Ohjelmistotekniikan menetelmät

Luento 7, 8.12.

Tänään luvassa kertausta ja uutta asiaa testauksesta

- Kurssilla on puhuttu erinäisistä ohjelmistojen tuottamiseen liittyvistä asioista
- Ohjelmistotuotantoprosessiin liittyy erilaisia *vaiheita*
  - Vaatimusmäärittely
  - Suunnittelu
  - Toteutus
  - Testaus
  - Ylläpito
- Joskus (perinteisesti) vaiheet tehdään peräkkäin
  - Tällöin puhutaan *vesiputousmallin* mukaan etenevästä ohjelmistotuotantoprosessista
- Vaihtoehtoinen tapa on työskennellä iteratiivisesti, eli toistaa vaiheita useaan kertaan peräkkäin ja limittäin
  - *Ketterä* ohjelmistotuotanto
- Ketteryys on valtaamassa alaa mutta perinteiset tavat myös käytössä
- Ennen kun jatkamme kurssin tärkeimpien teemojen kertaamista, puhutaan hieman testaamisesta

Kertausta testauksesta

- Tarkoitus on varmistaa, että ohjelmisto on *riittävän laadukas* käytettäväksi
- Testaus jakautuu vaiheisiin, joita kutsutaan myös *testaustasoiksi*:
  - **Yksikkötestaus**
    - Toimivatko yksittäiset metodit ja luokat kuten halutaan?
  - **Integraatiotestaus**
    - Varmistetaan komponenttien yhteentoimivuus
  - **Järjestelmä/hyväksymistestaus**
    - Toimiiko kokonaisuus niin kuin vaatimusdokumentissa sanotaan?
- **Regressiotestauksella** tarkoitetaan järjestelmälle muutosten ja bugikorjausten jälkeen ajettavia testejä, jotka varmistavat, että muutokset eivät riko mitään aiemmin tehtyä
  - Regressiotestit koostuvat yleensä yksikkö-, integraatio- ja järjestelmä/hyväksymätesteistä
- Koska testejä joudutaan suorittamaan usein, kannattaa ne automatisoida esim. JUnitin avulla
- Lähes kaikki kurssilla tähän mennessä kirjoitetut testit ovat olleet *yksikkötestejä*

Testien laatu

- Normaalien syötteiden lisäksi on testeissä erityisen tärkeää tutkia testattavien metodien toimintaa virheellisillä syötteillä ja erilaisilla raja-arvoilla
- Testien laatua on kurssin aikana mitattu kolmella tavalla:
  - Testien **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-käännösympäristöä, oli testien laatumittareiden käyttö helppoa...

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 ”automaattisesti” hyvin testattavissa olevaa ja turhia riippuvuuksia ei koodissa yleensä esiinny
- Seuraavalla sivulla Ohjelmoinnin perusteiden viikon 3 tehtävän *Lottoarvonta* ratkaisu
- Luokka *Lottorivi* sisältää nyt riippuvuuden Random-luokkaan
  - Lottorivi-olion konstruktori luo Random-olion, jonka avulla lottonumerot arvotaan
- Luokan testaaminen on nyt erittäin vaikeaa, testeissähän ei pystytä kontrolloimaan mitenkään Random-olion arpomia lukuja
- Lottorivistä tulisi testata ainakin seuraavat asiat
  - lottorivi ei voi sisältää samaa arvoa useampaan kertaan
  - lottorivi voi saada arvoja väliltä 1-39

Lottorivi-olio luo konstruktorissa käyttämänsä Random-olion

```
public class Lottorivi {
	private ArrayList<Integer> numerot;
	Random random;

	public Lottorivi() {
		this.random = new Random();
		this.arvoNumerot();
	}

	public ArrayList<Integer> numerot() {
		return this.numerot;
	}

	public void arvoNumerot() {
		this.numerot = new ArrayList<>();

		while (this.numerot.size() < 7) {
			int numero = random.nextInt(39) + 1;

			if (!numerot.contains(numero)) {
				numerot.add(numero);
			}
		}
	}
}
```

Lottorivi ja riippuvuuden injektointi

- Ongelmana Lottorivin testaamisessa ei sinänsä ole riippuvuus Random-olioon, vaan se, että lottorivin ulkopuolelta ei ole tapaa päästä käsiksi Lottorivin luomaan Random-olioon
- Muutetaan lottorivin konstruktoria seuraavasti:

```
public Lottorivi(Random random) {
    this.random = random;
    this.arvoNumerot();
}
```

- Eli lottorivi muodostetaan nyt luomalla Random-olio, joka annetaan lottoriville konstruktorin parametrina:

```
Random random = new Random();
Lottorivi lotto = new Lottorivi(random);
```

- Olemme nähneet vastaavaa tekniikkaa sovellettavan tällä kurssilla ja esim. Ohjelmoinnin jatkokurssilla moneen kertaan
- Tekniikasta käytetään nimitystä **riippuvuuden injektoint**i (engl. dependency injection)
  - riippuvuutena olevaa olioa ei luoda konstruktorissa tai luokan sisällä
  - konstruktorille annetaan valmiiksi luotu riippuvuus, konstruktori laittaa riippuvuutena olevan olion talteen oliomuuttujaan

Lottorivi, jolle injektoidaan Random-olio

```
public class Lottorivi {
    private ArrayList<Integer> numerot;
    Random random;

    public Lottorivi() {  // olio mahdollista luoda myös vanhaan tapaan
        this(new Random());
    }

    public Lottorivi(Random random) {
        this.random = random;
        this.arvoNumerot();
    }

    public ArrayList<Integer> numerot() {
        return this.numerot;
    }

    public void arvoNumerot() {
        this.numerot = new ArrayList<>();
        while (this.numerot.size() < 7) {
            int numero = random.nextInt(39) + 1;

            if (!numerot.contains(numero)) {
                numerot.add(numero);
            }
        }
    }
}
```

Injektointi mahdollistaa testaamisen

- Koska lottorivi saa käyttämänsä Random-olion konstruktorin parametrina, voimme helposti luoda omia versioitamme Randomista ja käyttää niitä apuna testaamisessa
- Seuraavassa luokka *RandomStub,* joka ”arpoo” sille konstruktorin parametrina annetut luvut

```
public class RandomStub extends Random {
    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);
    }

}
```

- Jos loisimme olion Random arpa = new StubRandom(10,15, 20, 25); ensimmäinen metodikutsu arpa.nextInt(n) palauttaisi luvun 10, seuraava 15 jne
- Koska RandomStub *perii* luokan Random, voidaan sitä käyttää kaikkialla randomin paikalla

Injektoitua RandomStubia hyödyntävä testi

- Testaaminen muuttuu nyt helpoksi
- Seuraavassa testi, joka varmistaa, että sama luku ei esiinny useaan kertaan lottonumerossa, vaikka Random-olio palauttaakin saman luvun monta kertaa

```
public class LottoriviTest {

    @Test
    public void eiSamojaNumeroita(){
        Random randomStubi = new RandomStub(1,1,1,2,3,4,5,6,7);
        Lottorivi rivi = new Lottorivi(randomStubi);
        List<Integer> odotettu = Arrays.asList(2,3,4,5,6,7,8);

        assertSamatNumerot(odotettu, rivi.numerot());

    }

    private void assertSamatNumerot(List<Integer> odotettu, List<Integer> tulos){
        assertEquals(odotettu.size(), tulos.size());
        for (Integer luku : tulos) {
            assertTrue(odotettu.contains(luku));
        }
    }

}
```

Integraatiotestaus

- Yksikkötestaus on käsitteenä suhteellisen hyvin ymmärretty
  - yksikkötestit kohdistuvat yhteen luokkaan
- Järjestelmä/hyväksymätestauksessa taas testataan ohjelmistontarjoamien toiminnallisuutta käyttöliittymän läpi samaan tapaan kuin loppukäyttäjä tulee järjestelmää käyttämään
- Kaikki näiden väliin jäävät testauksen muodot ovat *integraatiotestausta*
- Kurssilla on jo nähty muutamia esimerkkejä integraatiotesteistä
- Viikon 6 laskareissa olleen palkanlaskennan testit ovat oikeastaan integraatiotestejä
  - https://github.com/mluukkai/OTM2015/tree/master/koodi/PalkanlaskentaMalli
- Vaikka testit kohdistuvatkin luokan Palkanlaskenta metodeihin, ne kuitenkin oleellisesti testaavat palkanlaskennasta, henkilöistä ja muutamasta muusta luokasta muodostuvan kokonaisuuden toimintaa
- Myös viikon 6 luennolla käsitellyn Biershopin ohjausolioiden testit ovat integraatiotestejä, sillä ne testaavat useamman luokan yhteistoimintaa

Integraatiotestaus ja ”hankalat” riippuvuudet

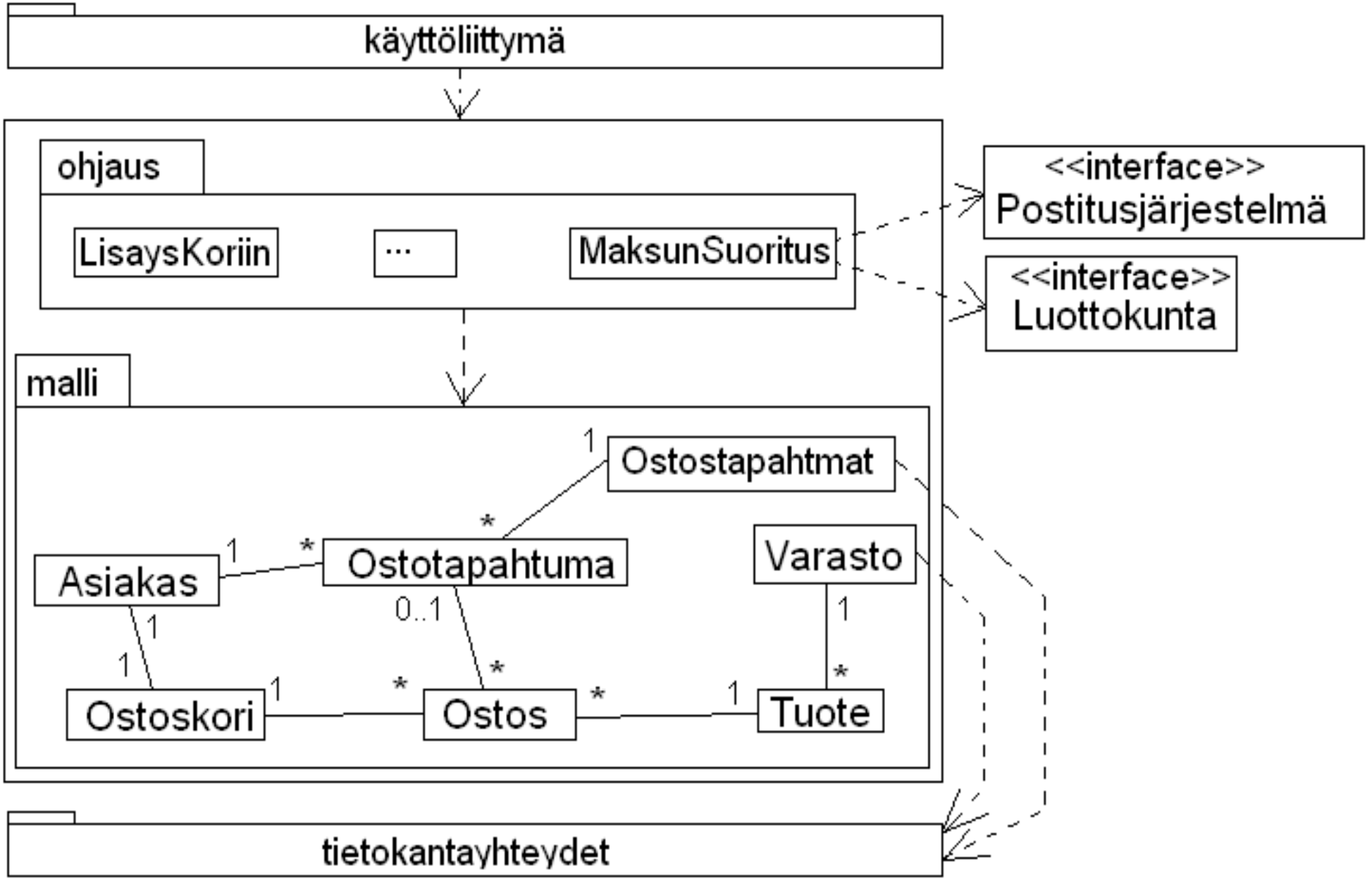
- Testien kannalta on hieman ikävää, jos testattavan järjestelmän jokin komponentti keskustelee tietokannan tai jonkin verkossa olevan komponentin kanssa
  - Testien suoritus hidastuu
  - Testien tulos saattaa olla riippuvainen tietokannan sisällöstä
  - Verkon yhteysongelmat voivat vaikuttaa testien toimivuuteen
- Näissä tilanteissa on järkevää 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

Palkanlaskentaan injektointu MaksupalveluStub

- Palkanlaskennassa palkkojen maksun yhteydessä suoritetaan tilisiirto luokan *MaksupalveluRajapinta* metodia *suoritaTilisiirto* kutsumalla
  - Konkreettisen tilisiirron tekeminen on tietenkin testien kannalta täysin tarpeetonta
- Palkanlaskennan mallivastauksen testeissä käytetäänkin todellisen maksupalvelurajapinnan sijaan stub-olioa:

```
public class MaksupalveluStub extends MaksupalveluRajapinta {
    @Override
    public void suoritaTilisiirto(Maksusuoritus maksu) {
        // stubin ei tarvitse tehdä mitään sillä Palkanlaskenta ainoastaan kutsuu metodia
    }
}

public class PalkanlaskentaTest {
    Palkanlaskenta p;
    @Before
    public void setUp() {
        Map<String, OutputBuilder> outputBuilders = new HashMap<>();
        outputBuilders.put("csv", new CsvBuilder());
        p = new Palkanlaskenta(new MaksupalveluStub(), outputBuilders);
    }

}
```

Kumpula Biershop: tuotteen lisäämistä testaava integraatiotesti



Tuotteen lisäämistä testaava integraatiotesti

- Myös Biershopin testeissä on testauksen kannalta ikävä, tietokantaa käyttävä TuoteDAO eli tietokantayhteydestä huolehtiva olio korvattu testien suorituksen aikana keskusmuistissa toimivalla tietokantayhteysoliolla:
- Tuotteen lisääminen:

```
@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 varastosaldoa, vie tuotteen koriin (eli koriin ilmestyy yksi oikean hintainen tuote)

Järjestelmätestaus

- Järjestelmätestauksessa tulee varmistaa toimiiko ohjelmisto sille vaatimusmäärittelyssä asetettujen vaatimusten mukaan
- Testauksen tulee tapahtua saman rajapinnan läpi, miten loppukäyttäjä tulee järjestelmää käyttämään, eli sovelluksesta riippuen joko komentoriviltä, graafisesta käyttöliittymästä tai webselaimesta
  - Tämä on hieman haastavampaa kuin metodien ja luokkien toimintaa testaavavien JUnit-testien tekeminen mutta ei suinkaan mahdotonta
- Järjestelmätestit kannattaa muodostaa ohjelmiston käyttötapausten mukaan
  - Eli jokaista käyttötapausta kohti tulee tehdä yksi tai useampi testi
- Tarkastellaan Ohjelmoinnin jatkokurssin ensimmäisen viikon tehtävää Laskin
  - http://www.cs.helsinki.fi/group/java/s15-materiaali/viikko8/#128laskin
  - https://github.com/mluukkai/OTM2015/wiki/Laskin
- Testataksemme ohjelmaa tekstikäyttöliittymän kautta, meidän on simuloitava käyttäjän syötteet sekä päästävä käsiksi ohjelman tulostuksiin
- Käyttäjän syötteiden simuloiminen on mahdollista *korvaamalla* ohjelman käyttämä *System.in* syötevirta omalla oliolla
- Vastaavasti voimme korvata *System.out* tulostusvirran siten, että tulostus meneekin testin generoimaan olioon ruudun sijaan

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
- Seuraavassa testi laskimen summaustoiminnolle

```
public class LaskinTest {
    ByteArrayOutputStream tulosvirta;

    @Before
    public void setUp() {
        tulosvirta = new ByteArrayOutputStream();
        System.setOut(new PrintStream(tulosvirta));
    }

    @Test
    public void summaJaLopetus() {
        String syote = "summa\n"+"1\n"+"2\n"+"lopetus\n";
        System.setIn(new ByteArrayInputStream(syote.getBytes()));

        Laskin laskin = new Laskin();
        laskin.kaynnista();

        String tulostus = tulosvirta.toString();

        assertTrue(tulostus.contains("summa 3"));
        assertTrue(tulostus.contains("laskuja laskettiin 1"));

    }
}
```

Laskimen testi, joka korvaa syöte- ja tulostevirran

- Testin simuloitu syöte on merkkijono, jossa yksittäiset syöterivit päättyvät rivinvaihtoon

```
String syote = "summa\n"+"1\n"+"2\n"+"lopetus\n";
```

- Merkkijonosta luodaan bittivirta, joka asetetaan *ohjelman syötevirran* arvoksi System.setIn(new ByteArrayInputStream(syote.getBytes()));
- setUp-metodissa testi asettaa uuden ByteArrayOutputStream-olion *ohjelman tulosvirraksi* metodin System.set avulla

```
tulosvirta = new ByteArrayOutputStream();
System.setOut(new PrintStream(tulosvirta));
```

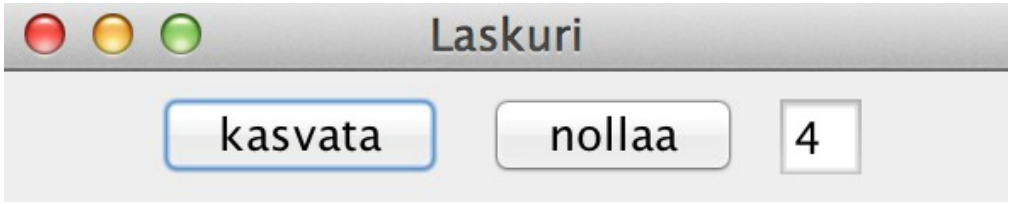
- Laskimen suorituksen jälkeen tulosvirtaolio muutetaan toString-metodilla merkkijonoksi ja tarkastetaan, että ohjelman tulostus on odotetun kaltainen

```
String tulostus = tulosvirta.toString();
assertTrue(tulostus.contains("summa 3"));
assertTrue(tulostus.contains("laskuja laskettiin 1"));
```

- 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
  - Tällöin testiä varten oltaisiin voitu luoda LukijaStub-olio, johon olisi sijoitettu simuloitu syöte

Graafisen käyttöliittymän testaus

- 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
- Testaus helpottuu oleellisesti, jos käyttöliittymän komponenteille asetetaan 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

- Tämän hetken johtava työkalu Swing-sovellusten testaamiseen on *AssertJ*
  - http://joel-costigliola.github.io/assertj/assertj-swing.html
- Testit näyttävät seuraavalta:

```
public class LaskuriTest extends AssertJSwingJUnitTestCase {
    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

```
window.textBox("tulos").requireText("1");
```

Koe

- Kurssikoe keskiviikkona 16.12. kello 17.00 salissa A111. Kokeessa vastausaikaa 2 tuntia 30 minuuttia
- Nippelitason detaljien sijasta kokeessa arvostetaan enemmän sovellusosaamista
  - Tosin eräät detaljit (joista mainitaan myöhemmin) ovat tärkeitä
- Kokeeseen saa tuoda mukanaan ***itse tehdyn, käsin kirjoitetun*** yhden A4-arkin (saa olla 2-puolinen) 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

Mikä on tärkeintä kurssilla/kokeessa?

- Kokonaiskuva: *Mitä? Miksi? Milloin? Missä? Miten?*
- Ohjelmistotuotantoprosessin vaiheet on syytä osata
- Testaamisen rooli ohjelmistotuotantoprosessissa ymmärrettäv
  - testejäkin saattaa joutua kirjoittamaan
  - kannattaa kerrata laskareiden testaukseen liittyvät asiat
- UML:sta ylivoimaisesti tärkeimpiä ovat
  - Luokkakaaviot
  - Sekvenssikaaviot
  - Käyttötapauskaaviot
- Käyttötapausmallinnus myös tärkeää, ks. kalvot 25-28
- *Käsiteanalyysi*, eli määrittelyvaiheen luokkamallin muodostaminen tekstistä
- Määrittelystä suunnitteluun eli *oliosuunnittelu*:
  - Luokkamallin tarkentuminen, uudet teknisen tason luokat
  - Toiminnan kuvaaminen sekvenssikaaviona
- *Takaisinmallinnus*, eli valmiista koodista luokka- ja sekvenssikaavioiden teko

Oliosuunnittelusta

- 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 huonoon suunnitteluun/koodihajuun: **refaktorointi**
  - Muutetaan koodin rakennetta parempaan suuntaan muuttamatta toiminnallisuutta

Oliosuunnittelu ja koe

- Luennolla 6 Biershop-esimerkin yhteydessä katsottiin hieman vastuupohjaista (responsibility driven) -oliosuunnittelutekniikka
  - Noudattaa kaikkia yleisiä oliosuunnitteluperiaatteita
- Viime viikon 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 refaktorointi paremmaksi
  - Ymmärrys, miksi periaatteet ovat olemassa ja milloin niitä saa rikkoa

Olioiden pysyväistalletus ja MongoDB

- Missä määrin aihe tulee osata kokeessa?
- Periaatteellisella tasolla tulee ymmärtää
  - Käsitteet: json, avain-arvo-pari, dokumentti ja kokoelma
  - Miten Mongo tallettaa Java-olioita
  - Olioiden yhteyksien (yhden suhde moneen ja monen suhde moneen) mallintaminen Mongossa
  - Periaatteet sille, minkälaisia kyselyitä mongolla voidaan tehdä
- Morphia-kirjastoa ei tarvitse osata käyttää olioiden talletukseen eikä kyselyiden tekemiseen
- Myöskään viikon 5 laskarien tehtävien 6 ja 7 kaltaisia kyselyjä ei tarvitse osata tehdä

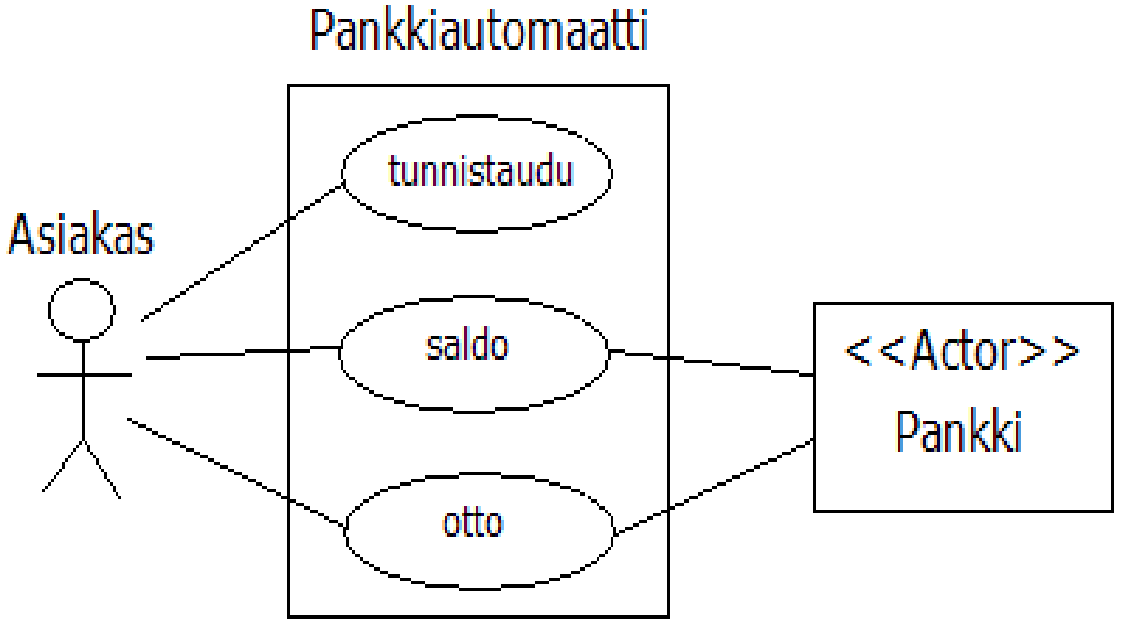
- **Sivuhuomio**: kurssin aikana ilmestynyt versio 3.2. toi Mongoon radikaalin uudistuksen, *$lookup*-operaation aggregaattikyselyihin. $lookup mahdollistaa kyselyt, joissa liitetään yhteen dataa monista kokoelmista, eli saman toiminnallisuuden, mitä relaatiotietokantojen join-operaatio tarjoaa
  - https://www.mongodb.com/blog/post/thoughts-on-new-feature-lookup

Käyttötapausmalli

- Kertausta muutamasta käyttötapausmallin ydinkohdasta

Käyttötapaukset ja käyttötapauskaaviot

- Pankkiautomaatin käyttötapaukset ovat *tunnistaudu, saldo* ja *otto*
- Käyttötapausten *käyttäjät* (engl. actor) eli toimintaan osallistuvat tahot ovat *Asiakas* ja *Pankki*
- Käyttötapausten ja käyttäjien suhdetta kuvaa UML:n **käyttötapauskaavio**
  - Tikku-ukolle vaihtoehtoinen tapa merkitä käyttäjä on laatikko, jossa tarkenne <<actor>>
  - Käyttötapausellipsiin yhdistetään viivalla kaikki sen käyttäjät
  - Kuvaan ei siis piirretä nuolia!



Käyttötapaukset ja käyttötapauskaaviot

- Huomattavaa on, että koska käyttötapausmallia käytetään vaatimusmäärittelyssä, *ei ole syytä puuttua järjestelmän sisäisiin yksityiskohtiin*
  - Pankkiautomaatin sisäiseen toimintaan ei oteta kantaa
  - Mitään automaatin sisäistä (esim. tietokantaa tai tiedostoja) ei merkitä kaavioon
  - *Tarkastellaan ainoastaan, miten automaatin toiminta näkyy ulospäin*
- Itse käyttötapauksen sisältö kuvataan *tekstuaalisesti*, käyttäen esim. Alistair Cockburnin käyttötapauspohjaa
- Käyttötapauskaavio toimii hyvänä yleiskuvana, mutta vasta tekstuaalinen kuvaus määrittelee toiminnan tarkemmin
  - Seuraavalla sivulla käyttötapauksen *Otto* tarkennus
- Seuraavan sivun käyttötapauskuvaus ei ota kantaa käyttöliittymään
- Seuraavana vaiheena saattaisi olla tarkemman, käyttöliittymäspesifisen käyttötapauksen kirjoittaminen

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

Poikkeuksellinen toiminta:

4a asiakkaan tilillä ei tarpeeksi rahaa, palautetaan kortti asiakkaalle

Luokkakaaviot

- Seuraavassa katsaus tärkeimpiin luokkakaavioihin liittyviin seikkoihin

Kahden abstraktiotason luokkamalleja

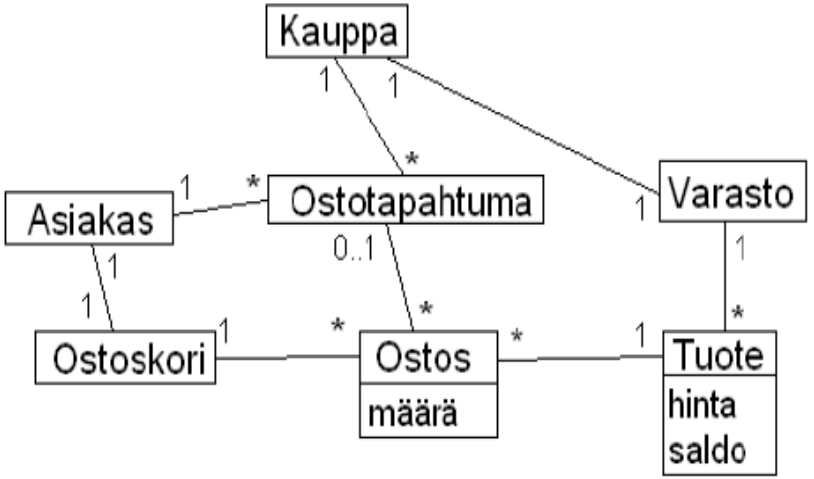
- Olemme huomanneet, että luokkamallia käytetään kahdella hieman erilaisella abstraktiotasolla
- *Määrittelyvaiheen luokkamalli kuvailee sovellusalueen käsitteitä ja niiden suhteita*
  - Muodostetaan usein käsiteanalyysin avulla
  - Käytetään myös nimitystä *kohdealueen luokkamalli* (engl. domain model)
  - **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

Kahden abstraktiotason luokkamalleja

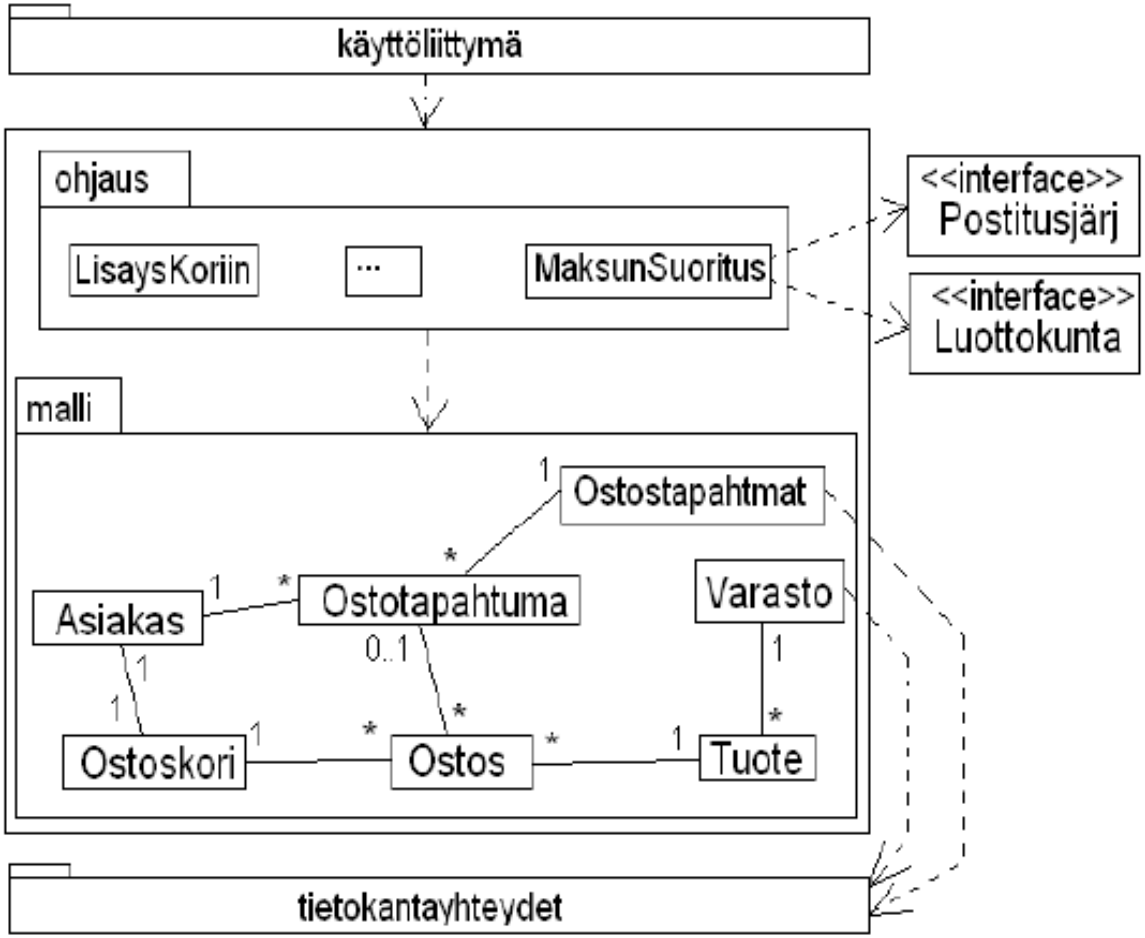
- Malleista tulee hieman erilaisia käsitteellisellä- ja suunnittelutasolla
  - Esim. onko Monopolin Pelaaja-luokka ”ihmisen malli” vai Javalla toteutettava olio?
  - Tämä aiheuttaa joskus hämminkiä
- Eli se, minkälainen mallista tulee, riippuu mallintajan näkökulmasta
- Tämän takia yleensä pelkkä kaavio 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.

Määrittelytaso vs. suunnittelutaso



- Viime viikolta:
- Biershop tarkentui abstraktista määrittelytason luokkamallista oliosuunnittelun seurauksena suunnittelutason luokkamalliksi



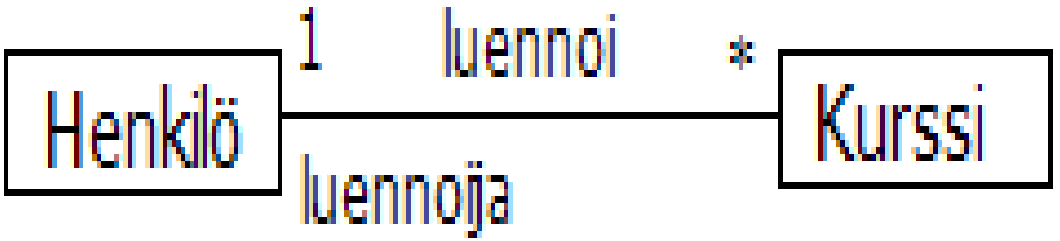
Yhteys ja kompositio

- Erilaiset yhteystyypit ovat herättäneet ajoittain epäselvyyttä
- Yhteydellähän 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 olioviite tai viitteitä oliomuuttujana*

```
public class Henkilö {

 ...

}
```

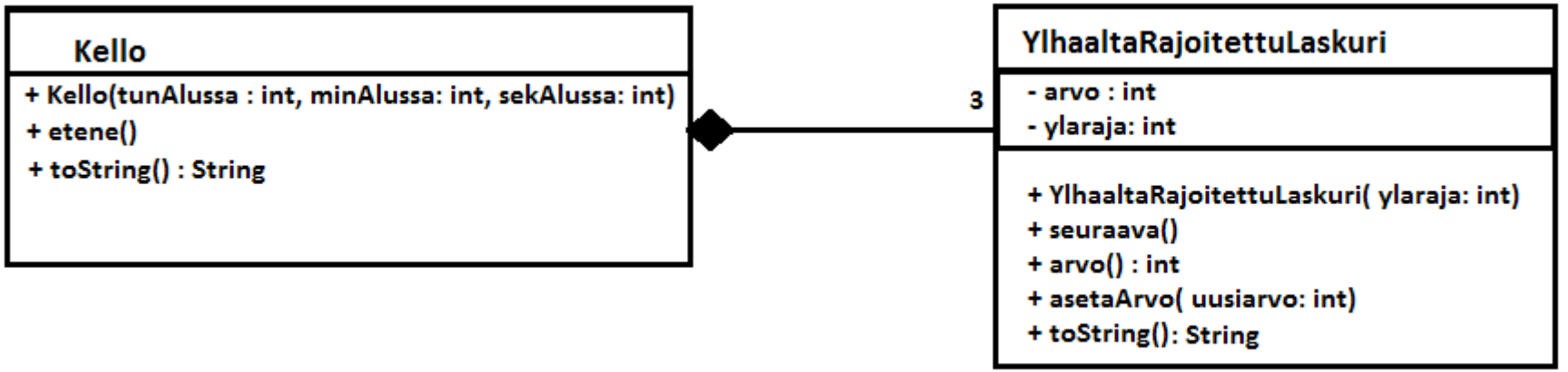
```
public class Kurssi{

    private Henkilo luennoija;

    …

}
```

- Jokaisella Kurssi-oliolla on yhteys yhteen Henkilö-olioon, joka on Kurssin suhteen roolissa luennoija



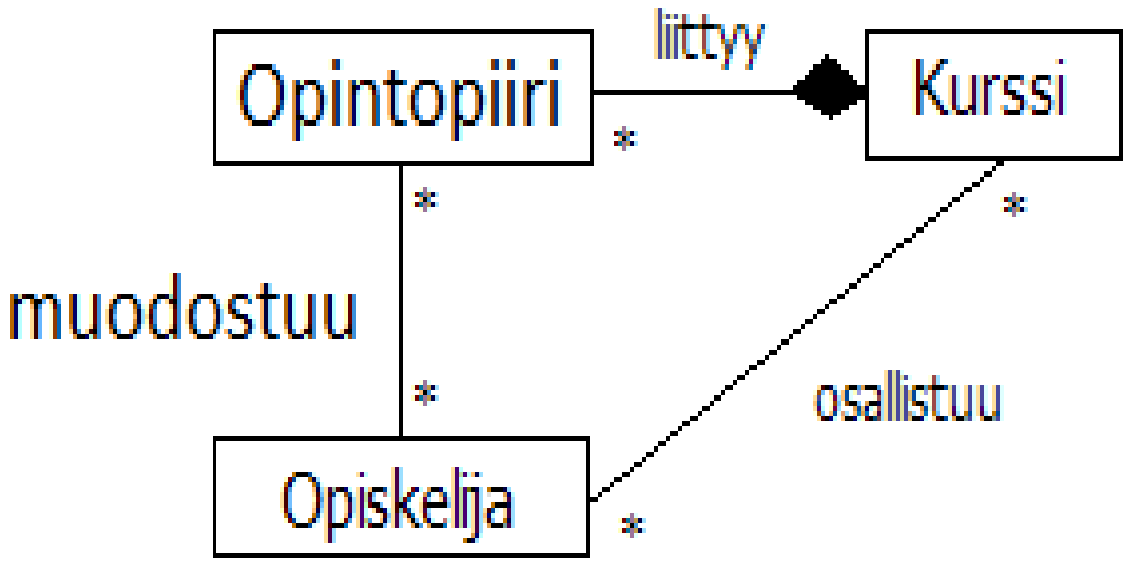
Yhteys ja kompositio

- 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 *olemassaoloriippuva* yhteyden toisesta osapuolesta, voidaan käyttää *kompositioita* eli mustaa salmiakkia
  - Laskuri liittyy tiettyyn Kello-olioon ja jos kello tuhotaan, tuhoutuu myös laskuri
  - *Merkitään salmiakki niihin olioihin, joista olemassaolo riippuu*
  - Ei olisi väärin ilmaista yhteyksiä normaaleina viivoina mutta musta salmiakki tarjoaa hyödyllistä lisätietoa yhteyden laadusta



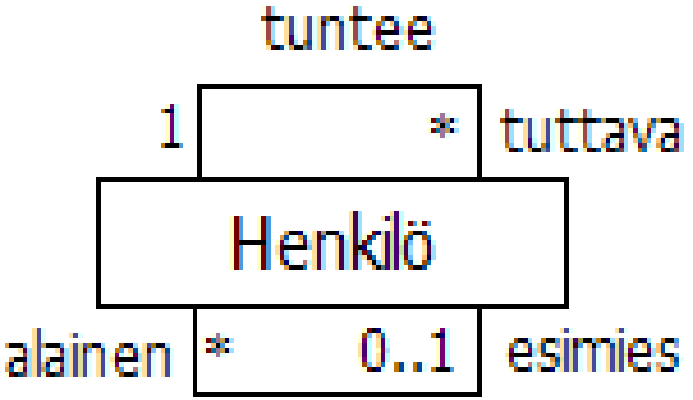
Yhteys ja kompositio

- Opintopiiri liittyy tiettyyn kurssiin
  - Kurssin loputtua myös opintopiiri lakkaa olemasta
  - Olemassaoloriippuvuus, eli voidaan käyttää kompositiota
- Joukko opiskelijoita muodostaa opintopiirin
  - Opiskelija voi kuulua myös useampaan opintopiiriin
  - Normaali yhteys



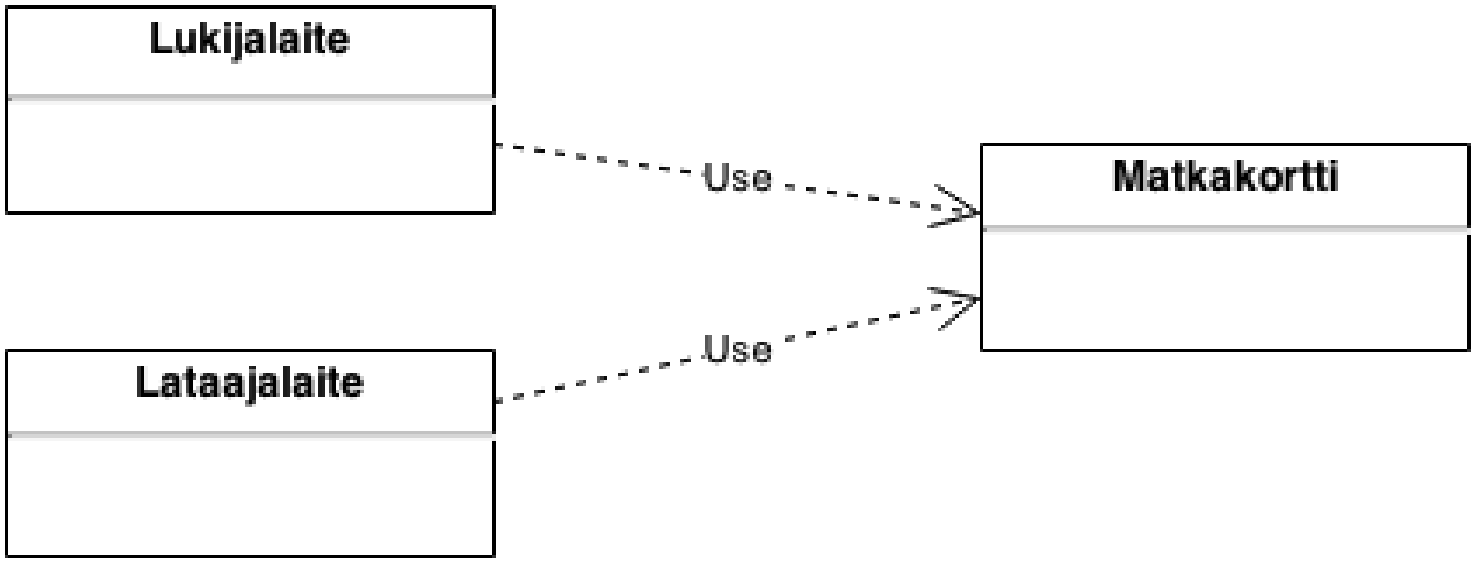
Rekursiivinen yhteys

- Saman luokan olioita yhdistävät yhteydet ovat joskus 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, joka on *roolissa esimies*
  - Eli Henkilöön voi liittyä esimies
- Henkilö-oliolla voi olla yhteys useaan Henkilö-olioon, jotka ovat *roolissa alainen*
  - Eli Henkilöllä voi olla useita alaisia
- Kuvan alempi yhteys siis sisältääkin kaksi yhteyttä: esimieheen ja alaisiin
- **Rekursiivisten yhteyksien kanssa on syytä käyttää roolinimiä**, pelkkä yhteyden nimi ei yleensä riitä ilmaisemaan selkeästi yhteyden laatua



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 suhden Matkakorttiin on riippuvuus
  - Esim. lataajalaite käyttää korttia vain ladatessan kortille lisää rahaa, laite ei muista kortteja pysyvästi



Yhteys vai riippuvuus?

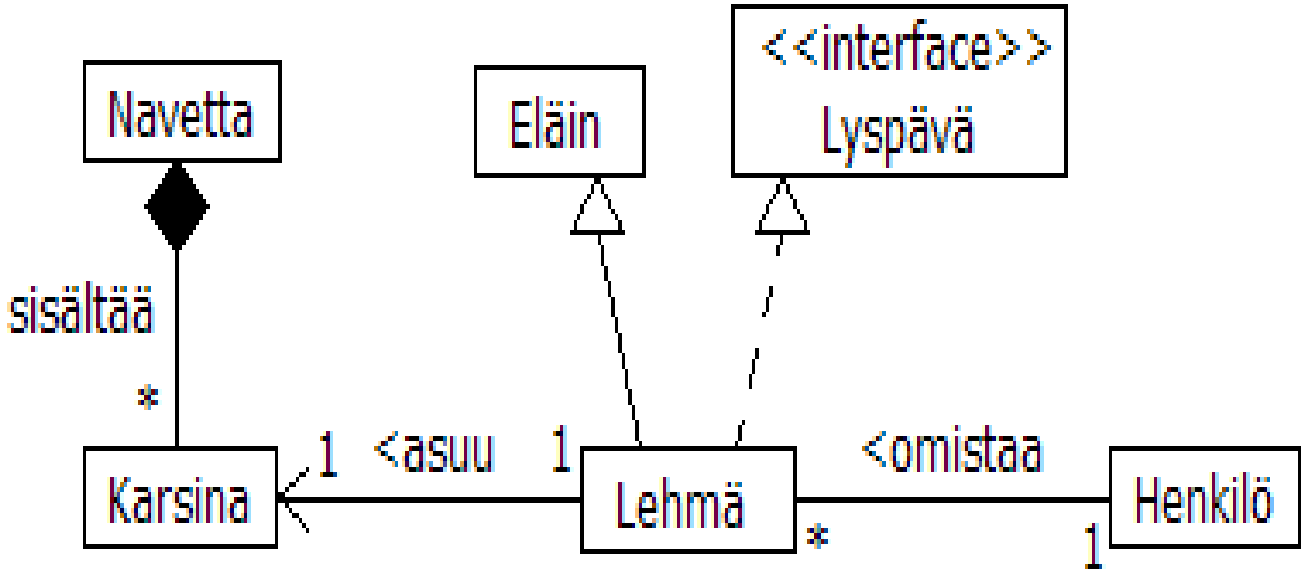
- Riippuvuuden yhteyteen merkitään joskus riippuvuuden tyyppi, esim. edellisessä kuvassa use, sillä lukija- ja lataajalaite käyttivät matkakorttia
- **HUOM:**
  - **Riippuvuuden yhteyteen ei merkitä osallistumisrajoitteita**
  - **Riippuvuuden suunta on aina merkittävä!**

- 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, ainakaan kaikkia
- Jotkut mallintajat eivät merkitse riippuvuuksia ollenkaan
- **Käytä siis riippuvuuksia hyvin harkiten**
- Jos luokkien välillä on yhteys, ei niille tarvitse enää merkitä riippuvuutta sillä yhteyden olemassaolo tarkoittaa että luokat ovat myös riippuvaisia toisistaan

Nuolenpäät on osattava merkitä oikein!

- Lehmä on Eläin-luokan *aliluokka:* valkoinen kolmio yliluokan päässä, normaali viiva
- Lehmä *toteuttaa* Lypsävä-*rajapinnan:* valkoinen kolmio rajapinnan päässä, katkoviiva
- Navetta sisältää karsinoita jotka ovat olemassaoloriippuvaisia navetasta: musta salmiakki olemassaolon takaavan luokan päässä
- Lehmä asuu karsinassa, lehmä tuntee karsinansa, mutta karsina ei lehmää: yhteydessä navigointinuoli lehmästä karsinaan päin
- Lehmä ja omistaja (joka on Henkilö-olio) tuntevat toisensa: normaali yhteys ilman nuolia



Navigointisuuntien merkitseminen

- Edellisellä sivulla merkitsimme Karsinan ja Lehmän väliseen yhteyteen navigointisuunnan
- Navigointisuuntia ei välttämättä merkitä ollenkaan, monisteesta:
  - *Navigointimäärittelyt kuuluvat matalan tason ohjelmointiläheiseen kuvaukseen. Korkeamman abstraktiotason kuvauksissa ne jätetään teknisen toteutuksen yksityiskohtina usein kokonaan kuvaamatta*
- Määrittelyvaiheen luokkakaavioon ei yleensä merkitä navigointisuuntia
- Toteutustason luokkakaavioissa, esim. takaisinmallinnettaessa koodia, navigointisuunta yleensä merkitään koska se on näissä tapauksissa tiedossa

Yhteyden osallistumisrajoitteet täytyy merkitä

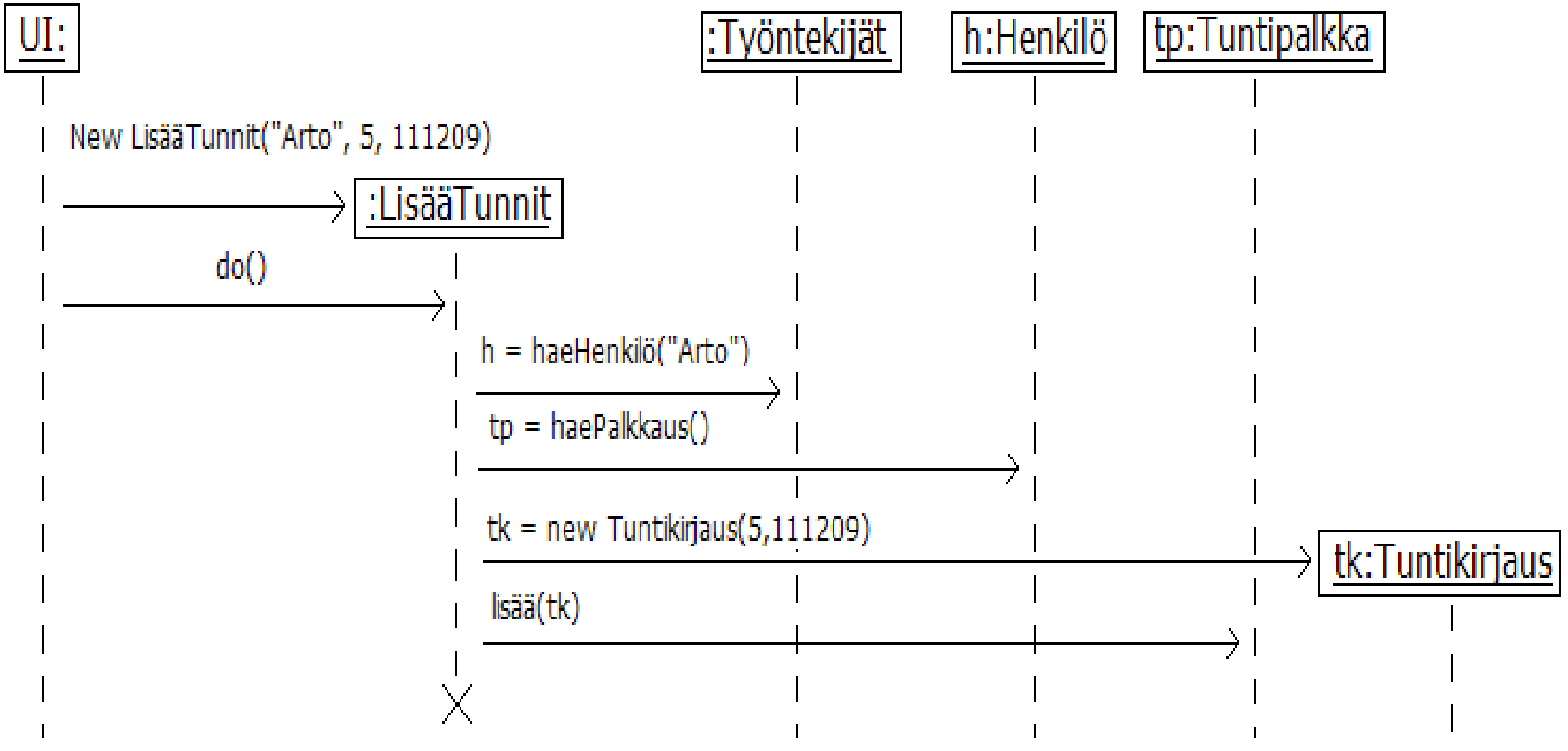
- Kalvon 40 kuvassa:
  - yhteen lehmäolioon liittyy tasan yksi Henkilöolio omistajana
  - Yhteen henkilöolioon taas voi liittyä monta Lehmäolioa yhteydessä omistaa
  - Mustan salmiakin yhteydessä ei tarvita osallistumisrajoitetta sillä salmiakki tarkoittaa käytännössä ykköstä, eli karsina sijaitsee yhdessä navetassa
- **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

- **Riippuvuuksiin** (tai perimiseen) **ei tule merkitä osallistumisrajoituksia** sillä kyseessä on luokkien ei olioiden välinen suhde

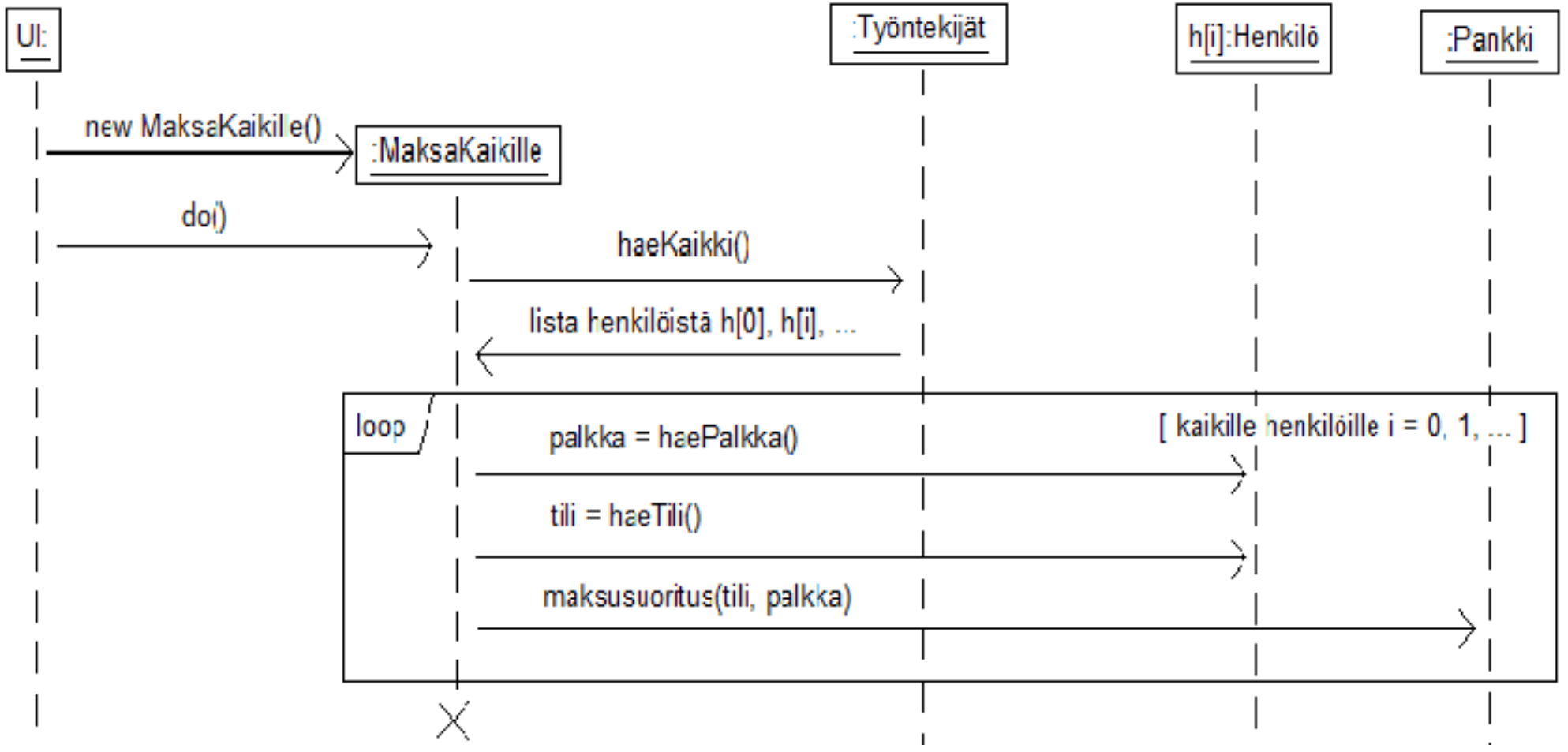
Sekvenssikaavioista

- Luokkakaavio ei kerro mitään olioiden välisestä vuorovaikutuksesta
- Vuorovaikutuksen kuvaamiseen paras väline on sekvenssikaavio
  - Kurssilla vilkaistiin myös kommunikaatiokaavioita, niitä ei kuitenkaan kokeessa tarvitse osata
- Sekvenssikaavioiden peruskäyttö on melko suoraviivaista
- Kun teet sekvenssikaavioita ole erityisen tarkka, että:
  - oliot syntyvät ”oikeassa kohtaa”, eli ylhäällä vain alusta asti olemassa olevat oliot
  - metodikutsu piirretään kutsujasta kutsuttavaan olioon
  - merkitset kaiken toiminnallisuuden kaavioon (myös metodikutsujen aiheuttamat metodikutsut)
  - parametrit ja paluuarvot on merkitty järkevällä tavalla
- Sekvenssikaavioiden ”vaikeudet” liittyvät toisto-, ehdollisuus- ja valinnaisuuslohkoihin
  - Näiden käyttökelpoisuus on rajallinen, ja niitä kannattaa käyttää harkiten
- Pari esimerkkiä, joista käy ilmi oleellinen sekvenssikaavioihin liittyvä
  - Molemmat ovat monisteen liitteenä A olevasta kurssille kuulumattomasta osasta *”Oliosuunnitteluesimerkki: Yrityksen palkanlaskentajärjestelmä”*

- Olio UI luo ensin luokan LisääTunnit-olion ja kutsuu sen do()-metodia
- do():n suoritus kutsuu Työntekijät-luokan metodia haeHenkilö
- Paluuarvona olevan Henkilö-olion *h* metodia haePalkkaus kutsutaan
- Luodaan uusi Tuntikirjaus-olio *tk*, joka lisätään haePalkkaus()-metodin palauttamalle oliolle *tp* metodikutsulla lisää()
- Lopuksi LisääTunnit-luokan olio tuhoutuu



- Olio UI luo ensin luokan MaksaKaikille-olion ja kutsuu sen do()-metodia
- do():n suoritus kutsuu Työntekijät-olion metodia haeKaikki() joka palauttaa listan Henkilö-olioita
  - Merkitään palautettua oliolistaa h[0], h[1], ..., eli lyhyesti h[i], i=0,1,2, ...
- Loopissa jokaiselle Henkilö-oliolle h[i] kutsutaan metodeja haePalkka() ja haeTili()
  - Sekä tehdään Pankki-oliolle metodikutsu maksusuoritus(), jonka parametreina on henkilöltä kysytyt palkka ja tilinumero



END