaiheet on syytä osata
- ▶ **Testaamisen rooli** ohjelmistotuotantoprosessissa ymmärrettävä
 - ▶ testejäkin saattaa joutua kirjoittamaan
 - ▶ kannattaa kerrata laskareiden testaukseen liittyvät asiat
- ▶ UML:sta ylivoimaisesti tärkeimpiä ovat
 - ▶ **Luokkakaaviot**
 - ▶ **Sekvenssikaaviot**
 - ▶ **Käyttötapauskaaviot**
- ▶ **Käyttötapausmallinnus**
- ▶ **Käsiteanalyysi**, eli määrittelyvaiheen luokkamallin muodostaminen tekstistä
- ▶ Määrittelystä suunnitteluun eli **oliosuunnittelu**:
- ▶ **Takaisinmallinnus**, eli valmiista koodista luokka- ja sekvenssikaavioiden teko

Kurssikoe

Oliosuunnittelusta kokeessa

- ▶ Oliosuunnittelun tehtävä on löytää/keksiä oliot ja niiden operaatiot siten, että ohjelma toimii kuten vaatimuksissa halutaan
 - ▶ Erittäin haastavaa: enemmän "art" kuin "science"
- ▶ Kurssilla on tarkasteltu muutamia yleisiä oliosuunnittelun periaatteita
 - ▶ **Single responsibility** eli luokilla vain yksi vastuu
 - ▶ **Program to an interface, not to concrete implementation**, eli suosi rajapintoja
 - ▶ **Favor composition over inheritance**, eli älä väärinkäytä perintää
 - ▶ Tarpeettomien **riippuvuuksien minimointi**
- ▶ Merkki huonosta suunnittelusta: koodihaju (engl. *code smell*)
- ▶ Lääke huoonon suunnittelun/koodihajuun: refaktoriointi
 - ▶ Muutetaan koodin rakennetta parempaan suuntaan muuttamatta toiminnallisuutta

Kurssikoe

Oliosuunnittelusta kokeessa

- ▶ Luennolla 5 Biershop-esimerkin yhteydessä katsottiin hieman vastuupohjaista oliosuunnitteluteknikkaa (engl. *responsibility driven*), jossa mietitään mitä vastuuta oliolla on ja mitä tietoane jakavat keskenään
- ▶ Laskareissa kokeiltiin **Test Driven Development**-menetelmää, joka "yhdistää" testauksen, ohjelmoinnin ja suunnittelun
- ▶ **Mitä aiheesta pitää osata kokeessa?**
 - ▶ Yleiset periaatteet
 - ▶ Periaatteiden soveltaminen yksinkertaisessa tilanteessa
 - ▶ Huonon koodin tunnistaminen ja refaktoriointi paremmaksi
 - ▶ Ymmärrys, miksi periaatteet ovat olemassa ja milloin niitä saa rikkoa ja mitä käy jos niitä rikkoo (tekninen velka)

Kurssikoe

Käyttötapaukset ja käyttötapauskaaviot

- ▶ Käyttötapausten ja käyttäjien suhdetta kuvaaa UML:n käyttötapauskaavio
- ▶ Huomattavaa on, että koska käyttötapausmallia käytetään vaatimusmäärittelyssä, ei ole syytä puuttua järjestelmän sisäisiin yksityiskohtiin
 - ▶ Tarkastellaan ainoastaan, miten sovelluksen toiminta näkyy ulospäin
- ▶ Itse käyttötapauksen sisältö kuvataan *tekstuaalisesti*, käyttäen esim. Alistair Cockburnin käyttötapauspohjaa
- ▶ Käyttötapauskaavio toimii hyvänä yleiskuvana, mutta vasta tekstuallinen kuvaus määrittelee toiminnan tarkemmin
- ▶ Seuraavana vaiheena saattaisi olla tarkemman, käyttöliittymäspesifisen käyttötapauksen kirjoittaminen

Käyttötapaus 1: otto

Tavoite: asiakas nostaa tililtään haluamansa määrän rahaa

Käyttäjät: asiakas, pankki

Esieheto: kortti syötetty ja asiakas tunnistautunut

Jälkieheto: käyttäjä saa tililtään haluamansa määrän rahaa

Jos saldo ei riitä, tiliä ei veloiteta

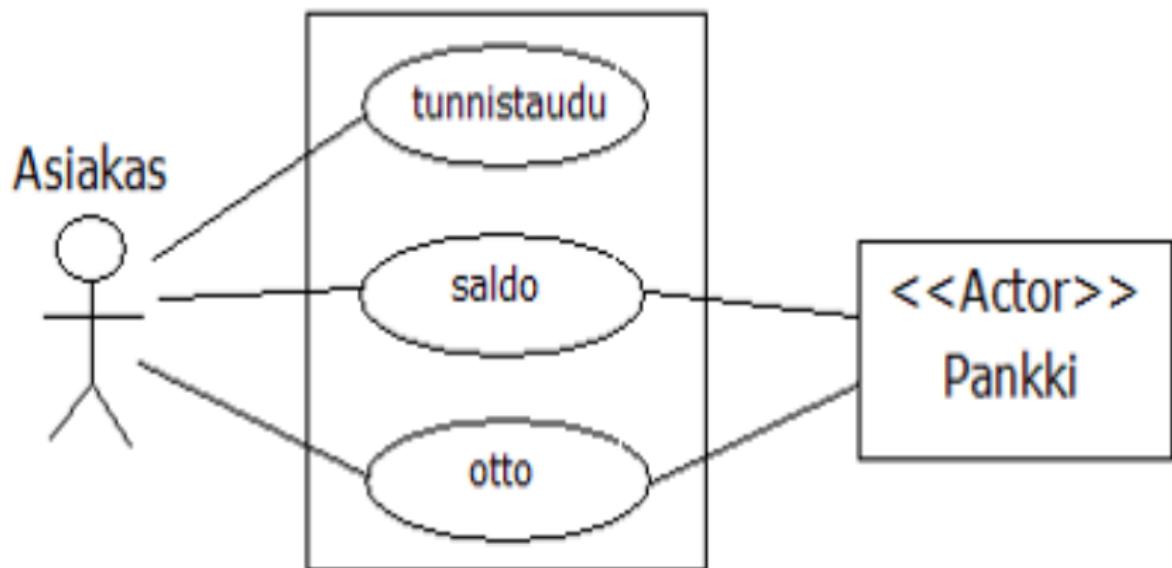
Käyttötapauksen kulku:

- 1 asiakas valitsee otto-toiminnon
- 2 automaatti kysyy nostettavaa summaa
- 3 asiakas syöttää haluamansa summan
- 4 pankilta tarkistetaan riittääkö asiakkaan saldo
- 5 summa veloitetaan asiakkaan tililtä
- 6 kuitti tulostetaan ja annetaan asiakkaalle
- 7 rahat annetaan asiakkaalle
- 8 pankkikortti palautetaan asiakkaalle

Poikkeuksellinen toiminta:

- 4a asiakkaan tilillä ei tarpeeksi rahaa, palautetaan kortti asiakkaalle

Pankkiautomaatti



Kurssikoe

Luokkakaaviot

- ▶ **Määrittelyvaiheen** luokkamalli kuvaailee sovellusalueen käsitteitä ja niiden suhteita
 - ▶ Muodostetaan usein käsiteanalyysin avulla
 - ▶ Käytetään myös nimitystä kohdealueen luokkamalli (engl. domain model)
 - ▶ Ei toiminnallisuutta määrittelyvaiheen luokkakaavioon, ainoastaan olioiden pysyväislaatuiset yhteydet kannattaa merkitä
- ▶ **Suunnittelutasolla** määrittelyvaiheen luokkamalli tarkentuu
 - ▶ Luokat ja yhteydet tarkentuvat
 - ▶ Mukaan tulee uusia teknisen tason luokkia
- ▶ **Suunnittelutason** luokkamalli muodostuu *oliosuunnittelun* myötä
 - ▶ Oliosuunnittelussa kannattaa noudattaa hyväksi havaittuja oliosuunnittelun periaatteita (mm. single responsibility...)
 - ▶ Suunnittelumallit (engl. design patterns) voivat tarjota vihjeitä suunnittelun etenemiseen, näitä ei kurssilla juurikaan käsitelty
 - ▶ Luovuus, kokemus, tieto ja järki tärkeitä suunnittelussa

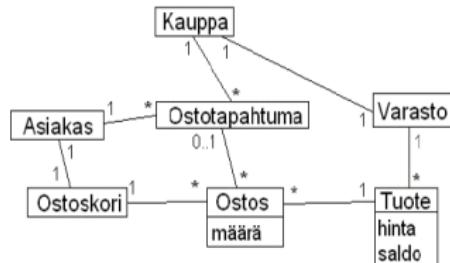
Kurssikoe

Luokkakaaviot

- ▶ Malleista tulee hieman erilaisia määrittely- ja suunnittelutasolla
- ▶ Eli se, minkälainen mallista tulee, riippuu mallintajan näkökulmasta
- ▶ Tämän takia **yleensä pelkkä kaavio ei riitä**: tarvitaan myös tekstiä (tai keskustelua), joka selvittää mallinnuksen taustaoletuksia
 - ▶ Tärkeintä, että kaikki asianosaiset tietävät taustaoletukset
 - ▶ Kun taustaoletukset tiedetään ja rajataan, löytyy myös helpommin "oikea" tapa (tai joku oikeista tavoista) tehdä malli
- ▶ Mallinnustapaan vaikuttaa myös mallintajan kokemus ja "orientaatio"
 - ▶ Kokenut ohjelmoija "näkee kaiken koodina" ja ajattelee väistämättä teknisesti myös abstraktissa mallinnuksessa
 - ▶ Tietokantaihminen taas näkee kaiken tietokannan tauluina jne.

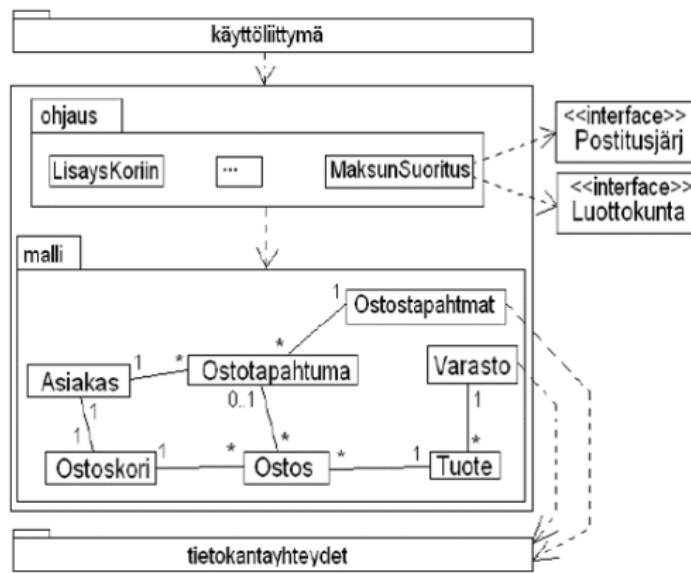
Kurssikoe

Luokkakaaviot



- ▶ Biershop tarkentui abstraktista määrittelytason luokkamallista oliosuunnittelun seurauksena suunnittelutason luokkamalliksi

Määrittelytaso vs. suunnittelutaso



Kurssikoe

Yhteysviivat

- ▶ Määrittelyvaiheen luokkakaavioon ei yleensä merkitä navigointisuuntia
- ▶ Toteutustason luokkakaavioissa, esim. takaisinmallinnettaessa koodia, navigointisuunta voidaan tarvittaessa merkitä, sillä se on näissä tapauksissa tiedossa

→ Assosiaatio

→ Perintä

- - - - - → Implementaatio

- - - - - → Riippuvuus

↔ Kooste

◆ Kompositio

Kurssikoe

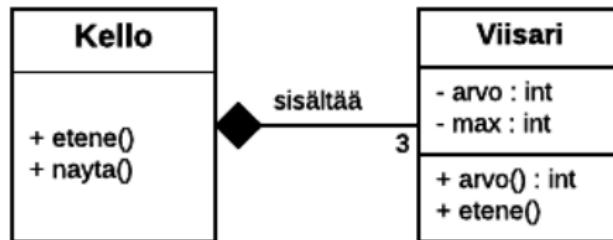
Yhteysviivat

- ▶ Eriiset yhteystyypit ovat herättäneet ajoittain epäselvyyttä
- ▶ **Yhteydellä tarkoitetaan "rakenteista" eli jollakin tavalla pitempiaikaista suhdetta kahden olion välillä**
- ▶ Jos kahden luokan olioiden välillä on tällainen suhde, merkitään luokkakaavioon yhteys (ja yhteyteen liittyvät nimet, roolit, kytkentärajoitteet ym.)
- ▶ Toteutusläheisissä malleissa voidaan ajatella, että luokkien välillä on yhteys jos luokalla on olio- tai viitteitä oliomuuttujana
- ▶ Esim. Kurssilla on private muuttuja joka on tyyppiä Henkilö

Kurssikoe

Yhteysviivat

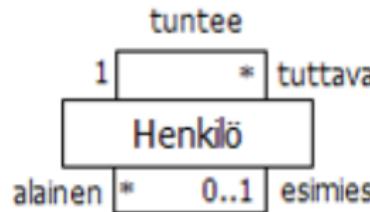
- ▶ Normaali yhteys, eli viiva on varma valinta sillä se ei voi olla "väärin"
- ▶ Usein määrittelyvaiheen luokkamalleissa käytetäänkin vain normaaleja yhteyksiä
- ▶ Yhteyteen voidaan laittaa tarvittaessa navigointisuunta eli nuoli toiseen päähän, tällöin vain toinen pää tietää yhteydestä
- ▶ Jos joku yhteyden osapuolista on olemassa oloriippuva yhteyden toisesta osapuolesta, voidaan käyttää *kompositioita* eli mustaa salmiakkia



Kurssikoe

Rekursiivinen yhteys

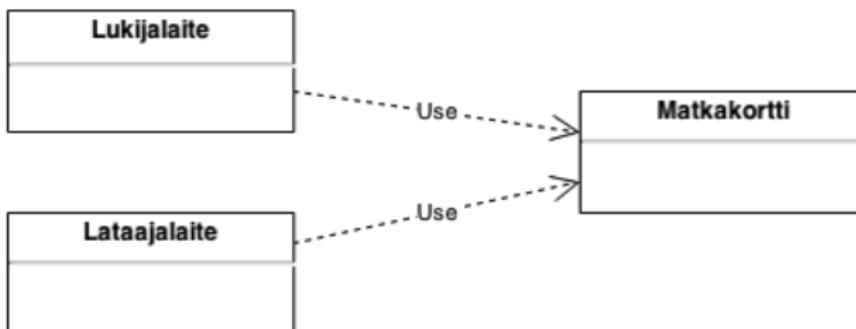
- ▶ Saman luokan olioita yhdistävät yhteydet ovat jokseenkin haasteellisia
- ▶ Henkilö tuntee useita henkilöitä
- ▶ Henkilöllä on mahdollisesti yksi esimies ja ehkä myös alaisia
- ▶ Yhteys tuntee liittää Henkilö-olioon useita Henkilö-olioita roolissa tuttava
- ▶ Henkilö-oliolla voi olla yhteys Henkilö-olioon, roolissa esimies
- ▶ Kuvan alempi yhteys siis sisältääkin kaksi yhteyttä: esimieheen ja alaisiin
- ▶ Rekursiivisten yhteyksien kanssa on syytä käyttää roolinimiä



Kurssikoe

Yhteys vai riippuvuus?

- ▶ **Riippuvuus on jotain normaalia yhteyttä heikompaa,** esim. jos luokan jollakin metodilla on olioarvoinen parametri, voidaan käyttää riippuvuutta (aiemmin luennolla esimerkki AutotonHenkilö riippuu luokasta Auto)
 - ▶ Riippuvuus ei siis ole rakenteinen vaan ajallisesti hetkellinen, esim. yhden metodikutsun ajan kestävä suhde
- ▶ Myös laskareista tuttujen Lataajalaitteen ja Lukijalaitteen suhdetta Matkakorttiin on riippuvuus
 - ▶ Esim. lataajalaite käyttää korttia vain ladatessan kortille lisää rahaa, laite ei muista kortteja pysyvästi



Kurssikoe

Yhteys vai riippuvuus?

- ▶ Riippuvuuden yhteyteen merkitään joskus riippuvuuden tyyppi, esim. edellisessä diassa «use», sillä lukija- ja lataajalaite käyttivät matkakorttia
- ▶ **Riippuvuuden yhteyteen ei merkitä osallistumisrajoitteita**
- ▶ **Riippuvuuden suunta on aina merkittävä!**
- ▶ Riippuvuuden ja yhteyden välinen valinta on joskus näkökulmakysymys
 - ▶ Miten kannattaisi esim. mallintaa Henkilön ja Vuokra-auton suhde?
 - ▶ Vaikka kyseessä voi olla lyhytaikainen suhde, lienee se järkevintä kuvata esim. autovuokraamon tietojärjestelmän kannalta normaalina yhteytenä
- ▶ Riippuvuuksia ei kannata välttämättä edes merkitä...
- ▶ Jotkut mallintajat eivät merkitse riippuvuuksia ollenkaan
- ▶ **Käytä siis riippuvuuksia hyvin harkiten**

Kurssikoe

Osallistumisrajoitteet

- ▶ Kuten sanottu, riippuvuuksiin ei merkitä osallistumisrajoitteita
- ▶ Merkitse kuitenkin osallistumisrajoitteet muissa tapauksissa!
- ▶ **Ei ole hyvä tapa jättää osallistumisrajoituksia merkitsemättä**, sillä ei ole selvää onko kyseessä unohdus vai tarkoitetaanko merkitsemättä jättämisellä sitä että yhteydessä olevien olioiden määrää ei osata määritellä

| | |
|------|--------------------------|
| 0 | Ei yhtään (harvinainen!) |
| 0..1 | Ei yhtään tai yksi |
| 1 | Tasan yksi |
| 0..* | Ei yhtään tai enemmän |
| * | Sama kuin 0..* |
| 1..* | Yksi tai enemmän |
| 5..9 | Viidestä yhdeksään |

Kurssikoe

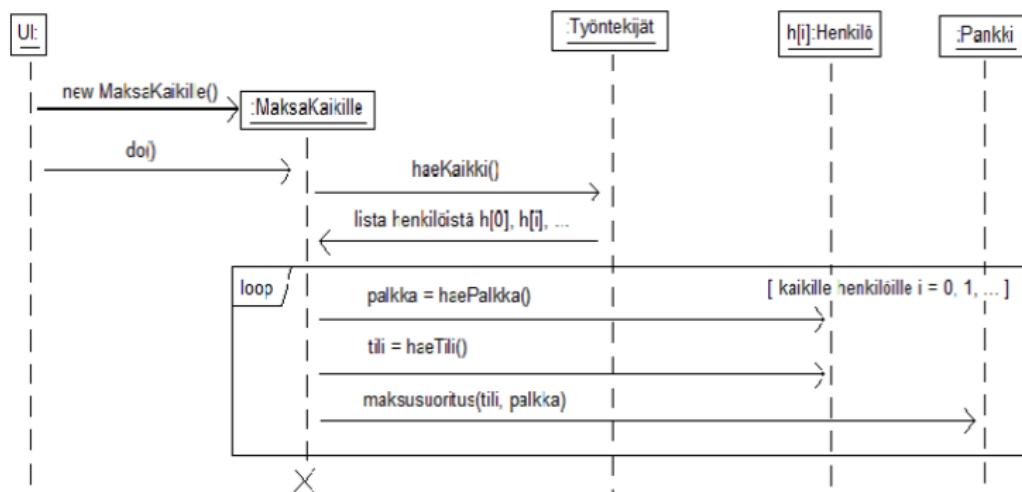
Sekvenssikaaviot

- ▶ Luokkakaavio ei kerro mitään olioiden välisestä vuorovaikutuksesta
- ▶ Vuorovaikutuksen kuvamiseen paras väline on sekvenssikaavio
- ▶ Sekvenssikaavioiden peruskäyttö on melko suoraviivaista
- ▶ **Kun teet sekvenssikaavioita ole erityisen tarkka, että:**
 - ▶ oliot syntyvät "oikeassa kohtaa", eli ylhällä vain alusta asti olemassa olevat oliot
 - ▶ metodikutsu piirretään kutsujasta kutsuttavaan olion
 - ▶ merkitset kaiken toiminnallisuuden kaavioon (myös metodikutsujen aiheuttamat metodikutsut)
 - ▶ parametrit ja paluuarvot on merkitty järkevällä tavalla
- ▶ Sekvenssikaavioiden "vaikeudet" liittyvät toisto-, ehdollisuus- ja valinnaisuuslohkoihin
 - ▶ Näiden käytökelpoisuus on rajallinen, ja niitä kannattaa käyttää harkiten!

Kurssikoe

Sekvenssikaaviot

- ▶ Esimerkki kurssille kuulumattomasta osasta
"Oliosuunnitteluesimerkki: Yrityksen palkanlaskentajärjestelmä"



fin