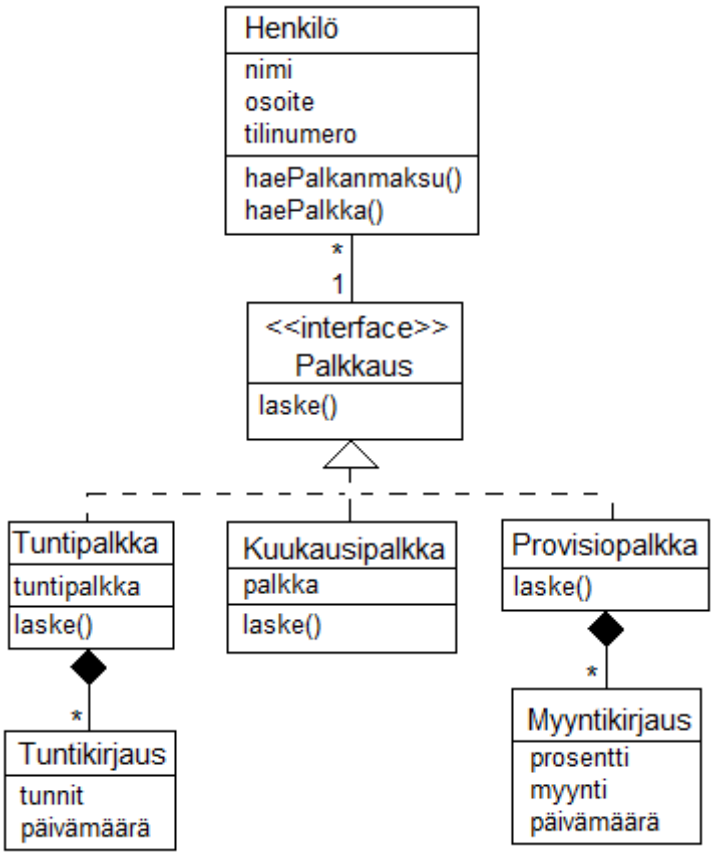
Oliosuunnitteluesimerkki: Yrityksen palkanlaskentajärjestelmä

Matti Luukkainen
10.12.2009

Tässä esitetty esimerkki on mukaelma ja lyhennelmä *Robert Martinin* kirjasta *Agile and Iterative Development* löytyvästä esimerkistä. Martinin esimerkki on huomattavasti laajempi, kattaen kirjasta peräti 103 sivua. Esimerkki ei varsinaisesti ”kuulu” syksyn 09 OhMa-kurssille, mutta on erittäin hyvä havainnollistus muutamasta oliosuunnitteluperiaatteesta ja näinollen hyvää oppia elämää varten. Esimerkki tulee todennäköisesti OhMa-monisteen syksyn 2010 versioon.

Esimerkki on kirjoitettu suhteellisen nopeasti, joten tekstissä on varmasti kirjoitusvirheitä. Pahoittelen virheistä ja tekstin lukeminen tapahtukoon lukijan omalla vastuulla. Varmuudella kirjoitusvirheettömiä ovat ainoastaan tyhjät paperit.

Yrityksen työntekijä (kuvataan luokkana Henkilö) voi saada joko kuukausi- tunti- tai provisiopalkkaa. Päätetään mallintaa tilanne liittämällä kuhunkin Henkilö-olioon Palkkaus-rajapinnan toteuttava olio (käytetty luennolla 7 esiteltyä ideaa erottaa henkilö ja henkilön rooli, tässä tapauksessa “palkkausrooli”):



Jokainen palkkaustyyppi siis osaa laskea palkan, eli käytännössä palauttaa palkkana olevan euromäärän. Tuntipalkkaan liittyy Tuntikirjauksia, eli päivämäärän ja työtuntien yhdistelmiä. Provisiopalkkaan taas liittyy Myyntikirjauksia, eli myyntisumman, provisioprosentin ja päivämäärän yhdisteitä. Provisiopalkatulla henkilöllä palkka muodostuu myyntisummasta kerrottuna provisioprosentilla. Provisioprosentti voi olla erilainen eri myyntien yhteydessä.

Metodien laske() toteutus on aika ilmeinen:

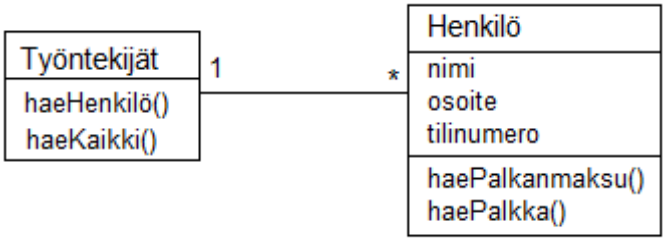
```
class Kuukausipalkka implements Palkkaus{
	double palkka;

	double laske(){ return palkka; }
}

class Tuntipalkka implements Palkkaus {
	double tuntipalkka;
	ArrayList<Tuntikirjaus> tunnit;

	double laske(){
		double p = 0;
		for ( Tuntikirjaus t : tunnit ){
			p += t.getTunnit() * tuntipalkka;
		}
		return p;
	}
}

class Provisiopalkka implements Palkkaus {
	ArrayList<Myyntikirjaus> myynnit;

	double laske(){
		double p = 0;
		for ( Myyntikirjaus m : myynnit){
			p += m.getProsentti() * m.getMyynti();
		}
		return p;
	}
}
```

Henkilö-olio tietää palkkansa kysymällä sitä omalta Palkkaus-rajapinnan toteuttamalta palkkauksen yksityiskohdista vastaavalta oliolta, eli luokan Henkilö-metodi haePalkka:

```
class Henkilö{
	Palkkaus palkkaus;
	double haePalkka(){
		double palkka = palkkaus.laske();
		return palkka;
	}
	Palkkaus haePalkanmaksu(){
		retutn palkkaus;
	}
}
```

Koodista huomaamme, että Henkilö-luokalla on myös metodi haePalkanmaksu(). Metodi palauttaa viitteen Henkilö-olioon liittyvään Palkkaus-rajapinnan toteuttavaan olioon. Metodin käyttötarkoitus selviää myöhemmin.

Järjestelmässä on yksi luokan Työntekijät olio, jonka tehtävänä on tuntea ja kaikki yksittäiset Henkilö-oliot.



Luokalla on metodi

Henkilö haeHenkilö(String nimi)

jonka avulla saadaan selville viite nimellä haettuun henkilöön, sekä metodi

ArrayList<Henkilö> haeKaikki();

joka palauttaa viitteet kaikkiin Henkilö-olioihin listana.

Käyttöliittymä (seuraavalla sivulla olevassa kuvassa pakkaus UI) on eristetty täysin edellä esitetyistä sovelluslogiikan olioista (eli Henkilöistä ja niihin liittyvistä Palkkaus-rajapinnan toteuttavista oliosta). Eristäminen on toteutettu käyttämällä ns. *komentoperiaatetta* (engl. *command pattern*), joka on yksi hyvin tunnettu suunnittelumalli. Komentoperiaatteen idea on melko samantapainen kuin luennoilla ja monisteessa esitelty ohjausperiaate, eli periaate, jonka mukaan jokaisella (tai ainakin osalla) käyttötapauksista on oma luokkansa, jonka oliot huolehtivat käyttötapauksen toiminnallisuuden aikaansaamisesta sovelluslogiikan olioiden yhteistyönä. Ohjausolioissa ja komento-oliossa on kuitenkin muutamia huomattavia eroja kuten kohta käy ilmi.

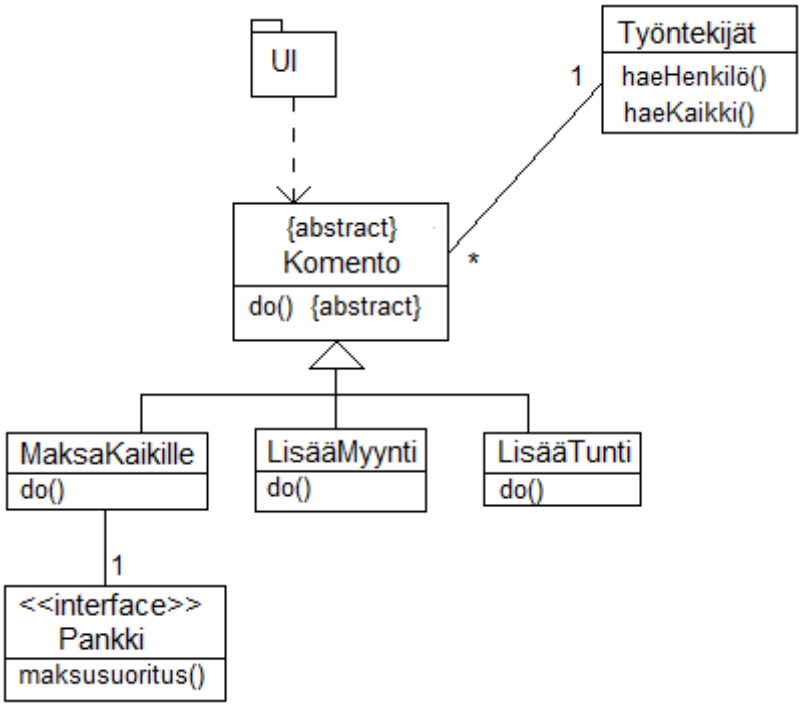
Komentoperiaatteessa on olemassa abstrakti luokka Komento, joka määrittelee ainoastaan yhden abstraktin metodin do(). Metodilla ei ole mitään parametreja. Jos on tarvetta kumota komentoja, voidaan luokkaan myös määritellä metodi undo(), jälleen ilman parametreja.

Jokainen erityinen komento määritellään abstraktin luokan Komento perivänä luokkana, joista jokainen toteuttaa metodin do() omalla tavallaan.

Toteutetaan nyt komennot seuraaviin toiminnallisuuksiin (joita voi itseasiassa ajatella järjestelmän käyttötapauksina):

- maksa kaikille palkka
- lisää työntekijälle työtunteja
- lisää työntekijälle myyntisumma ja siihen liittyvä provisioprosentti

Jokaista edellistä vastaavat luokat siis määritellään perimään Komento. Luokkakaavio seuraavassa:



Näistä luokka MaksaKaikille tuntee rajapinnan Pankki, jonka avulla se pystyy tekemään maksusuorituksia työntekijöiden tileille. Rajapinta Pankki on oikeastaan fasadi, jonka taakse on piilotettu kaikki se koodi, joka ottaa yhteyttä pankin tietojärjestelmään.

Käyttöliittymän kannalta toiminnallisuuden suorittaminen on erittäin helppoa. Täytyy ainoastaan luoda sopiva Komento-luokan aliluokan olio ja kutsua sen metodia do().

Mistä komennon aliluokat saavat syötteensä? Nythän ainoalla metodilla do ei ole mitään parametreja!

Syöte tulee konstruktorin parametreina. Ideana on, että *jokaista* toimenpidettä varten luodaan oma komennon aliluokan olio, jolle sitten kutsutaan do(). Luotaessa komennon aliluokan olioa, syöte annetaan konstruktorin parametreina.

Seuraavassa Javana se, mitä (joku) käyttöliittymäolio tekee kunkin toiminnallisuuden yhteydessä:

Suoritetaan palkanmaksu:

```
Komento k = new MaksaKaikille();
k.do();
```

Lisätään työntekijälle työtunteja:

Oletetaan tässä, että on olemassa käyttöliittymäolio nimeltään *lomake*, jolta syötetiedot kysellään.

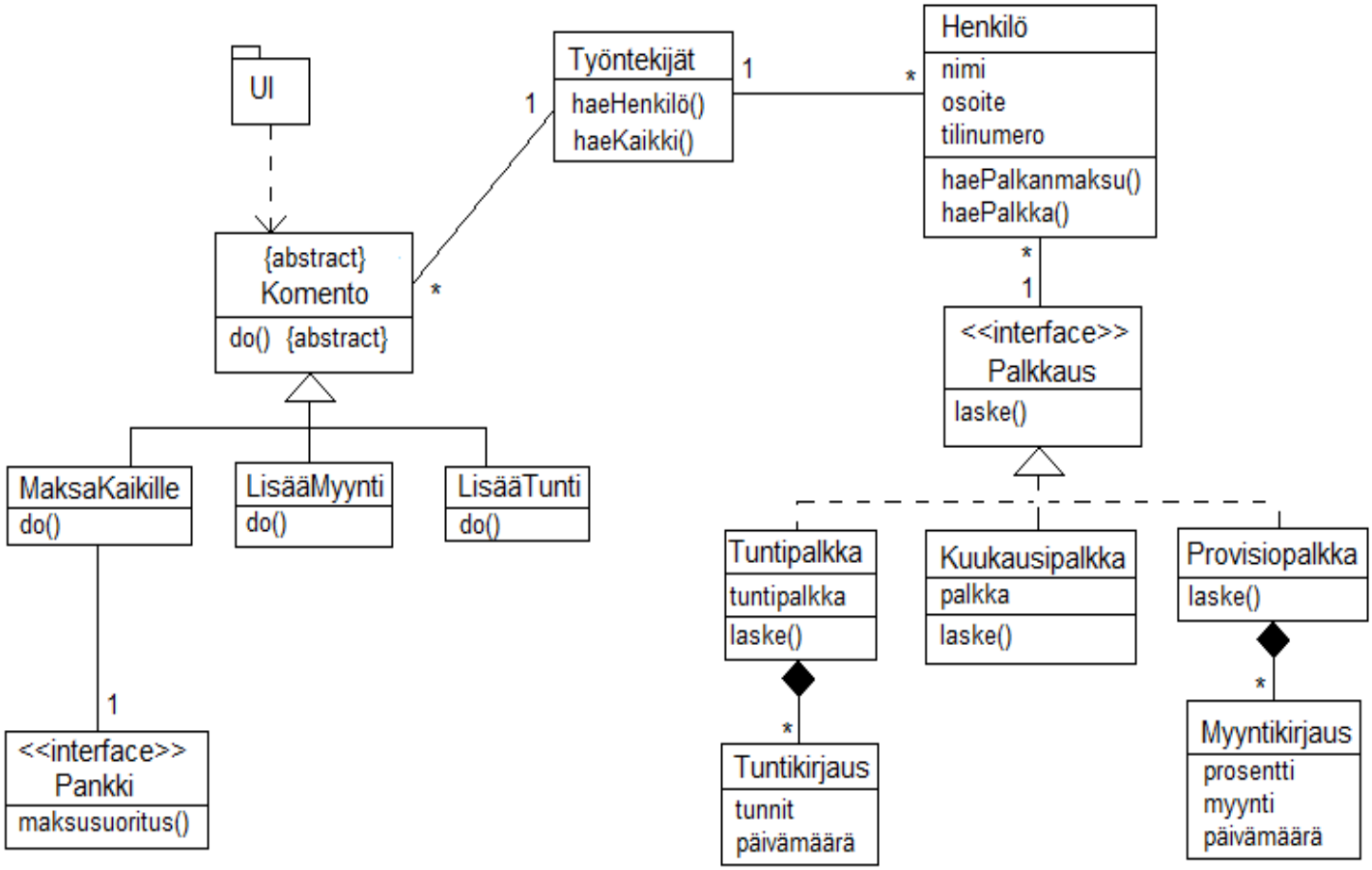
```
String nimi    =  lomake.getNimi();
int tunnit     =  lomake.getTunnit();
Date päiväys   =  lomake.getPaiva();

Komento k = new LisääTunti(nimi, tunnit, päiväys);
k.do();
```

Lisätään työntekijälle myyntisumma ja siihen liittyvä provisioprosentti:

```
String nimi     =  lomake.getNimi();
double myynti=  lomake.getMyynti();
double pros     =  lomake.getProsentti();
Date päiväys   =  lomake.getPaiva();

Komento k = new LisääTunti(nimi, myynti, pros, päiväys);
k.do();
```

Huomaamme, että käyttöliittymän kannalta asia on todella yksinkertainen! Tässäkin olisi vielä mahdollisuus yksinkertaistamiseen, jos käytettäisiin ns. tehdasperiaatetta (engl. factory pattern). Tällöin käyttöliittymän ei edes tarvitsisi tietää Komennon aliluokista mitään, se ainoastaan pyytäisi oliotehtaalta sopivia Komento-oliota.

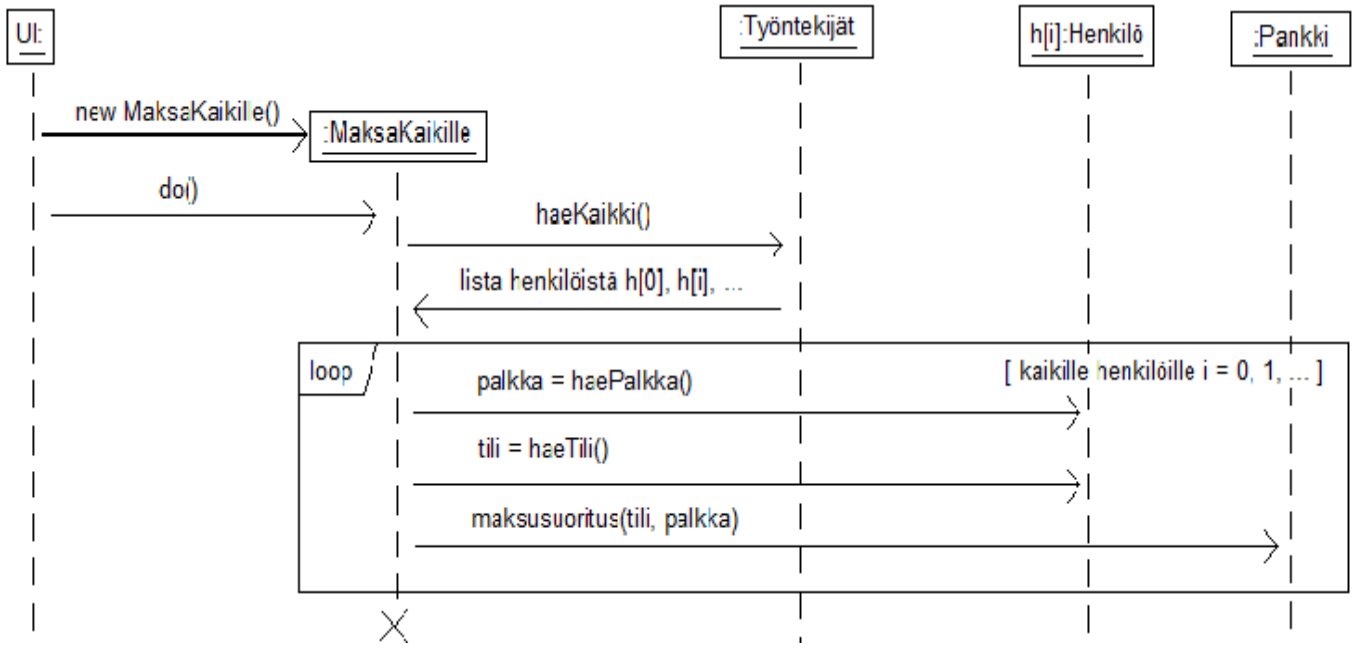
Seuraavassa luokkakaavio, jossa koottuna kaikki edellinen:



Kaikki monimutkaisuus on siis eristynyt Komento-luokan aliluokkien sisälle. Suunnittelemme seuraavaksi mitä kukin erillinen komento tekee. Kaikki Komento-luokan aliluokkien oliot ovat siis jonkin käyttöliittymäolion luomia ja käyttöliittymäolio kutsuu Komentojen do()-metodia.

MaksaKaikille

Ensin pyydetään Työntekijät-luokalta lista työntekijöistä. Sen jälkeen kysytään jokaisen työntekijän palkkaa ja tilinumeroa ja maksetaan palkka pankin avulla. Sekvenssikaavio seuraavassa. Huomionarvoista on, että Henkilö-oliot eivät tiedä itse palkkaansa, vaan aivan kuten aiemmin jo mainittiin, kutsuttaessa Henkilön metodia haePalkka(), kysyy se palkkansa omalta Palkkaus-rajapinnan toteuttamalta olioltaan. Sekvenssikaaviossa tätä ei kuitenkaan ole näytetty.



Koodina:

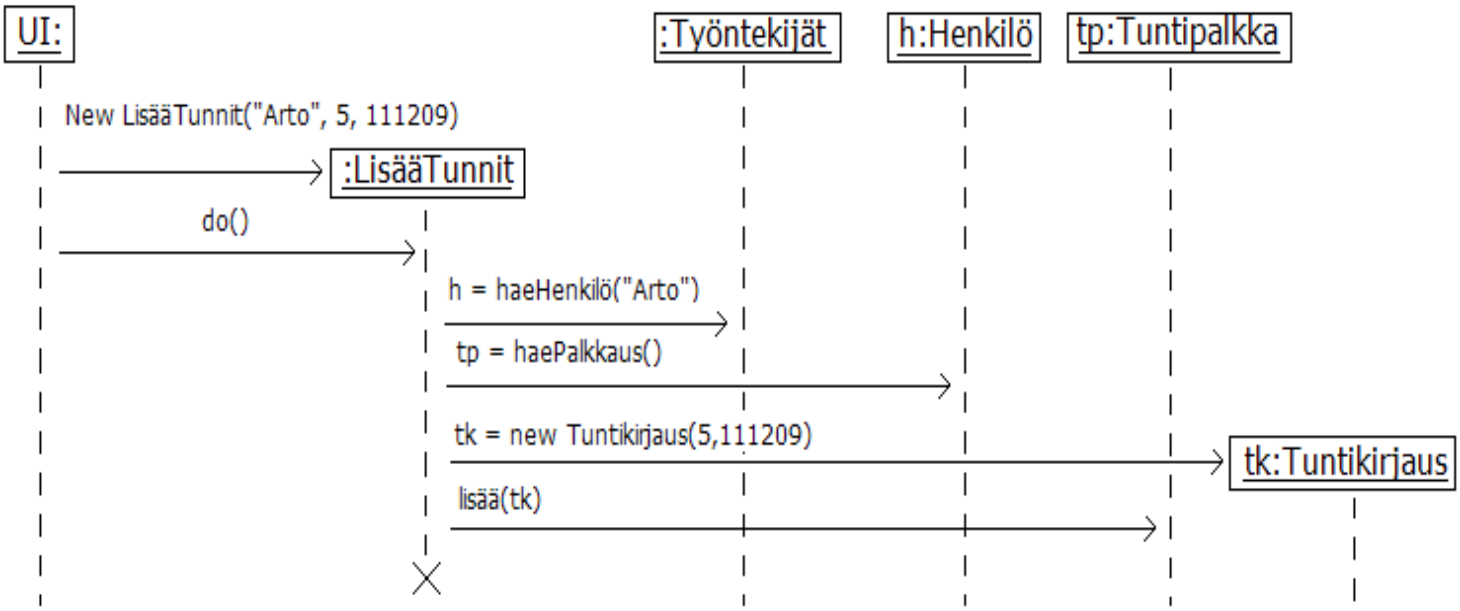
```
class MaksaKaikille extends Komento {
        void do(){
                ArrayList<Henkilö> henkilöt = tyontekijat.haeKaikki();
                for ( Henkilö h : henkilöt ){
                        double palkka = h.haePalkka();
                        int tilinro = h.haeTiliNro();
                        pankki.maksusuoritus( tilinro, palkka );
                }
        }
}
```

Huomionarvoista tässä ja seuraavissakin skenaarioissa on se, että *yksi Komento-olio on kertakäyttöinen*, eli käyttöliittymä luo komento-olion tarvittaessa, suorittaa se metodin do() ja sen jälkeen olioa ei enää käytetä uudelleen. Komennon seuraavaa suorituskertaa varten luodaan aina uusi olio.

LisääTunti

Ensin haetaan henkilö jolle tunnit lisätään. Kuten aiemmin päätimme, tuntikirjaus liittyy Henkilö-oliolla olevalle Palkkaus-rajapinnan toteuttavalle oliolle. Henkilölle on määritelty operaatio haePalkkas(). Operaatio palauttaa Henkilö-olioon liittyvän Palkkaus-rajapinnan toteuttavan olion.

Kun henkilö-olio on tiedossa, haetaan Henkilöön liittyvä Palkkaus-rajapinnan toteuttava olio, joka on tuntipalkkaa saavalla henkilöllä luokan Tuntipalkka olio. Luodaan uusi Tuntikirjaus-olio, joka lisätään Henkilön Palkkaus-rajapinnan toteuttavalle oliolle. Sekvenssikaaviona:



Koodina:

Luokalla Henkilö on siis operaatio haePalkkaus:

```
class Henkilö{
        Palkkaus palkkaus;
        Palkkaus haePalkkaus(){
                return palkkaus;
        }
        /* … */
}
```

Eli Henkilö-oliolta saadaan tarvittaessa viite siihen liittyvään Palkkaus-rajapinnan toteuttavaan olioon. Nyt lisätään tuntipalkatulle henkilölle työtunteja, ja jotta tunnit voidaan lisätä, on haePalkkaus()-metodin palauttama olio muunnettava todelliseen tyyppiinsä, eli TuntiPalkkaus-olioksi. Näin tapahtuu seuraavassa koodissa metodin do() toisella rivillä.

```
class LisääTunnit extends Komento {
        String nimi;
        int tunnit;
        Date päiväys;

        LisaaTunnit(Stiring n, int t, Date p){
                nimi = n;
                tunnit = t;
                päiväys = p;
        }

        void do(){
                Henkilö h = työntekijat.haeHenkilö(nimi);
                Tuntipalkka tp = (Tuntipalkka) h.haePalkkaus();
                Tuntikirjaus tk = new Tuntikirjaus(tunnit, päiväys);
                tp.lisää(tk);
        }
}
```

LisääMyynti

Luokan toteutus on lähes samanlainen kuin luokan LisääTunnit. Esitetään seuraavassa ainoastaan koodi. Koska nyt kyseessä provisiopalkattu henkilö, muunnetaan haePalkkaus()-metodin palauttama olio tyyppiin Myyntikirjaus.

```
class LisääMyynti extends Komento {
        String nimi;
        double myynti;
        double prosentti;
        Date päiväys;

        LisaaMyynti(Stiring n, double m, double pr, Date pv){
                nimi = n;
                myynti = m;
                prosentti = pr;
                päiväys = pv;
        }

        void do(){
                Henkilö h = työntekijat.haeHenkilö(nimi);
                Provisiopalkka pp = (Provisiopalkka) h.haePalkkaus();
                Myyntikirjaus mk = new Myyntikirjaus(myynti, prosentti, päiväys);
                pp.lisää(mk);
        }
}
```

Huomioita

Yksi tämän esimerkin motivaatioita oli näyttää, miten, ns. role-player-periaatteen mukaan usealla oliolla (Henkilö-olio ja siihen liittyvä Palkkaus-rajapinnan toteuttama olio) esitetyn työntekijän käsittely tapahtuu suunnittelutasolla.

Toiminnon MaksaKaikille yhteydessä siis kutsuttiin Henkilön metodia haePalkka(). Tämän suorituksen Henkilö delegoi omalle Palkkaus-rajapinnan toteuttamalle oliolleen.

Toimintojen LisääTunti ja LisääMyynti kohdalla tilanne on hiukan monimutkaisempi. Näissä toiminnoissa on lisättävä olio (Tuntikirjaus ja Myyntikirjaus) Palkkaus-rajapinnan toteuttaman olion alle. Asia on hoidettu siten, että Komento-olio pyytää Henkilöltä sen Palkkaus-rajapinnan toteuttaman olion (TuntiPalkkaus tai Provisiopalkkaus) ja kutsuu sitten tälle metodia, jonka avulla kirjaus lisätään olion alle.

Kumpi on suositeltavampi ohjausperiaate vai komento-olioiden käyttö? Riippuu tilanteesta. Komento-oliot voivat olla erittäin käytännöllisiä tilanteessa, jossa halutaan mahdollistaa myös komentojen undo()-operaatio. Tällöin voidaan ”muistaa” jo suoritetut komennot, ja tarvittaessa voidaan näille helposti suorittaa undo()-metodi. Esim. piirto-ohjelma kannattaisi toteuttaa tekemällä jokaisesta piirtotoimenpiteestä oma Komento-olio, jolle on do()-metodin lisäksi toteutettu myös undo() joka kumoaa piirtotoimenpiteen aikaansaamat vaikutukset. Pitämällä sitten kirjaa suoritetuista Komento-oliosta, voitaisiin melko helposti toteuttaa piirto-ohjelmalle undo-toiminnallisuus.

Komento-oliot sopivat yleensä ainoastaan rajallisen kokoisten operaatioiden suorittamiseen. Jos ajatellaan esim. lainaustapahtumaa kirjastosta, oitaisiin määritellä käyttötapaus, joka sisältää lainaajan tunnistamisen sekä useiden kirjojen lainaamisen. Tällaiselle käyttötapaukselle olisi suhteellisen helppo tehdä käyttötapauskohtainen ohjausolio. Komento-olion kaikki syötedata annetaan konstruktorin parametrina, joten jos dataa on paljon ja datan määrä ei ole ennalta tiedossa (useiden kirjojen lainaaminen) ei komento-olio sovi kunnolla tarkoitukseen.

On myös mahdollista käyttää molempia. Eli käytetään käyttötapauskohtaisia ohjausoliota, jotka sitten kutsuvat sopivia komento-oliota.