Simo Rantanen

Myöhäinen luokittaminen olio-ohjelmoinnissa

Tietojärjestelmätieteen
kanditaatintutkielma
13.05.2003

Jyväskylän yliopisto
Tietojenkäsittelytieteen laitos
Jyväskylä

TIIVISTELMÄ

SISÄLLYSLUETTELO

1 JOHDANTO

Perintä on nykyisten olio-ohjelmoinnin tärkein periaate. Perintä perustuu siihen, että osa tosimaailman olioista jakaa samoja ominaisuuksia ja käyttäytymismalleja. Perintä siis mahdollistaa ilmaista olioiden piirteiden jakamista deduktiivisesti; perityt oliot jakavat vanhenpiensa ominaisuudet. Vaikka perintä on tunnetusti hyvä ja tehokas ominaisuus, niin silti tutkijat kiistelevät perinnän tarkoituksesta ja käytöstä. Ongelmallista perinnässä on, että perinnän käyttötapoja on liikaa, Meyer (Meyer 1996) laskee niitä olevan jopa 12! Lisäksi ongelmia syntyy kun perintää käytetään yhä useampaan ja useampaan liiketoimintamalliin (application domain). Tällöin olioiden välille syntyy uudenlaisia riippuvuussuhteita (Swen 2000). Riehlen (Riehle 1998) mukaan tämä näkyy varsinkin ohjelmistokehyksissä.

Toisaalta osa tutkijoista painottavat lähdekoodin uudelleenkäyttön merkitystä olio-ohjelmoinnissa. Myöhäinen luokittaminen on myös lähdekoodin uudelleenkäyttöä tehostava keino. Seuraava esimerkki lähdekoodin uudelleenkäytöstä on muotoiltu Granston ja Russon (Granston 1991) mukaan. Kuvitellaan, että on olemassa kaksi luokkakirjastoa, jotka tarjoavat näyttökomponentteja (olioita) X-ikkunointijärjestelmään. Toisen hierarkian juuriluokka on `OpenLookObject` ja toisen `MotifObject`. Oletetaan lisäksi, että molemmissa luokissa on virtuaaliset metodit `display()` ja `move()`. Olisiko nyt mahdollista koota lista ikkunoista, jossa on ikkunoita molemmista luokkakirjastoista yhtäaikaisesti? Kyllä voidaan, mutta ei ilman eksplisiittistä tyyppien erottelua (case-lauseita, siltaluokkia, jne.) tai huomattavaa uudelleenkoodausta. Jos kuitenkin kyseiset oliot voitaisiin luokittaa uudelleen jälkikäteen, olisi ratkaisu triviaali: tehdään uusi tyyppi `WindowObject`, joka sisältää metodit `display()` ja `move()`. Tämän jälkeen voitaisiin muodostaa `WindowObject` tyyppisiä viitteitä sekä `OpenLookObject` -olioihin ja `MotifObject` -olioihin.

Swenin (Swen 2000) ja Baumgartnerin (Baumgartner 1995) mukaan olio-ohjelmointikielet, jotka käyttävät vain perintää, voivat kärsiä seuraavista ongelmista:

1. Joitakin olioiden suhteita ei voida määritellä vain perinnällä.
2. Joskun on vaikeaa tai jopa mahdotonta esitellä uusia luokkia takautuvasti perintää käyttäen.
3. Abstraktien tyyppien hierarkiat ja perinnällä saadut luokkahierarkiat ovat joskus mahdotonta sovittaa toisiinsa.
4. Perinnällä on liian monta käyttötarkoitusta nykyisissä olio-ohjelmointikielissä (Taivalsaari 1995). Jokaisen ohjelmointi ohjelmointiparadigman mukaiset käsitteet pitäisi olla yksiselitteisiä, jotta niiden käyttö olisi selkeää.

Harvat tuotantokäytössä olevat olio-ohjelmointikielet tukevat olioiden myöhäistä luokittamista. Perinteisesti luokkapohjaisissa olio-ohjelmointikielissä on käytössä periaate 'luokka on tyyppi', mikä on riittämätön sekä mallintamisen että lähdekoodin uudelleenkäytön kannalta. Myöhäistä luokittamista voidaankin siis ajatella uutena tyyppiabstraktioiden luontikeinona.

Tutkielman rakenne on seuraava: luvussa 2 esitellään yleistämisen periaate olio-ohjelmointiin ja verrataan sitä periytymiseen. Luku siis kertoo yleistämisestä käsitteellisen mallintamisen kannalta. Luvussa 3 katsotaan vastaavasti myöhäisen luokittamisen toteutukseen liittyviä asioita, kuten eri ohjelmointikielen ominaisuuksien vaikutuksia yleistämiseen. Tutkielma on perustuu kirjallisuuteen ja pyrkii yhdistämään hajanaisia näkemyksiä kattavasti eri näkökulmista.

2 KÄSITTEELLINEN MALLINTAMINEN

2.1 Määritelmiä

Perintä on keino toteuttaa inkrementaalista ohjelmointikehitystä. Tässä jo olemassa oleviin kokonaisuuksiin lisätään jotain uutta ja saadaan aikaan uusi kokonaisuus. Perintä siis mahdollistaa *deduktiivisen* tai *erikoistavan* ohjelmoinnin, jossa suuntaus on yleisestä yksittäisiin. Formaalisti perintä voidaan ilmaista seuraavasti:

$$\mathbf{R} = \mathbf{P} \circ \Delta\mathbf{R}.$$

Tässä **R** on uusi luotu kokonaisuus, **P** on vanha kokonaisuus ja $\Delta\mathbf{R}$ on ne uudet ominaisuudet, jotka erottavat **R**:n **P**:stä. Lisäksi $\circ$ on operaatio, joka yhdistää uudet ominaisuudet ja vanhan kokonaisuuden.

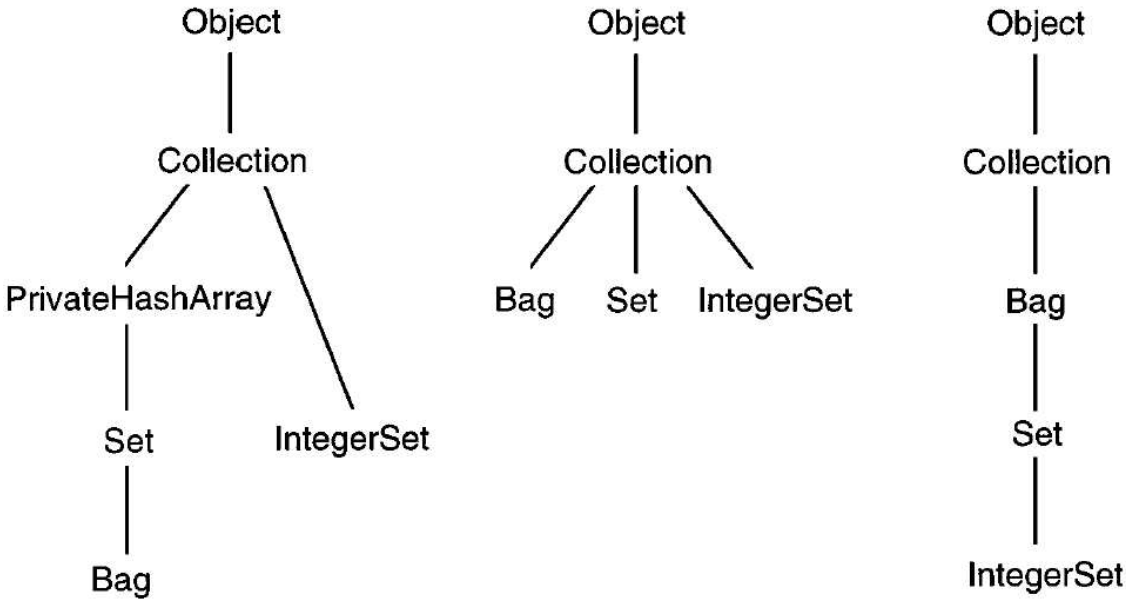
Luokittaminen on periaate, joka tukee periytymistä. Siinä kokonaisuudet ryhmitellään luokiksi. Luokka on pysyvä, muuttumaton kokonaisuus. *Instantointi* on vastaavasti periaate, jossa luokista tehdään eri olioita, jotka voivat vaihtua tiloiltaan ja attribuuteiltaan.

Aliluokituksella tarkoitetaan toteutuskeinoa, jolla toteutetaan koodin uudelleenkäyttöä ja esitellään luokat. Sen sijaan *alityypitys* on vaihdannaisuusominaisuus: alityyppiä voi käyttää vapaasti ylityyppinsä sijasta.

2.2 Periytyminen ja erikoistaminen

Kuten aiemmin mainittiin, periytymistä voidaan käyttää joko käsitteellistä mallintamista tukevana keinona tai koodin uudelleenkäyttöön tukevana keinona. Tämä kaksijakoisuus on yksi perinnän isoja ongelmia. Koskimies (Koskimies 2000) antaakin ohjenuoran käyttää pääsääntöisesti perintää käsitteelliseen mallintamiseen. Tämä siksi, että käsitteellisestä mallintamisesta seuraa (yleensä) koodin uudelleenkäyttöä, mutta toisinpäin sama ei päde.

Perintää siis pitäisi käyttää käsitteelliseen erikoistamiseen, kuitenkin tämäkään ei ole niin yleistä kuin olettaisi. KUVA 1 esittelee osaa SmallTalkin luokkahierarkiaa, jossa on käytetty perintää muuhunkin tarkoitukseen (koodin uudelleenkäyttö, optimointi) kuin erikoistamiseen. Taivalsaari (Taivalsaari 1995) mainitseekin, että tämän tyylistä käyttöä perinnälle on kritisoitu yleisesti. Monet olio-ohjelmointikielet sallivatkin perinnän yhteydessä keinoja, jotka ovat käsitteellisen erikoistamisen kanssa ristiriidassa.



Kuva 1 aliluokitus, alityypitys ja erikoistaminen.

Lisäksi perinnän ja erikoistamisen suhde on paljon monimutkaisempi kuin alunperin luultiin. Suurin osa ongelmista syntyy, koska olio-ohjelmointikielet eivät millään tapaa pakota perinnän käyttämistä erikoistamiseen. Tätä varten kehitettiin *yhteensopivuussääntöjä*. Esimerkiksi Wegner (Wegner 1990) käyttää neljän kategorian yhteensopivuustasoja:

1. *Poistaminen* (engl. cancellation) sallii aliluokille operaatioiden vapaan uudelleenmäärittelemisen ja vapaan poiston.
2. *Nimiyhteensopivuus* (engl. name compatibility) sallii aliluokissa operaatioden uudelleenmäärityksen, mutta aliluokan on pidettävä yliluokkansa operaatioiden nimet. Toisin sanoen tämä taso ei salli aliluokassa minkään yliluokan operaation poistoa.
3. *Rajapintayhteensopivuus* (engl. interface compatibility, signature compatibility) vaatii täyden syntaktisen yhteensopivuuden yliluokan ja aliluokan välille. Tässä tasossa siis aliluokan uudelleen määritellyt operaatiot täytyvät olla parametreiltään samalaiset kuin yliluokassa.
4. *Käytösyhteensopivuus* (engl. behaviour compatibility) olettaa täyden yhteensopivuuden yli- ja aliluokan käyttäytymisen suhteen. Tämä tarkoittaa, että aliluokka ei voi erota yliluokkansa käyttäytymisestä suuresti.

Koska ensimmäiset kolme yhteensopivuusmuotoa perustuvat vain syntaktisiin seikkoihin, on niiden tarkistaminen helppoa. Näistä kolmesta muodosta käytetäänkin yleensä nimitystä *heikko perintä* (engl. nonstrict inheritance). Tietysti käytösyhteensopivuutta on vaikea tarkistaa ja siitä käytetäänkin nimitystä *vahva perintä* (engl. strict inheritance). Esimerkiksi Eiffelissä on yritetty perintää vahvistaa erityisten metodien tulo- ja jättöehtojen avulla, joissa yliluokassa metodille annetaan kyseiset ehdot loogisina lauseina. Tietysti aina loogisten lausekkeiden muotoilu ei ole helppoa ja niihinkin voi tulla virheitä, joten tämäkään ei takaa täydellistä käytösyhteensopivuutta.

2.3 Yleistäminen

Vaikkakin käsitteellinen erikoistaminen on selvästi ohjelmistosuunnittelun kannalta tärkeä ja toimiva tapa, niin aina ei ole helppoa löytää eri luokkien yleisiä piirteitä ennen kuin luokat ovat jo toteutettu. Tämä pitää paikkaansa varsinkin ohjelmistokehyksissä, joissa monet oliot ovat monitoiminallisia. Esimerkiksi ikkunointijärjestelmässä ikkunoihin pitää pystyä piirtämään, toisaalta ikkunat ovat usein sisäkkäisiä (koostettavia) ja kolmanneksi ikkunoiden pitäisi reagoida syötelaitteisiin. Jos kaikkia ominaisuuksia ei huomaa jo suunnitteluvaiheessa, niin joudutaan luokkahierarkian yliluokkia muuttamaan jälkikäteen. Yliluokkien muuttaminen jälkikäteen on tunnetusti riskialtista vyörymäefektin takia: yliluokan muuttaminen aiheuttaa virheitä sen välittömissä aliluokissa, joiden korjaaminen aiheuttaa virheen näiden aliluokissa ja niin edelleen. Luonnollisesti yksittäisperiytyminen aiheuttaa mahdollisia lisäongelmia, sillä nyt ikkunat voidaan periä vain yhdestä ' pääominaisuudesta' , mutta pitäisikö se olla koottava, piirrettävä vai ohjattava ? Yksi tapa ratkaista ongelma olisi suunnitella jokainen ikkunatyyppi erikseen ja lopuksi kerätä niiden yhteiset ominaisuudet. Tätä kutsutaan *yleistämiseksi*.

Yleistäminen tai y*leistävä periytyminen* (engl. inheritance by generalization, upward inheritance) on keino, jossa uusien kokonaisuuksien luonti tapahtuu juuri toisinpäin kuin normaalissa perinnässä. Ideana on muodostaa uusia kokonaisuuksia ottamalla yhteisiä piirteitä jo olemassa olevista kokonaisuuksista. Täten yleistäminen mahdollistaa *induktiivisen ohjelmoinnin*, jossa suuntaus on yksittäisistä yleiseen.

Yleistäminen on siis, perinnän tapaan, myöskin käsitteellistä mallintamista tukeva keino. Etuna yleistämisessä on se, että se sallii *myöhäisen luokittamisen*, jossa uusia tyyppejä luodaan jälkikäteen. Nimensä mukaan myöhäinen luokittaminen vastaa normaalin perinnän luokittamisen periaatetta.

Vaikkakin yleistäminen tuntuu varsin luontevalta olio-ohjelmoinnissa, niin myöhäisen luokittamisen toteutustavoissa on vielä hiomista. Monia eri tyylisiä toteutuksia on esitetty sekä teoreettisille ohjelmointikielille että lisäyksinä nykyisille kielille. Toteutuksen helppouteen vaikuttaa myös se, kuinka paljon yleistämisen eri muotoja otetaan mukaan. Yleistämisen mahdollisista eri toteutuksista puhutaan lisää luvussa 3.

Edellisessä luvussa kerrottiin luokkien yhteensopivuustasoista. Yhteensopivuustasoilla ja yleistämisellä on selvät rajat; Luonnollisestikin yleistämisen täytyy toimia joko rajapintayhteensopivuuden tai käytösyhteensopivuuden tasolla. Pelkkä nimiyhteensopivuus ei riitä, sillä muuten itse toteutuksessa ei pakosti olisi esittelyä vastaavaa operaatiota. Lisäksi on huomattava, että myöhäinen luokittaminen mahdollistaa ' luonnollisella tavalla' operaatioiden poistamisen, jonka yleensä katsotaan olevan huono ominaisuus periytymisessä käsitteellisen mallintamisen kannalta (Pedersen 1989).

Yleistäminen ei ole olio-ohjelmoinnin kannalta uusi asia, mutta se ei selvästikään ole silti yleistynyt mihinkään tuotantokäytössä olevaan kieleen. Kuitenkin osa olio-ohjelmointikielistä tukee joitakin yleistämisen muotoja, mm. abstraktit luokat (C++) , rajapinnat (Java), signatures (GNU C++:n laajennos) ja metodi-osoittimet (Object Pascal).

2.4 Tyypitys

Normaali ohjelmistosuunnittelu tukee jo yleistämistä, sillä siinähän suunnittelijan tehtävä on kerätä eri kokonaisuuksien yhteisiä piirteitä ja sitten tehdä näistä yliluokkia. Tästä herääkin kysymys onko yleistäminen sitten

todella tarpeellista. Vastaus on kyllä, sillä aina ei perinnällä voida ilmaista ali- ja ylityyppien eroja.

Seuraava klassinen esimerkki perinnän heikkoudesta ovat esitelleet mm. Baumgartner ja kump. (Baumgartner 1995b), Pedersen (Pedersen 1989) ja Snyder (Snyder 1986). Olkoon meillä abstraktit tyypit `Jono` ja `KPJono` (kaksipäinen jono). Tyyppi `Jono` tarjoaa operaatiot alkioiden lisäykseen jonon loppuun ja alkioiden poistoon jonon alusta. Tyyppi `KPJono` tarjoaa samat operaatiot kuin `Jono` ja lisäksi operaatiot alkioiden poistoon jonon lopusta sekä alkioiden lisäykseen jonon alkuun. Täten `KPJono` on tyypin `Jono` alityyppi.

Kuitenkin helpoin tapa toteuttaa `Jono` ja `KPJono` on periyttää luokat juuri päinvastoin tyypityksen kanssa! Luonnollinen valinta kaksipäisen jonon toteuttamiseen olisi kaksoislinkitetty lista. Tällöin taas jonon toteutus olisi helppo: peritään lista kaksoislinkitetystä listasta ja poistetaan, tai piilotetaan ylimääräiset operaatiot.

Cook ja kump. (Cook 1990) huomauttaakin, että ei ole mahdollista määritellä alityyppisuhdetta perinnällä ja samalla sallia aliluokan piilottaa yliluokkansa operaatioita, jotta ohjelmointikielen tyypitysjärjestelmä olisi pätevä. Täten ratkaisua ei voi tehdä siististi perintää käyttäen. Myöhäisellä luokittamisella tehty ratkaisu ongelmaan on liitteessä 1.

Samalaisia esimerkkejä alityyppien ja perinnän ristiriidoista on monia, esimerkiksi Koskimies (Koskimies 2000) esittää saman ongelman ikkunan ja värillisen ikkunan välillä.

2.5 Formalisointi

Olkoot **O**, **P** ja **Q** luokkia. Nyt yleistäminen voidaan formalisoida asettamalla operaattori ∇ :

$$R \nabla (O , P , Q) = \forall c : \frac{O.c \wedge P.c \wedge Q.}{R.c} ,$$

missä predikaatti ' .on totta luokalle **O** ja ominaisuudelle **c**, jos ja vain jos **O**:lla on ominaisuus **c**. (Swen 2000) (Pedersen 1989)

Swen (Swen 2000) näppärästi huomauttaakin, että näin formalisoituna ∇ *on relaatio ja siten seuraavat ominaisuudet pätevät:*

1. $O \nabla O$.
2. $O \nabla P \wedge P \nabla O \rightarrow O = Q$.
3. $O \nabla P \wedge P \nabla Q \rightarrow O \nabla Q$.

Tästä seuraa, että yleistäminen muodostaa luokille osittaisen järjestyksen (vrt. luokkahierarkiat). Lisäksi on helppo huomata, että mikä tahansa luokka voidaan indusoida ' tyhjäksiluokaksi tai itsekseen. Toisaalta voidaan huomata, että *yksittäisyleistäminen* (vrt. yksittäisperiytyminen) on hyödyllistä vain monimuotoisuuden toteutukseen luokkien välille tai operaatioiden poistamiseen. (Swen 2000)

3 MYÖHÄINEN LUOKITTAMINEN

3.1 Eksplisiittinen ja implisiittnen yleistäminen

Myöhäinen luokittaminen on siis luokittamista vastaava periaate, jossa ryhmitetään yhteen, kategorioidaan, yhteisiä piirteitä. Erona normaaliin luokittamiseen oli se, että vanhaan luokkaan lisäämisen sijasta nyt kerätään jo olemassa olevista luokista yhteisiä piirteitä. Myöhäinen luokittaminen voidaankin jakaa sen mukaan kerätäänkö yhteiset ominaisuudet automaattisesti vai pitääkö ne kirjata esittelyyn eksplisiittisesti. Jos yleistäminen tehdään automaattisesti, niin sitä kutsutaan *implisiittiseksi yleistämiseksi*, ja vastaavasti *eksplisiittisessä yleistämisessä* yleiset piirteet pitää kirjata itse. Edellä annetun formalisoinnin mukaanhan automaattinen yhteisten ominaisuuksien kerääminen on mahdollista.

ESIMERKKI 1 havainnoi ekplisiittistä ja implisiittistä yleistämistä. Selvästi molemilla on omat hyvät puolensa. Eksplisiittisessä yleistämisessä näkyy luodun tyypin rajapinta selvästi, joten siihen kuuluvat metodikutsut ovat helppo tarkastaa. Baumgartnerin (Baumgartner 1997) mallissa on toteutettu myöhäinen luokittaminen C++:lle juuri näin. Vastaavasti implisiittisessä yleistämisessä on se hyvä puoli, että yleistettävien luokkien muutokset tulevat automaatisesti myös myöhäisesti luokitettuun tyyppiin. Sekä Pedersen (Pedersen 1989) että Swen (Swen 2000) ehdottivat artikkeleissaan tällaista tyyliä, mutta Schefl ja Neuhold (Scherfl 1988) kritisoivat implisiittistä yleistämistä nimikonfliktien takia, joista lisää myöhemmin. Lisäksi Baumgartner (Baumgartner 1995b) on toteuttanyt ns. `sigof`-rakenteen, jolla saadaan kerättyä luokasta automaattisesti kaikki rajapintarakenteet.

```
LateClass SimonKandi
{
// käyttäjä listaa itse
// haluamiensa luokkien
// yhteiset piirteet
      void kappale1();
      void kappale2();
}
```

```
AutoLateClass AutoKandi
      : Lähde1
      , Lähde2
      , ...
{
// sisältää Lähde1:n,
// Lähde2:n, jne. yhteiset
// piirteet
}
```

ESIMERKKI 1. Eksplisiittinen ja implisiittinen yleistäminen

3.2 Tarkentaminen ja nimikonfliktit

Yksi mielenkiintoinen asia myöhäisessä luokittamisessa on se, että sallitaanko myöhäisesti luokitellussa kerätä myös eri luokkien yhteisiä attribuutteja ja sallitaanko edelleen uusien metodien luonti tai vanhojen tarkentaminen. Vaikka käsitteellisen mallintamisen kannalta olisi luonnollista, että eri luokista voitaisiin kerätä yhteiset piirteet ja sen jälkeen tarkentaa tarvittasti uutta luokkaa, niin tämä kärsii monista ongelmista.

Selvästi tarkentaminen kulkee käsi kädessä implisiittisen ja eksplisiittisen yleistämisen kanssa. Implisiittisessä yleistämisessä tuntuisi kummalta, jos muuten automaattista toimintoa täytyisi myöhemmin paikata tarkennuksilla. Täten tarkentaminen sopiikin paremmin ekplisiittiseen yleistämiseen. (Scherfl 1988)

Oman ongelman tarkentamiseen tuo *nimikonfliktit*, jotka toimivat juuri toisinpäin kuin moniperinnässä: yleistämisessä nimikonflikti syntyy jos yleistettävissä luokissa on erinimiset metodit, jotka toimivat samalla tavalla, tai erinimiset attribuutit, jotka ilmoittavat samaa asiaa. Vaikkakin nimikonfliktit tuntuvat erilaisilta perinnässä ja yleistämisessä, niin Scherfl ja Neuhold (Scherfl

1988) mainitsevat, että ongelma on oleinnaisilta osin sama ja täten se on ratkaistavissa täysin samalla tavalla kuin moniperinnässä. Tietysti nimikonflikteja ei tule, jos kieli sallii vain yksittäisyleistämisen.

Nimikonfliktien ratkaiseminen voidaan tehdä lähinnä kahdessa paikassa, joko luokan esittelyssä tai sitten sijoitusvaiheessa. *Aikaisessa aliasoinnissa* eri luokkien vastaavat nimet ilmaistaan jo luokan esittelyssä. Vastaavasti *myöhäisessä aliasoinnissa* vastaavat nimet ilmaistaan vasta, kun myöhäisesti luokkiteltuun instanssiin sijoitetaan oliota. ESIMERKKI 2 havannoi aikaista ja myöhäistä sitomista.

```
LateClass Kandi
      : Lähde1, Lähde2,
      ...
{
      void kappale1()
            : Lähde1::johdanto,
            : Lähde2::yhteenveto;
      void kappale2();
};
...
Lähde1 olio = new Lähde1();
Kandi& kandi = olio;
// kutsuu Lähde1::johdanto
kandi.kappale1();
```

```
LateClass Kandi
{
      void kappale1();
      void kappale2();
};
...
Lähde1 olio = new Lähde1();
Kandi& kandi = olio {
      kappale1 : johdanto
};
// kutsuu Lähde1::johdanto
kandi.kappale1();
```

ESIMERKKI 2. Aikainen ja myöhäinen aliasointi.

Aikainen aliasointi sopii paremmin implisiittiseen yleistämiseen, sillä tällöin voidaan vain tarkentaa mitkä ominaisuudet ovat aliaksia toisilleen. Aikainen sitominen muistuttaakin vähän Eiffelin perittyjen ominaisuuksien uudelleennimeämistä (Meyer 1988). Vastaavasti myöhäinen sitominen sopii paremmin eksplisiittiseen yleistämiseen, sillä tällöin esiteltyyn ' rajapintaan'

voidaan sijoittaa mikä tahansa olio, joka toteuttaa kyseisen rajapinnan.

Kuten moniperinnässä, niin myös yleistämisessä, on mahdollista ratkaista nimikonfliktit yksinkertaisesti siten, että kielletään nimikonfliktit kokonaan. Tämähän selvästi sopii eksplisiittiseen yleistämisen ' rajapinnoille' mutta rajoittaa kyllä myöhäisen luokittamisen hyötyä, sillä samalle asialle on usein monta synonyymia. Esimerkiksi johdannon mukaisessa ikkunointijärjestelmässä voisi toisessa luokkahierarkiassa olla metodi `show()` ja toisessa `display()`.

3.3 Yleistämisen tasot

Eksplisiittisen ja implisiittsen yleistämisen lisäksi voidaan myöhäinen luokittaminen jakaa sen mukaan, kuinka myöhäisesti luokitellut tyypit suhtautuvat normaaleihin, luokilla tehtyihin[1], tyyppeihin. *Viitetasolla* myöhäisesti luokitetut tyypit toimivat siten, että niiden instanssit toimivat viitteinä joihinkin luokasta tehtyyn olioon, eli tällä tasolla myöhäisesti luokitetuista tyypeistä ei voi luoda uusia olioita. *Oliotasolla* myöhäisesti luokitetut tyypit toimivat täysivaltaisina tyyppeinä ja niitä voidaan käyttää kuten normaalien luokkien instansseja. ESIMERKKI 3 havannoi eri tasojen eroja.

1 Luokkien lisäksi luonnollisesti voidaan olla myös atomisia tyyppejä, kuten int C++:ssa.

```
Interface Kulkuneuvo {                  LateClass Kulkuneuvo
      ...                                     : Rekka
      void käynnistä();                       : Vene
      void sammuta();                   {
      ...                                     ...
};                                            void käynnistä() from Vene;
...                                           void sammuta() from Vene;
// onnistuu, jos Auto:ssa on                  ...
// vastaavat metodit                  };
Auto* auto = new Auto();                ...
Kulkuneuvo* kn = auto;                  // Kulkuneuvo on täysipainoinen
// seuraava tuottaa virheen             // luokka
Kulkuneuvo* kv = new Kulkuneuvo(); Kulkuneuvo* kv = new Kulkuneuvo();
```

ESIMERKKI 3. Viitetason ja oliotason yleistäminen

Viitetasolla myöhäisesti luokitellut tyypit muistuttavat C++:n abstrakteja luokkia tai Javan rajapintoja. Ainoana erona näillä on se, että myöhäisesti luokitellun tyypin instanssiin voidaan sijoitaa minkä tahansa luokan olio, vaikka luokka ei ole ' luvannutkaantoteuttanutkaan kyseistä rajapintaa. Vaikka ero ei tunnu aluksi suurelta, niin Baumgartner (Baumgartner 1995) huomattaakin, että jo tällä saavutetaan kaksi huomattavaa eroa:

1. Alityypitys erotetaan periytymisestä. Tyyppi T on myöhäisesti luokitellun tyypin M alityyppi riippumatta luokkahierarkiasta; riittää että T toteuttaa M:ssä annetut ominaisuudet.
2. Myöhäisesti luokitellut tyypit toimivat jo olemassa olevien luokkahierarkioiden kanssa. Abstraktit luokat eivät auta, jos käytetään lähdekoodia joka on olemassa vain käännetyssä muodossa, sillä tällöin jouduttaisiin käyttämään siltaluokkia tai case-lauseita.

Viitetasolla ollaan siis kiinnostuneita rajapinnoista, joten eksplisiittinen yleistäminen tuntuisi sopivan paremmin tälle. Esimerkiksi Baumgartnerin (Baumgartner 1997) Signatures for C++ toimii näin, ja kyseisssä C++:n

laajennoksessa rajoitetaan myöhäisesti luokitellyt tyypit olemaan vain joko osoittimia tai viitteitä. Viitetason suhde tarkentamiseen on myös selvä, sillä tarkentamista ei voida tehdä, sillä kysehän on rajapinnoista. Tosin Baumgartner (Baumgartner 1995) huomattaa, että sekä uusien metodien, tai lähinnä metodirunkojen, määritteleminen että oletustoteutuksien määritteleminen on mahdollista. Viitetason tyyppejä kutsutaan rajapinnoiksi (engl. interface, signature).

Siinä missä viitetasolla toimiminen rajoittuu selvästi rajapintoihin, niin oliotasolla asiat vaikeutuvat huomattavasti. Koska nyt myöhäiset luokat ovat täysivaltaisia luokkia, täytyy kerätä yleistettävistä luokista toteutusmetodit tai ylikirjoittaa metodit itse. Pedersen (Pedersen 1989) huomauttaakin, että tällöin on pakko tietää yleistettävien luokkien toteutus lähdekoodimuodossa, jotta tarvittavat attribuutit voidaan kerätä yleistettyyn luokkaan. Lisäksi Pedersen suosittelee, että metodien ylikirjoittaminen olisi mahdollista yleistettävässä luokassa, koska pakosti mikään yleistettävistä luokista ei tarjoa sopivaa yleistä toteutusta. Lisäongelmia aiheuttaa vielä attribuuttien tai metodien semanttiset suhteet yleistettävissä luokissa. Schrefl ja Neuhold (Scherfl 1988) jakavatkin nämä suhteet neljään katogoriaan:

1. *rooli-suhteessa* luokat esittävät samaa tosimaailman kokonaisuutta eri tilanteessa tai kontekstissa. Esimerkiksi ' yliopistotyöntekijä Meikeläinen' ja ' Työntekijä Meikeläinen' mallintavat samaa henkilöä eri rooleissa.
2. *Historia-suhteessa* luokat esittävät samaa kokonaisuutta, mutta eri aikoina. Esimerkiksi ' Hakija' ja ' Työntekijä' ovat historia-suhteessa.
3. *Vastine-suhteessa* luokat jakavat joitakin ominaisuuksia ja mallintavat tosimaailman vaihtoehtoisia tilanteita. Esimerkiksi ' Juna-yhteys' ja ' Lentokone-yhteys' voisivat olla vastine-suhteessa, sillä molemmilla on lähtö- sekä tulokaupunki ja aikataulu yhteisinä ominaisuuksina. Junayhteys ja Lentokoneyhteys ovat vastike-suhteessa, sillä molempien instansseista voi löytyä lentoyhteys, joka kulkee Helsingistä Tampereelle, ja junayhteys, joka

kulkee Helsingistä Tampereelle.

4. *Kategoria-suhteessa* olevat luokat myös jakavat joitakin yhteisiä ominaisuuksia ja mallintavat tosimaailman vaihtoehtoisia tilanteita. Esimerkiksi ' öljyvoimalaitos' ja ' hiilivoimalaitos' ovat kategoria-suhteessa. Se miksi kyseiset voimalaitokset eivät ole vastine-suhteessa on, että molempien instansseista ei löydy ' vastaavia' olioita.

3.4 Tyypitys ja rajapintojen periytyminen

Yksi asia mihin myöhäinen luokittaminen vaikuttaa suuresti on (staattisesti tyypitettyjen) olio-ohjelmointikielten tyypitys, sillä nythän alityypitys erotetaan aliluokituksesta. *Tyypintarkastussäännöt* ovat ohjelmointikielen säännöt, joilla tarkastetaan onko tyyppi T *tyyppiyhteensopiva* tyypin S kanssa. Jos tyyppi T on tyyppiyhteensopiva tyypin S kanssa, niin tyypin T instansseja voidaan käyttää kuten ne olisivat tyypin S instansseja. Pitämällä alityypityksen ja aliluokkien välisen suhteen yksinomaisena saavutetaan se etu, että ohjelmointikielen tyypintarkastus yksinkertaisutuu. Tällöin tyyppi T on tyyppiyhteensopiva tyypin S kanssa, jos T on S:n aliluokka. Tällöin kääntäjän on helppoa ja nopeata tarkistaa tyypitys. (Pedersen 1989)

Yleistäminen monimutkaistaa tyypintarkastussääntöjä, mutta jos myöhäinen luokittaminen rajoitetaan viitetasolle saadaan tyypintarkastussäännöt pysymään suhteellisen yksinkertaisina. Tällöin täytyy vain tarkistaa sijoituksissa onko sijoitettavan olion luokassa rajapinnassa määritellyt ominaisuudet. Tätä voidaan tietysti nopeuttaa välimuistilla, kuten Baumgartnerin (Baumgartner 1995b) C++:n laajennoksessa.

Yksi mielenkiintoinen osa viitetason yleistämisessä tyypintarkastuksen kannalta on se, että kahden rajapinnan tyyppiyhteesopivuutta ei voida päätellä

rajapintojen perintäjärjestyksellä. Rajapinnathan ovat samoja, jos ja vain jos niissä on samat metodit. Tämä tarkoittaa, että kaksi erinimistä rajapintaa voivat määritellä saman tyypin ja täten olla tyyppiyhteensopivia. Lisäksi rajapinta voi olla toisen alityyppi, vaikka niillä ei ole mitään perintäsuhdetta.

Yksi tyyppijärjestelmän kannalta hankalahko ominaisuus on *rekursiivit tyypit*. Rekursiivinen tyyppi on tyyppi, joka viittaa itseensä. Yleinen rekursiivinen tyyppi on esimerkiksi listan alkio, joissa on osoitin seuraavaan listaan alkioon (samaan tyyppiin). Luonnollisesti myös yleistetyissä luokissa voi olla rekursiivia tyyppejä, ja niiden perintä poikkeaakin luokkien perinnästä. Rajapintojen perintä pitää olla lähinnä tekstipohjaista korvaamista, eli kopioidaan ylityypissä määrätyt metodit alityyppiin siten, että jokainen ylityypin rekursiivinen viittaus muutetaan viittaamaan alityyppiin.

Seuraava esimerkki tästä on muotoiltu Baumgartnerin ja Russon (Baumgartner 1995b) mukaan. Tässä esimerkissä käytetään toista yleistä mallia, missä perinnän kanssa joudutaan hankaluuksiin: algebraa. Olkoot `Ryhmä` algebrallista ryhmää esittävä abstrakti tyyppi ja `Rengas` vastaavasti algebrallista rengasta esittävä abstrakti tyyppi sekä lisäksi `Matriisi` luokka, joka on tyyppiyhteensopiva kummankin rajapinnan kanssa. ESIMERKKI 4 havannoi tätä. Jos esimerkissä rajapintojen perintä olisi tehty kuten luokkien perintä, niin `tulo`:n kutsu ei läpäisisi tyypintarkistusta.

```
interface Ryhmä {
      Ryhmä* summa(Ryhmä *);
      // ...
};

interface Rengas : Ryhmä {
      Rengas* tulo(Rengas *);
      //...
```

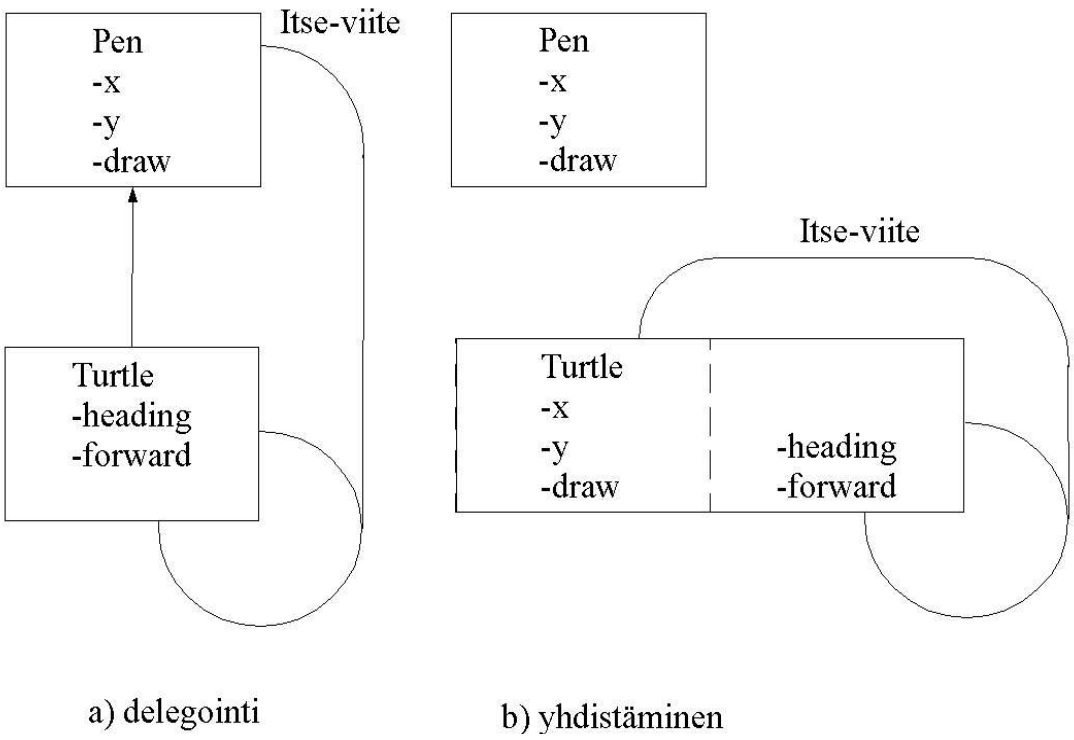
```
};

Rengas* p = new Matriisi();
Rengas* q = p->tulo(p->summa(p)); // p * (p + p)
```

ESIMERKKI 4. Rekursiiviset tyypit ja niiden perintä.

3.5 Virtuaalitaulut ja delegointi

Olio-ohjelmoinnissa metodien suorittamiseen (engl. message dispatching) on kaksi päätoteutusta, virtuaalitaulut ja delegointi. Virtuaalitauluissa jokaisella luokalla on virtuaalitaulu, joka sisältää osoittimia olion metodeihin. Virtuaalitaulu siis sisältää myös yliluokkien metodit ja tätä kutsutaankin *yhdistämiseksi* (engl. concatenation). Toinen tapa järjestää metodien lähetys on käyttää isäntäviitettä olion aliolioon (yliluokkaan). Jokainen metodikutsu, jota nykyinen olio ' ei ymmärrä' siirretään eteenpäin alioliolle. Tätä kutsutaan delegoinniksi. KUVA 2 havannoi delegoinnin ja yhdistämisen eroja.

Yleistäminen voidaan toteuttaa riippumatta siitä, kumpaa systeemiä metodien suorittamiseen käytetään. Kuitenkin metodien suorittaminen vaikuttaa yleistämiseen, sillä yleensä delegoivissa järjestelmissä halutaan kaikki ymmärtämättömät viestit lähettää eteenpäin ja virtuaalitauluissa tulee ongelmaksi taulun luonti ja käsittely. Seuraavissa esimerkeissä käytetään Baumgartnerin ja Russon (Baumgartner 1997) sekä Schreflen ja Neuholdin (Scherfl 1988) malleja. Tässä esitellään toteutukset varsin yleisellä tasolla ja tarkemmat yksityiskohdat kannattaa katsoa kyseisistä artikkeleista.

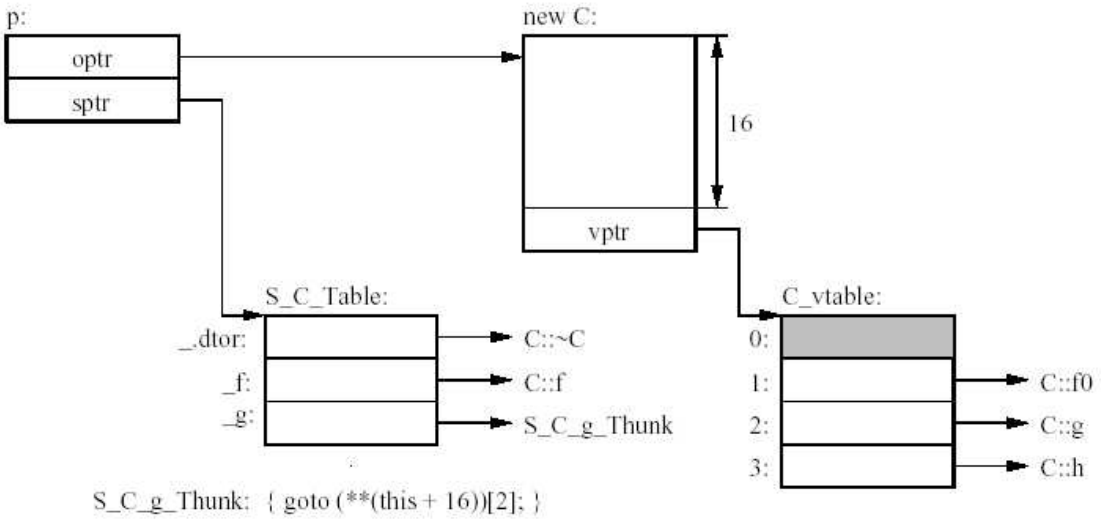


Kuva 2Delegointi ja yhdistäminen

Ohjelmointikielissä, joissa on käytössä virtuaalitaulut täytyy myöhäisesti luokitetuille tyypeille muodostaa kyseessä oleva virtuaalitaulu. Jos kyseessä on oliotason yleistäminen, niin ongelma ratkeaa samaan tapaan kuin perinnässä, mutta joitakin muutoksia on tehtävä esimerkiksi aliasoinnin takia. Sen sijaan mielenkiintoista virtuaalitauluissa on viitetason yleistäminen, sillä nyt viitteeseen on liitettävä oma virtuaalitaulunsa. Viite siis koostuu parista <olio-viite,virtuaalitaulu-viite>. KUVA 3 kuvaa tällaista paria, siinä on käytetty kääntäjän edustaa myöhäsen luokittamisen toteutukseen. Myöhäisesti sidottujen metodien osalta joudutaan tekemään ns. tynkämetodi, joka kutsuu itse olion luokan virtuaalitaulun kautta myöhäisesti sidottua metodia (KUVA3, S_C_g_Thunk). Täten sekä aikaisesti että myöhäisesti sidottujen metodien kutsumiseen tarvitaan yksi ylimääräinen viite, joten kutsuminen vie n. 2 kertaa sen ajan mitä normaalista. Tosin jos toteutus tehtäisiin kääntäjän tausta-osalla, niin tulokset olisivat parempia. Tietysti yksi tehokkuuteen vaikuttava asia on

se, että kopioidessa rajapintaa toiseen joudutaan kopioimaan myös virtuaalitaulu pelkän osoittimen sijaan. Samoin KUVA 3 näyttää selvästi, miksi on luonnollista kieltää uusien muuttujien lisääminen rajapintaan; muuttujat pitäisi sijoittaa joko viiteparin jälkeen tai virtuaalitauluun, jolloin pakosti kääntäjän toteutus monimutkistuisi.

Vastaavast delegoivissa ohjelmointikielissä ongelma on yksinkertaisempi. Näissä kielissä riittää muodostaa delegoiva luokka, joka lähinnä hoitaa aliasoinnin. Esimerkiksi Schreflin ja Neuholdin mallissa, joka perustuu SmallTalkiin, saavutetaan yleistäminen perimällä juuriluokasta `Object` erityinen luokka, jota he kutsuvat metaluokaksi. Tässä luokassa on viite yleistettyyn luokkaan, joten on mahdollista edelleen delegoida viestit viitteenä olevaan luokkaan.



Kuva 3Virtuaalitaulut (Baumgartner 1997, s. 175)

3.6 Metodi-osoittimet

Yhtenä yleistämisen erikoistapauksena voidaan pitää sellaista rajapintaa, jossa on määritelty vain yksi metodi. Tällaiset rajapinnat on huomattu käteväksi

esimerkiksi GUI-kirjastoissa. Usein näitä kutsutaan joko *signaloinniksi* (QT 2003) (Pulkkinen T. ym. 2003) tai *metodiosoittimeksi*. Metodiosoittimen nimi on luonnollinen, sillä nyt rajapinta koostuu vain olion viitteestä ja yhden metodin viittestä, jolloin on tarpeeksi tietoa suorittaa kyseinen metodi. Lisäksi metodi-osoittimen voi suorittaa ilman virtuaalitauluja, sillä nyt riittää tallentaa pelkästään suoritettavan metodin osoite. Yleensä metodiosoittimia käytetään eri tapahtumien käsittelyyn, esimerkiksi `buttonClick`-tapahtumaan. Ainakin Object-Pascalissa tuetaan metodi-osoittimia ja Delphi/Kylix käyttääkin niitä paljon graafisissa luokissa.

4 YHTEENVETO

Olio-ohjelmointi on nykyajan ohjelmoinparadigmoista käytetyin, koska sillä pystytään mallintamaan tosimaailman olioita, se tarjoaa hyvän abstraktiotason yksinkertaistaen ongelmia ja se johtaa koodin uudelleenkäyttöön. Olio-ohjelmoinnin keskeisimpiä käsitteitä on perintä, joka kuitenkin kärsii muutamista ongelmista. Näistä osa aiheutuu perinnän ja tyypityksen yhdessäpidosta.

Lisäämällä olio-ohjelmointiin yleistämisen periaatteen, voidaan olio-ohjelmointia tehostaa. Tällöin erotetaan alityypitys aliluokituksesta, joka parantaa varsinkin jo olemassa olevien luokkakirjastojen yhteensopivuutta ja auttaa mallintamaan niitä tosimaailman kohteita, joissa aliluokitus eroaa alityypityksestä. Lisäksi yleistäminen mahdollistaa yliluokan operaatioiden poiston, mikä muuten ei voi olla mahdollista, jos ohjelmontikielen tyypitysjärjestelmä on eheä.

Yleistämistä tukeva myöhäisen luokittamisen periaate voidaan lisätä mihin tahansa olio-ohjelmointikieleen. Lisäksi myöhäisen luokituksen voi vielä tehdä yksinkertaisesti ja tehokkaasti.

Tutkielmassa on esitelty myöhäiseen luokittamiseen liittyviä periaatteita, joista selvästi käytännölliseen olio-ohjelmointikieleen ei kannata valita kaikkia ' pelejä ja rensseleitä' ,vaan pysyä niissä osissa, joita ei voida normaalilla perinnällä toteuttaa.

Vaikka myöhäinen luokitus ei ole mikään ' hopealuoti'nii,n silti se suurentaa nykyistä arsenaalia huomattavasti.

VIITTEET

Baumgartner G., 1995. Type abstaction using signatures. Using and Porting GNU CC, Stallman. R.M, Free Software Foundation

Baumgartner G. ja Russo V.F., 1995. Signatures: A Language Extension for Improving Type Abstraction and Subtype Polymorphism in C++. Technical Report CSD-TR-95-051

Baumgartner R., Russo V.F., 1997. Implementing Signatures for C++. ACM Transactions on Programming Languages and Systens

Cook W.R., Hill W.L., Canning P.S., 1990. Inheritance is not subtyping.. Proceedings of the 17th Annual ACM Sympsosium of Programming Languages

Granston E.D. ja Russo V.F., 1991. Signature-based polymorphism for C++. Proceedings of the 1991 USENIX C++ Conference

Koskimies K., 2000. Oliokirja.

Meyer B., 1988. Object-oriented Software Construction. Prentice-Hall.

Meyer B., 1996. The many faces of inheritance: a taxonomy of taxonomy . IEEE Computer

Pedersen H.C., 1989. Extending Oridinary Inheritance Shecmes to Include Generalization. OOPSLA 1989 Proceedings

Pulkkinen T. ym., 2003. libsigc++ kotisivu. , saatavilla www muodossa <http://libsigc.sourceforge.net/>

Riehle D. ja Gross T., 1998. Role Model Based Framework Design and Integration.

Scherfl M., Neuhold E.J., 1988. Object class definition by generalization using upward inheritance. Proceedigns of The Fourth International Conference on Data Engineering

Snyder, A., 1986. Encapsulation and Inheritance in object-oriented programming languages. Conference Proceedings of OOPSLA ' 86

Swen B., 2000. Object-oriented programming with induction. ACM SIGPLAN Notices

Taivalsaari A., 1995. On the Notion of Inheritance. ACM Computing Surveys

Trolltech company, 2003. QT 3.1 Whitepaper, saatavilla www muodossa <ftp://ftp.trolltech.com/qt/pdf/3.1/qt-whitepaper-a4-web.pdf>

Wegner P., 1990. Concepts and paradigms of object-oriented programming. ACM OOPS Messenger

LIITE 1. Kaksipäisen jono ongelma

Esimerkki jonon ja kaksipäisen jonon ongelmaan käyttäen myöhäistä luokittamista. Esimerkki on Baumgartnerin ja Russon C++ laajennoksesta signatures.

```
template <typename T> class DoubleLinkedList {
public:
void enqueHead(T);
T dequeHead();
void enqueTail(T);
T dequeTail();
};

template <typename T> signature DEQueue {
void enqueHead(T);
T dequeHead();
void enqueTail(T);
T dequeTail();
};

template <typename T> signature Queue {
void enqueTail(T);
T dequeHead();
};

Queue<int>* q1 = new DoubleLinkedList<int>;
DEQueue<char *>* q2 = new DoubleLinkedList<char*>;
```

Baumgartner ja Russo huomauttaa, että sama efekti voidaan saavuttaa käyttämällä C++:n moniperintää tekemällä `Queue`:sta ja `DEQueue`:sta abstrakteja tyyppejä ja periyttää `DoubleLinkedList` näistä. Mutta tämäkin moniperinnällä tehty ratkaisu ei toimi, jos pitäisi lisätä pinotyyppi, `Stack`, jossa olisi metodit `push` ja `pop`. Moniperinnässä täytyisi joko esittää uusi (silta)luokka, joka

toteuttaa `push`:in ja `pop`:in delegoimalla ne `DoubleLinkedList`:lle tai lisätä `push` ja `pop` `DoubleLinkedList`:iin. Molemmat selvästi ovat huonoja ratkaisuja. Sen sijaan signature:lla voitaisiin tehdä suoraan tyyppi `Stack` ja käyttää myöhäistä aliasointia sijoituksesssa, jotta `enqueHead` nimettäisiin `pop`:ksi ja `dequeHead` nimettäisiin `push`:ksi.