Pasi Manninen

MONIPERINTÄ C++-, EIFFEL- JA JAVA-OHJELMOINTIKIELISSÄ

Tietojärjestelmätieteen
pro gradu -tutkielma
21.5.2002

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

TIIVISTELMÄ

Manninen, Pasi Juhani
Moniperintä C++-, Eiffel- ja Java-ohjelmointikielissä / Pasi Manninen.
Jyväskylä: Jyväskylän yliopisto, 2002.
70 s.
Pro gradu -tutkielma.

Tutkielman tavoitteena on selvittää, mitä on perintä sekä tutkia tarkemmin erityisesti moniperintää. Tavoitteena on myös tutkia, mihin moniperintää käytetään ja kuinka sitä voidaan käyttää C++-, Eiffel- ja Java-ohjelmointikielissä. Tutkielmassa vertaillaan näiden kielten selkeimpiä eroja moniperinnän ja sen käyttömahdollisuuksien suhteen sekä arvioidaan myös näiden toteutusten hyviä ja huonoja puolia. Näiden moniperinnän toteutuksiin annetaan myös joitakin parannusehdotuksia.

Tutkielman aluksi esitellään teoriaa perinnästä ja sekä sen erilaisia käyttö- ja jakotapoja. Keskeisimpänä esitellyistä jakotavoista on jako käsitteelliseen mallintamiseen ja muuhun perintään eli ohjelmiston uudelleenkäyttöön tähtäävään perintään. Seuraavaksi syvennytään moniperintään sekä siihen liittyvään teoriaan ja tutkimukseen. Seuraavissa luvuissa esitellään moniperintä käytännössä kohteenamme olevissa kielissä ja lopuksi vertaillaan näiden kielten tarjoamia vaihtoehtoja keskenään. Tutkielman alun teoreettinen osa on tutkimusotteeltaan kirjallisuuskatsaus, mutta olio-ohjelmointikieliä käsitteleviltä ja vertailevilta osiltaan myös konstruktiivinen.

AVAINSANAT: Perintä, moniperintä, olio-ohjelmointi, C++, Eiffel, Java

SISÄLLYSLUETTELO

1 JOHDANTO 1

2 PERINTÄ JA MONIPERINTÄ 4
2.1 Perintä 4
2.1.1 Yleistä perinnästä 4
2.1.2 Perinnän jakotapoja 8
2.1.3 Perinnän tarkastelumallit 10
2.1.4 Perinnän toteutus 10
2.2 Moniperintä 11
2.2.1 Yleistä moniperinnästä 11
2.2.2 Perinnän tarkastelumallit moniperinnässä 12
2.2.3 Haarautuva ja riippumaton moniperintä 13
2.2.4 Moniperinnän ongelmia 15
2.3 Yhteenveto 16

3 C++:N MONIPERINTÄ 18
3.1 Yleistä 18
3.1.1 Periytymistapa 18
3.1.2 Suojamääreet periytymistavassa 19
3.2 Ongelmakohtia 20
3.2.1 Yksityinen periytymistapa 20
3.2.2 Nimikonfliktit 22
3.2.3 Haarautuva moniperintä 27
3.3 Korjausehdotuksia 31
3.4 Yhteenveto 32

4 EIFFELIN MONIPERINTÄ 33
4.1 Yleistä 33
4.1.1 Perintälauseke 33
4.1.2 Nimet 35
4.1.3 Syrjäytys 36
4.2 Ongelmakohtia 37
4.2.1 Uudelleennimeäminen 37
4.2.2 Piirteiden valinta 39
4.2.3 Kovariantti syrjäytystapa 41
4.3 Korjausehdotuksia 43
4.4 Yhteenveto 44

5 JAVAN MONIPERINTÄ 45
5.1 Yleistä 45
5.1.1 Perintä 45
5.1.2 Syrjäytys 46
5.1.3 Liittymäluokkien moniperintä 47
5.2 Ongelmakohtia 49
5.2.1 Nimikonfliktit 49
5.2.2 Rajoittuneisuus 53
5.3 Korjausehdotuksia 55

5.4 Yhteenveto 56

6 KIELTEN VERTAILU 57
6.1 Periytymistapa 57
6.1.1 Yleistä periytymisen vertailusta 57
6.1.2 Suojamääreet 58
6.1.3 Lukumääräsuhteet 59
6.2 Syrjäytys 61
6.2.1 Yleisiä eroja 61
6.2.2 Syrjäytyksen määräytyminen 62
6.3 Moniperinnän nimikonfliktit 63
6.4 Yhteenveto 64

7 YHTEENVETO 66

LÄHDELUETTELO 69

1 JOHDANTO

Tässä tutkielmassa tullaan selvittämään, mitä on moniperintä, ja tutkitaan miten sitä voidaan käyttää kolmessa erilaisessa ohjelmointikielessä: C++:ssa, Eiffelissä ja Javassa. Näistä kielistä C++ ja Java ovat tällä hetkellä erittäin hallitsevassa asemassa ohjelmistoalalla. Nämä kaksi ovat myös laajassa kaupallisessa käytössä ja tutkimuksen sekä mielenkiinnon kohteena yliopistomaailmassa. Vastaavasti kiinnostus Eiffeliin näyttäisi tällä hetkellä olevan rajoittunut vain yliopistomaailmaan. C++:n ja Javan merkittävä asema näyttäisi jatkuvan myös tulevaisuudessa, näkyvissä ei olekaan mitään varteenotettavaa haastajaa, joka mullistaisi ohjelmistoalan. Eiffel on kuitenkin joidenkin perintään ja sen hallintaan liittyvien monipuolisten – mutta myös monimutkaisten – ominaisuuksiensa puolesta mielenkiintoinen vertailukohta näille kahdelle laajempaa kaupallista menestystä saavuttaneelle kielelle.

Olio-ominaisuuksista perintä on ehkä kaikkein oliokeskeisin. Juuri perintä tekee kielestä oliosuuntautuneen (Koskimies 2000, s. 69). Moniperintä on perinnän mielenkiintoinen ja monimutkaisempi muoto. Moniperintä oliokielissä on hyvinkin kiistelty ja myös paljon tutkittu olio-ominaisuus. Sitä on usein pidetty liian monimutkaisena eikä riittävän hyödyllisenä, jotta se olisi mahdollisten aiheuttamansa vaivojensa arvoinen (Koskimies 2000, s. 80). Toisaalta perinnän käyttöä yleensä ohjelmoinnissa on pidetty usein liiallisena, kun koostaminenkin riittäisi (Sakkinen 1989, s. 43). Nyt onkin mielenkiintoista lähteä tutkimaan miten tämä keskeinen, mutta kiistelty oliopiirre on toteutettu kohteena olevissa kielissä ja minkälaisia mahdollisuuksia nämä kielet tarjoavat tämän käyttöön.

Keskeisenä ongelmana onkin selvittää, mitä ovat perintä ja moniperintä sekä tutkia:

- Millaisia ratkaisuja ja mahdollisuuksia kielet tarjoavat moniperintään ja sen järkevään sekä hallittuun käyttöön olio-ohjelmoinnissa?
- Mitä ongelmia näiden kielten moniperinnän toteutuksissa on ja mitä on tehty hyvin?
- Miten kielissä on päädytty tällaisiin toteutuksen ratkaisuihin?
- Onko näiden kielten moniperinnän toteutuksissa jotakin parannettavaa?

- Voidaanko moniperinnän aiheuttamia mahdollisia ongelmatilanteita välttää käyttämällä jotain toista tekniikkaa?

Tutkielmassa tullaan sivuamaan jonkin verran sellaisia kielten keskeisiä käsitteitä ja ominaisuuksia, jotka ovat seurausta moniperinnästä tai ovat muuten välttämättä esiteltävä samassa yhteydessä perinnän ja moniperinnän kanssa. Tutkimuksen varsinainen itsetarkoitus ei ole valita kielistä ehdottomasti parasta vaihtoehtoa, tarkoitus on paremminkin esitellä moniperintää ja vertailla näiden kielten tarjoamia toteutuksia keskenään. Tutkielmassa esitetään parannusehdotuksia kielten moniperinnän toteutukseen ja myös tapoja perinnän hallitumpaan sekä järkevämpään käyttöön käytännön ohjelmointityössä. Tämä tutkimus suoritetaan pääasiallisesti kirjallisuuskatsauksena. Kolmannessa, neljännessä, viidennessä ja kuudennessa luvussa tutkimusote on myös osittain konstruktiivinen. Toisessa luvussa käsittelemme pääasiallisesti teoriaa perinnästä ja moniperinnästä. Kolmas, neljäs ja viides luku keskittyvät moniperinnän esittelyyn ja arviointiin C++:ssa, Eiffelissä ja Javassa. Kuudes luku on kielten ominaisuuksien vertailuluku ja seitsemäs luku on tutkielman yhteenvetoluku.

Tutkielman toisessa luvussa käsittelemme siis yleisesti perintää sen erilaisia muotoja sekä sitten tarkemmin moniperintää. Käymme läpi teoriaa sekä tehtyä tutkimusta perinnän ja erityisesti moniperinnän alueella. Tämä luku antaa meille tarpeeksi tietoa, jotta seuraavien lukujen moniperinnän arviointi ja vertailu kohteenamme olevissa kielissä olisi mahdollista ja mielekkäästi esiteltävissä. Luvussa tulemme esittämään myös perinnän erilaisia tyylejä tai käyttötapoja sekä näkemyseroja, joista keskeisimpänä ehkä jako *käsitteelliseen erikoistumiseen* (conceptual specialization) ja muuhun perintään eli pääasiallisesti ohjelmakoodin perintään. Kun keskeiset termit ja teoriaa on esitelty tarpeeksi, luvussa siirrytään moniperintään ja sen ominaispiirteisiin. Seuraavissa kolmessa ohjelmointikieliä ja näiden ominaisuuksia tutkivissa luvuissa keskitymmekin pääasiallisesti vain moniperintään ja sen myötä esiin tuleviin ongelmiin.

Kolmannessa luvussa tarkastelemme moniperintää sekä tämän ongelmia C++-ohjelmointikielessä. Keskeisimmiksi ongelmiksi nousee C++:n moniperinnän rajoittumattomuus ja monimutkaisuus sekä nimikonfliktit, jotka käytännössä ovat

ongelmana tutkielmamme jokaisessa kielessä. Neljäs luku keskittyy moniperinnän tarkasteluun Eiffel-ohjelmointikielessä. Eiffelin moniperinnän ongelmat liittyvät myös monimutkaisuuteen ja laajuuteen. Eiffelissä on ainoana tutkielmamme kielenä pyritty ratkaisemaan moniperintää vaivaavat nimikonfliktit. Tosin tämän ongelman ratkaisu voi tuoda mukanaan lisää monimutkaisuutta jo ennestäänkin laajaan moniperintään Eiffel-ohjelmointikielessä. Viidennessä luvussa tarkastelemme suhteellisen rajoitettua moniperintää Java-ohjelmointikielessä. Javan moniperinnän suurin ongelma onkin sen rajoittuneisuus – sen käyttömahdollisuudet ovat hyvin rajalliset verrattuna tutkielmamme kahteen muuhun kieleen.

Kuudennessa luvussa tarkastelemme keskeisimpiä eroja kielten välillä. Tässä vaiheessa on havaittavissa joitakin hyvinkin suuria eroja kielten ominaisuuksien ja moniperinnän toteutusten välillä. Tässä luvussa käsittelemme myös joitakin mahdollisia ratkaisuja ja rakenteita, mitä jollain kielellä voi tehdä mutta mikä toisten moniperinnässä ei ole mahdollista. Tutkielman päättävä seitsemäs luku on yhteenveto koko tutkielmasta.

2 PERINTÄ JA MONIPERINTÄ

Tässä luvussa tulemme käsittelemään perintää ja moniperintää. Ensin käymme läpi yleistä teoriaa ja terminologiaa sekä olio-ohjelmoinnin käsitteitä, jotka liittyvät perintään ja joiden ymmärtäminen on välttämätöntä moniperinnän käsittelyn kannalta. Seuraavaksi esittelemme erilaisia perinnän jakotapoja ja sitten siirrymme tutkielman päämielenkiinnon kohteeseen eli moniperintään. Moniperintää voidaan pitää eräänlaisena perinnän erikoistapauksena, joka tuo monia uusia asioita ja vaatimuksia sekä monimutkaisuutta esille perinnän toteutuksen, toiminnan ja myös käytön suhteen olio-ohjelmoinnissa.

2.1 Perintä

2.1.1 Yleistä perinnästä

Ohjelmoinnissa *perintä* (inheritance) esiteltiin varsinaisesti ensi kertaa SIMULA 67 -ohjelmointikielessä. Tällöin nimi tosin oli vielä *konkatenaatio* (concatenation) (Sakkinen 1989, s. 39). Nykyään olio-ohjelmoinnissa on vakiintunut termiksi perintä, mutta joitakin vaihtoehtoisia termejä on vielä käytössä. Javassa puhutaan perinnän yhteydessä *laventamisesta* (extend) ja C++:ssa *johdetuista luokista* (derived class) (Gosling 1997) ja (Stroustrup 1997). Simula-ohjelmointikielen ensi kertaa esittelemä perintäominaisuus rajoittui vain yksittäisperintään. Näin luokalla sai siis olla vain yksi *välitön yliluokka* (direct superclass). Yksittäisperinnässä luokka perii tämän yliluokkansa ominaisuuksia. Nykyään vielä monet oliokielet ovat pitäneet kiinni tästä rajoituksesta, vaikka moniperinnän sallivia kieliä on jo monia, kuten kaikki tämän tutkielman käsittelemät kielet. Perinnän on sanottu tulleen ohjelmointimaailmaan luonnontieteistä ja lähinnä biologiasta, jossa erilaisten lajien (tyyppien) välille on kehitelty periytymishierarkioita eli *taksonomioita* (taxonomy) (Cardelli 1988, s. 138).

Oliokielissä tieto jaetaan ja ryhmitellään sopiviin yksiköihin eli luokkiin. Luokka toteuttaa tietoabstraktion ja näin se toimii tiedon koteloinnin yksikkönä (Wegner 1987, s. 170). Tieto sijoitetaan luokkaan sen *attribuutteihin* (attribute) ja sitä

muokataan sekä hallitaan luokan *operaatioilla* (operation). Attribuutteja ja operaatioita kutsutaan luokan *piirteiksi* (features). Perintä nähdään usein oliokeskeisyydessä vastakohtana *koostamiselle* (aggregation). Koostamisessa otetaan käyttöön jo olemassa olevia luokkia. Nämä koosteen osat, joita kutsutaan myös *komponentiksi* (component) otetaan mukaan koosteluokkaan sen attribuuteiksi. Yleensä attribuutit eivät pidä sisällään varsinaisesti koosteen osaolioita, vaan ainoastaan viitteitä näihin. Perinnässä sen sijaan laajennetaan jo olemassa olevia luokkia perimällä näistä uusi luokka. Peritylle luokalle siirtyvät siis sen yliluokkien piirteet, sen omien piirteiden lisäksi. Piirteiden perimisen lisäksi luokka voi myös *syrjäyttää* eli uudelleen määritellä (redefine) tai *viivästyttää* eli peruuttaa (defect, cancel) yliluokilta perittyjä piirteitä (Taivalsaari 1996, s. 439). Joskus jopa piirteiden *uudelleennimeäminen* (rename) on mahdollista. Esitetään perintä vielä matemaattisemmin, kuten Bracha ja Cook ovat sen esittäneet (Bracha 1990, s. 2):

$$C = \Delta(P) \oplus P.$$

Tässä *P*:n voidaan katsoa olevan (yliluokalta) perityt piirteet ja Δ *(P)* on tyypin itsensä tähän lisäämät (myös muuttamat tai peruuttamat) piirteet. Näille kahdelle piirrejoukolle tehdään sitten binaarinen yhteenlaskuoperaatio, josta saadaan tulosjoukko *C*. Mikäli kummassakin joukossa on samanniminen piirre, binaarisessa yhteenlaskussa otetaan piirre tulosjoukkoon vasemmanpuoleisesta lähtöjoukosta. Näin tapahtuu siis piirteen syrjäytys. On hyvä huomioida, että edellä esitetty kaava koskee yksittäisperintää.

Edellä mainittuja perinnässä olevia mahdollisuuksia eli syrjäytystä, viivästyttämistä ja uudelleennimeämistä ei ole koostettaessa käytössä. Perinnän, niin kuin myös koostamisen perusajatus on ohjelmiston eli ohjelmakoodin uudelleenkäytettävyyden lisäämisessä, mutta ohjelmiston analyysin näkökulmasta myös kohdeympäristön mahdollisimman tarkassa käsitteellisessä mallintamisessa. Näiden kahden eri näkemyksen vastakkainasettelu tulee esille useasti myös tässä tutkimuksessa.

Perinnässä luokka siis perii piirteitä yliluokaltaan. Perivä luokka on yliluokkansa *aliluokka* (subclass). Perivä luokka on myös välittömän yliluokan välitön aliluokka. Joskus puhutaan myös *vanhemmasta* (parent) ja *lapsesta* (child) tai *kantaluokasta* (base class) ja jo aiemmin mainitusta johdetusta luokasta. Tässä tutkielmassa pyrimme yhdenmukaisuuden vuoksi käyttämään termejä yliluokka ja aliluokka. Kaikkia luokan – sekä välittömiä, että näiden kautta perittyjä – yliluokkia voidaan kutsua *esivanhemmiksi* (ancestor). Vastaavasti kaikkia aliluokkia – välittömiä ja näiden omia aliluokkia – kutsutaan *perillisiksi* (decendant) tai *jälkeläisiksi*.

Perinnässä muodostuu erityinen suhde perijän ja perittävän eli yli- ja aliluokan välille. Voidaan katsoa, että perittävä antaa joitakin piirteitä tai ominaisuuksia perijälle (Taivalsaari 1996, 439). Näin perijä saa omakseen omien piirteidensä lisäksi esivanhempiensa kaikki tai rajatusti halutut piirteet. Aliluokka siis *erikoistuu* (specialization) ja luokkahierarkiassa aliluokasta yliluokkaan siirryttäessä yliluokka *yleistää* (generalization). Näin ollen aliluokka täydentää luokkaa jonka se perii, usein lisäämällä siihen uusia piirteitä. Syrjäyttäminen tapahtuu lisäämällä aliluokkaan samannimisiä piirteitä kuin yliluokassa on.

Yliluokan operaatioiden syrjäyttäminen ja niiden *myöhäinen sidonta* (late binding) synnyttää olio-ohjelmoinnissa *monimuotoisuutta* (polymorphism). Kuormittamiseen emme puutu sen enempää kuin selvittämällä sen eron syrjäyttämiseen: kuormittamisessa operaation *kutsumuoto* (signature) on erilainen, vaikka sen nimi onkin sama kuin jonkin toisen operaation nimi. Syrjäyttämisessä operaation kutsumuodon täytyy olla identtinen syrjäytettävän operaation suhteen, muuten syrjäyttäminen ei toteudu ja sen sijaan tapahtuukin nimen kuormittaminen. Syrjäyttäminen voi tosin tapahtua joskus vaikka kutsumuoto ei olisikaan identtinen, jos se täyttää muuten tietyt kriteerit. Tällaista vapaamuotoisempaa syrjäytystapaa voidaan käyttää joissain ohjelmointikielissä mm. Eiffelissä, joka on yksi tämän tutkielman kohdekielistä. Tällöin kuormitus ei yleensä ole mahdollista.

Periytymistä käsiteltäessä oliokielten luokat samaistetaan usein *tyypeiksi*. Näin voidaan puhua luokan *instanssien* eli *olioiden* tyypeistä. Olion tyyppinä voidaan pitää siis sitä luokkaa mistä se on luotu, jota kutsuaan myös sen *perusluokaksi* (Koskimies 2000, s. 68). Olion tyyppiä voidaan pitää myös sen vanhempien *alityyppinä* (subtype).

Luokka voi olla myös *abstrakti* jolloin siitä ei voi luoda olioita. Abstraktissa luokassa on yksi tai useampia abstrakteja eli avoimia operaatioita, jotka ovat ilman toteutusta. Abstraktin luokan perivä luokka joutuu antamaan toteutuksen näille operaatioille, jotta siitä voisi tulla *konkreetti* eli luokka josta voidaan luoda olioita.

Perintäsuhteet luokkien välillä muodostavat luokkahierarkian. Yksittäisperinnän vallitessa luokkahierarkia on puumainen, tällöin siinä on yleensä yksi *juuriluokka* (root class) ja useita *lehtiluokkia* (leaf class). Perinnän luokkahierarkia voi olla myös *metsä*. Tällainen tilanne ilmenee sellaisissa kielissä – kuten esimerkiksi C++:ssa – joissa käytetään luokkia mutta joissa ei ole luokkahierarkian yhteistä juuriluokkaa. Hierarkian juuriluokalla ei ole koskaan yliluokkaa ja lehtiluokilla ei ole aliluokkia. Moniperinnässä tämä tilanne monimutkaistuu, kuten seuraavassa kohdassa tulemme havainnollistamaan.

Perintää on jopa pidetty oliokeskeisyyden tärkeimpänä piirteenä, joka juuri erottaa oliokielet muista ohjelmointikielistä (Wegner, 1987). Perintä on myös ollut jatkuvan kiistelyn ja tutkimuksen kohteena mm. sen tuoman uudelleenkäytettävyyden, monimutkaisuuden ja merkityksensä johdosta (Taivalsaari 1996, 438-479). Perinnän käytöllä voi olla monia erilaisia tarkoituksia (Meyer 1997, 494). Perinnän katsotaan rikkovan jossain määrin *tiedon kapselointia* (data encapsulation), koska perinnässä aliluokka pääsee usein suoraan käsiksi yliluokkansa piirteisiin (Snyder 1986, 38-45). Tällöin luokassa oleva tieto ei olekaan suojattu kaikilta muilta luokilta. Tätä voidaan tosin kontrolloida muuttamalla luokkien *suojamääreitä* (access specifiers). Piirteiden näkyvyyttä eli siis saavutettavuutta aliluokasta voidaan kontrolloida suojamäärein. Jos oliot sen sijaan ajatellaan kapseloinnin yksikkönä luokkien sijasta, perintä ei edes riko kapseloinnin käsitettä, koska olio on luotu luokastaan ja on tyyppinä myös kaikkia oman luokkansa esivanhempia.

On huomioitava, että piirteiden näkymättömyys aliluokkaan ei kuitenkaan välttämättä estä niiden periytymistä. Suojamääreillä voidaan vaikuttaa perintään ja myös osittain monimutkaistaa sitä. Joissain oliokielissä kuten esimerkiksi C++:ssa suojamääreet vaikuttavat perintään huomattavasti, tähän palaamme tarkemmin tulevissa luvuissa. Perinnän ideana voidaan myös nähdä juuri aliluokan erityinen suhde yliluokkaansa nähden. Luokalla voi olla siis useita erilaisia *palvelurajapintoja* (interface): yksi

jälkeläisluokille ja toinen muille luokille eli *asiakasluokille* (client). Tämä saadaan aikaan edellä mainituilla eritasoisilla suojamääreillä; näillä luokan piirteet voidaan piilottaa muilta kuin luokan välittömiltä jälkeläisiltä.

2.1.2 Perinnän jakotapoja

Perintä voidaan jakaa luokkien välisten perintäsuhteiden määrän mukaan yksittäisperintään ja moniperintään (Koskimies 2000, 68-90). Moniperinnässä luokalla on välittöminä vanhempina useampi kuin yksi luokka, kun taas yksittäisperinnässä luokka perii suoraan vain yhden luokan. Tulevissa luvuissa tulemme syventymään juuri moniperintään.

Perintä voidaan jakaa myös käsitteelliseen perintään eli käsitteelliseen erikoistumiseen ja muuhun perintään. Muuta perintää pidetään pääasiallisesti puhtaana ohjelmakoodin uudelleenkäyttönä. Tätä jakoa on verrattu myös suoraan Aristoteleen perinteiseen asioiden jakoon *merkityksellisiin* (essential) ja *sattumanvaraisiin* (accidental). Juuri ensimainittua perinnän käyttöä pidetään puhtaana eli Aristoteleen merkityksellisenä perintänä, kun taas kaikkea muuta pidetään sattumanvaraisena (Taivalsaari 1996, s. 445). Mutta niin kuin Taivalsaari edellä mainitussa artikkelissaan toteaa, ei yksikään tunnettu ohjelmointikieli takaa mekanismeja joilla voitaisiin taata, että perintää käytettäisiin ainoastaan käsitteelliseen perintään. Tällöin perinnän käyttö jää ohjelmoijan kurin ja teoreettisen tietämyksen (ammattitaidon) varaan. Muunlaista eli sattumanvaraista perinnän käyttöä on pidetty jopa perinnän väärinkäyttönä. Tällaisesta perinnän käytöstä on lukuisia esimerkkejä alan kirjallisuudessa ja jopa ohjelmointikielten luokkakirjastoissa.

Perinnän käyttäminen juuri käsitteelliseen mallintamiseen on vahvasti esillä eurooppalaisessa näkökulmassa olioparadigmaan, kun taas amerikkalaiset tutkijat näkevät perinnän enemmin ohjelmakoodin uudelleenkäytön näkökulmasta. Periytyminen voidaan nähdä siis tiukasti joko teknisenä ohjelmiston uudelleenkäyttöä tukevana tai käsitteellistä mallintamista tukevana tekniikkana. (Koskimies 2000, s. 71)

Perintä voidaan jakaa myös sen käyttötarkoituksen tai tyylin mukaan eri kategorioihin. Esimerkiksi Meyer on esittänyt artikkelissaan (Meyer 1996, 105) kolme pääkategoriaa: *mallin perinnän* (model inheritance), *toiminnan perinnän* (software inheritance) ja *muuttavan perinnän* (variation inheritance). Näille kolmelle pääkategorialle on lisäksi esitelty yhteensä kaksitoista erilaista alakategoriaa. Näistä neljän voidaan katsoa kuuluvan puhtaasti käsitteellisen perinnän piiriin ja loput uudelleenkäytön piiriin (Koskimies 2000, s. 75). Neljä mallin perinnän perintätyyliä: *alityypitys* (subtyping), *näkymän perintä* (view inheritance), *rajoittava perintä* (restriction inheritance) ja *laajentava perintä* (extension inheritance) kuuluvat selkeästi siis käsitteelliseen mallintamiseen, kun loput kahdeksan ovat enemmänkin ohjelmakoodin perintää. Tosin neljä muuttavan perinnän tyyliä ovat vähän kumpaakin, sekä mallin että toiminnan perintää (Joyner 1999, s. 118). Näitä eri variaatioita emme kuitenkaan käsittele tässä tutkielmassa tämän tarkemmin, koska alakategoriat poikkeavat toisistaan niin vähän. Artikkelin käsittelemistä perinnän jakotavoista mielenkiinto kohdistuukin mallin perinnän ja toiminnan perinnän eroavaisuuksiin.

Ohjelmakoodin uudelleenkäytöstä ja käsitteellisen perinnän eroista voidaan ottaa esimerkki Meyerin Eiffel-ohjelmointikieltä kuvaavasta kirjasta (Meyer 1997, s. 530). Säiliöluokkien moniperintä esimerkki sisältää käsitteellisen perinnän ja toiminnan perinnän käytön yhdistämisen. Puhtaan käsitteellisesti esitelty luokka ARRAYED_STACK ei voi millään olla sekä pinon että taulukon jälkeläinen. Pinoa on tarkoitus käsitellä ainoastaan sen päällimmäisestä elementistä eli kaikkein suurimmasta indeksistä, kun taas taulukkoon voidaan perinteisesti viitata aina käyttämällä mitä tahansa haluttua indeksiä. Näin ollen luokka ei voi puhtaasti käsitteellisesti ajatellen olla millään tyypiltään sekä pino että taulukko. Kyseisessä rakenteessa joitain taulukon ominaisuuksia, jotka eivät sovi pinolle, on piilotettu, jotta niitä ei voisi käyttää. Meyer onkin tunnetusti suosinut perinnän, erityisesti moniperinnän käyttöä ohjelmakoodin uudelleenkäyttöön (Koskimies 2000, s. 80).

2.1.3 Perinnän tarkastelumallit

Perinnän tarkastelu voidaan myös jakaa lineaariseen, attribuuttikeskeiseen tai alioliomallin näkökulmaan. Lineaarisessa tarkastelumallissa vanhemmat sekä niiden piirteet peritään lineaarisesti, siinä järjestyksessä kun ne on esitelty luokan esittelyssä. Samannimiset piirteet, mikäli niiden tyypit sallivat, samaistetaan yhdeksi (Cardelli 1988, s. 144). Näin piirteiden perintäjärjestys voi siis vaikuttaa niiden periytymiseen. Cardellin artikkelissaan käsittelemä perintä on implisiittisesti tapahtuvaa ja se perustuukin *rakenneyhtäläisyyteen* (subtype relation), eikä tavanomaisempaan oliokielissä käytettyyn *nimiyhtäläisyyteen* (name equality), jossa periytymissuhde ilmaistaan yksinkertaisesti luettelemalla perittyjen yliluokkien nimet luokan esittelyssä.

Attribuuttikeskeisessä mallissa yliluokkia voidaan periä attribuutti kerrallaan eikä vain kokonaisina olioina. Tätä tarkastelumallia voidaan käyttää apuna hahmotettaessa monimutkaisia moniperintähierarkioita esimerkiksi tutkimassamme Eiffel-ohjelmointikielessä. Alioliomallissa vanhemmat eli yliluokat nähdään luokan *aliolioina* ja itse aliluokka näiden *ylioliona* (Sakkinen 1989). Yliluokat peritään näin siis kokonaisina aliolioina. Aliolionäkökulma helpottaakin jossain tapauksissa moniperinnän tarkastelua. Näitä malleja tarkastelemme lisää seuraavassa kohdassa moniperinnän yhteydessä, jossa erityisesti alioliomallin käsittely on havainnollisempaa ja mielekkäämpää.

2.1.4 Perinnän toteutus

Perinnän toteutuksen ohjelmointikielessä voidaan katsoa tapahtuvan joko delegaation tai konkatenaation eli liitoksen avulla (Taivalsaari 1996, s. 459). Niin kuin aiemmin mainitsimme, Simulan perintää kutsuttiin alunperin juuri konkatenaatioksi. Delegointitekniikassa yliluokan eli aliolion piirteeseen päästään käsiksi viitteen avulla. Tämä tekniikka voidaan rinnastaa koostamiseen jos ajatellaan, että kyseiseen olioon pääsisi luokasta käsiksi myös, mikäli luokassa olisi yliluokkatyyppinen viite attribuuttina. Liitoksessa taas yliluokkaan eli aliolioon pääsee käsiksi, koska se on liitetty eli siis sisälletty osaksi luokkaa. Näin mitään erillistä viitettä yliluokkaan ei

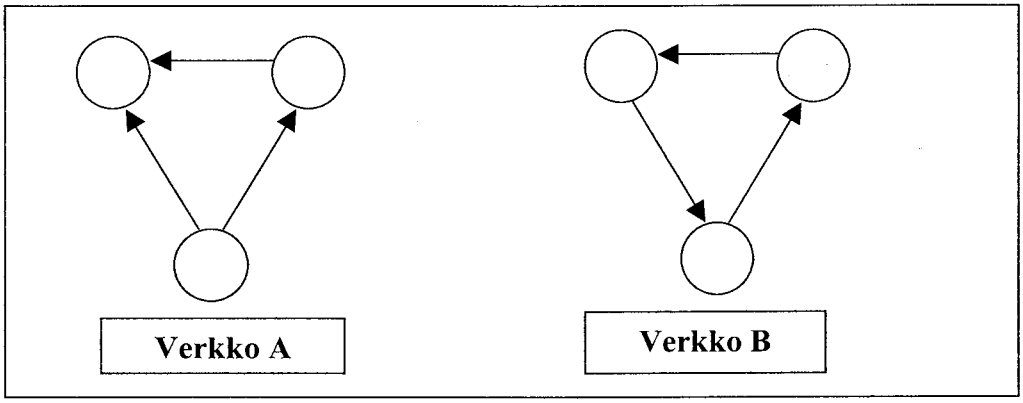
tarvita. Liitostekniikka vastaakin paremmin varsinaista perinnän perusajatusmallia, kun taas delegointi viittaa ennemminkin vastuun siirtämiseen jonnekin muualle, perinnän kyseessä ollessa jollekin luokan yliluokalle.

2.2 Moniperintä

2.2.1 Yleistä moniperinnästä

Sekä käsitteellisen mallintamisen että koodin uudelleenkäytön kannalta on joskus tarkoituksenmukaista luopua yksittäisperiytymisen rajoituksellisuudesta (Koskimies 2000, s. 80). Tällöin peritään useampia luokkia. Moniperinnässä luokalla on useita välittömiä esivanhempia, tarkemmin sanoen luokka perii suoraan useamman yliluokan. Tämä ominaisuus monimutkaistaa luokkien periytymishierarkiaa huomattavasti. Hierarkiaa ei voida enää kuvata puumaisena, jossa olisi yksi juuriluokka ja josta hierarkia leviäsi päättyen lehtiluokkiin. Perintähierarkiasta muodostuukin tällöin verkko, DAG (directed acyclic graph).

Verkossa luokat ovat sen *solmuja* ja perintäsuhteet solmujen välissä olevia *välejä*. Solmujen ja niiden välien muodostamaa reittiä verkossa jostain tietystä solmusta eli siis luokasta toiseen voidaan kutsua luokkien väliseksi *perintäpoluksi* (derivation path) (Sakkinen 1992, s. 79). Jonkin tietyn yliluokan voidaan katsoa olevan *saavutettavissa* (accessible) jostain sen perillisestä tiettyä perintäpolkua pitkin, mutta samanaikaisesti sama yliluokka voi olla *saavuttamattomissa* (inaccessible) jotain toista polkua pitkin. Tällainen tilanne on mahdollinen ainakin C++:ssa, jossa suojamääreillä voidaan vaikuttaa luokkien periytymiseen (Sakkinen 1992, s. 79). Verkon syklittömyys luokkien välisissä suhteissa tarkoittaa sitä rajoitetta, että mikään luokka ei voi olla itsensä esivanhempi eli periä itseään suoraan tai epäsuorasti (Koskimies 2000, s. 80). Seuraavassa kuviossa (KUVIO 1) on kuvattu kaksi verkkoa:



KUVIO 1: Perintähierarkia verkkona

Näissä kahdessa verkossa on kummassakin kolme solmua. Näitä verkkoja voidaan pitää periytymishierarkioina, missä solmujen välien nuolet osoittavat periytymisen suunnan. Kuviossa olevista verkoista verkko A on syklitön ja näin kyseessä voisi olla siis periytymishierarkia. Tässä tilanteessa hierarkian alin solmu perisi toisen solmuista suoraan ja toisen suoraan sekä epäsuorasti toisen vanhemman kautta. Kuvion verkko B sen sijaan on syklillinen, joten kyseessä ei voisikaan olla periytymishierarkia. Tässä tapauksessa hierarkian jokainen solmu olisi rekursiivisesti itsensä vanhempi ja tämä ei luonnollisestikaan voi olla mahdollista.

Syklittömyysrajoitetta lukuun ottamatta verkko voikin laajeta lähes miten paljon tahansa. Moniperinnän käyttöä ei yleensä olekaan mitenkään muuten rajoitettu sen sallivissa oliokielissä. Esimerkiksi tutkielmamme kohteina olevissa kolmessa oliokielessä ei ole minkäänlaisia rajoitteita moniperinnän perintäsuhteiden lukumäärille; näin verkon solmujen ja välien määrän kasvaminen jää täysin ohjelmoijan vastuulle.

2.2.2 Perinnän tarkastelumallit moniperinnässä

Alioliomallissa yliluokat vastaavat siis aliolioita ja luokka itse ylioliota. Moniperinnässä luokkaan (yliolioon) voi kuulua useita aliolioita (yliluokkia). Alioliomalli helpottaa erityisesti moniperinnän tarkastelua, varsinkin sen

monimutkaisempia ilmentymiä. Esimerkiksi haarautuvassa moniperinnässä voidaan periä useita samantyyppisiä aliolioita eli siis periä sama yliluokka useasti. Alioliomallille vastakkaisena voidaan nähdä jo aiemmin mainittu attribuuttikeskeinen perintä. Tällöin yliluokkia ei peritä kokonaisina alioliona, mutta näiden piirteitä voidaan periä attribuutti kerrallaan halutulla tavalla.

Lineaarisessa mallissa yliluokkia taas peritään tietyssä järjestyksessä ja tällöin järjestyksellä on merkitystä perittyihin piirteisiin. Lineaarisessa moniperinnässä samannimiset piirteet eivät aiheuta moniperinnän nimiristiriitoja, vaan ne syrjäyttävät konfliktitilanteessa toisiaan. Esimerkiksi luokan esittelyn yhteydessä luetelluista yliluokista piirteet peritään siten, että aina aikaisemmin esitellyn luokan piirre syrjäyttää mahdollisen myöhemmän samannimisen piirteen. Lineaarista mallia on tutkittu ja käsitelty tarkemmin Colnet'n ym. kirjassa (Colnet 1991). Lineaarisen lähestymistavan samannimisten piirteiden syrjäytyksellä ei pystytä ratkaisemaan nimikonflikteja täydellisesti ja tavassa ilmeneekin vain sille ominaisia ongelmia (Colnet 1991, s. 35).

2.2.3 Haarautuva ja riippumaton moniperintä

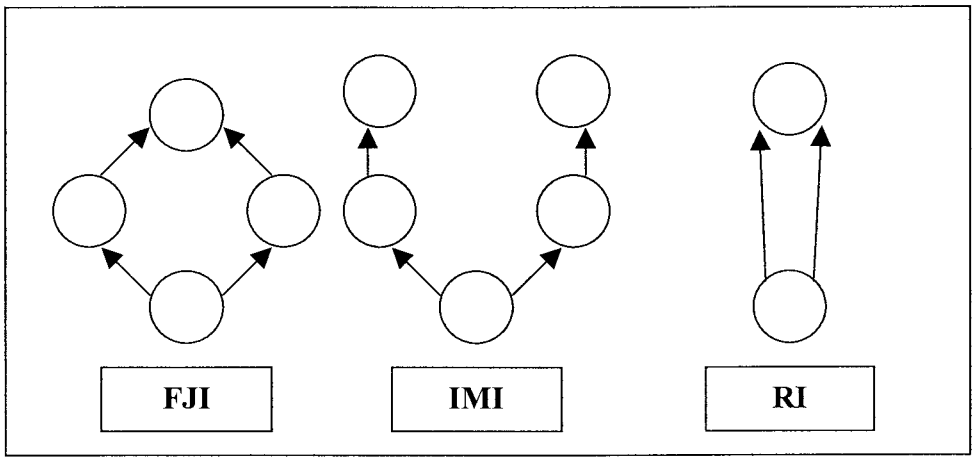
Perintähierarkiaverkon monimutkaistuessa voi usein syntyä siis erittäin hankalia tilanteita. Haarautuvassa moniperinnässä luokan jokin yliluokka voidaan periä luokkaan:

- useammin kuin kerran epäsuorasti usean eri yliluokan kautta.
- kerran suoraan ja lisäksi epäsuorasti vähintään yhden yliluokan kautta.

Tällöin luokalla on monta eri periytymispolkua tähän yliluokkaan. Tällaista tilannetta Sakkinen kutsuu *haarautuvaksi perinnäksi* (FJI, fork-join inheritance) (Sakkinen 1989, s. 46). Tällöin useaa eri periytymispolkua pitkin peritty yliluokka voidaan joko nähdä luokassa yhtenä alioliona tai useana erikseen perittynä alioliona. Haarautuvassa perinnässä samannimisiä piirteitä ei siis välttämättä samaisteta, vaan jokin toistuvasti peritty piirre sisältyy luokkaan useasti sen useaan yliluokka -tyyppiseen alioliioon. Tällöin nimiristiriitojen välttämiseksi on piirteeseen viitattava eksplisiittisesti viitaten

oikeaan (yliluokkaan) aliolioon, jotta tiedetään mitä näiden samannimisistä piirteistä tarkoitetaan.

Saman yliluokan useasta periytymisestä Meyer käyttää termiä *toistuva perintä* (RI, repeated inheritance), mutta tämä poikkeaa kuitenkin jonkin verran haarautuvasta perinnästä (Meyer 1996, s. 543). Toistuvassa perinnässä luokka voi periä jonkun tietyn yliluokan useasti myös suoraan. Tällaisessa tilanteessa luokalla voi olla useita samantyyppisiä aliolioita. Esimerkiksi seuraavan kuvion (KUVIO 2) oikeanpuolimaisessa verkossa on kyseessä toistuva perintä:



KUVIO 2: Haarautuva, riippumaton ja toistuva moniperintä

Joskus luokalla on kuitenkin useampi yliluokka, mutta millään yliluokilla - luokka itse mukaan lukien - ei ole yhteistä vanhempaa. Tällöin tilannetta kutsutaan *riippumattomaksi moniperinnäksi* (IMI, independent multiple inheritance). Tämä on jonkin verran yksinkertaisempi variaatio moniperinnästä. Riippumatonta moniperintää kuvaa edellisen kuvion keskimmäinen verkko.

Esitellään edellä mainittu tilanne haarautuvasta perinnästä käyttäen C++:n syntaksia (KUVIO 2, vasemmanpuolinen verkko):

Esimerkki 2.1:

```
class A { /*...*/ };
class B : public A { /*...*/ };
```

```
class C : public A { /*...*/ };
class D : public B, public C { /*...*/ };
```

Tässä tapauksessa luokka D perisi luokan A sekä luokan B että luokan C kautta. Alioliomallin mukaan luokka D voisi periä kaksi erillistä A -alioliota ja näin sekä sen B että C -alioliolla olisi omansa. Mikäli D -luokasta haluttaisiin viitata sen A -yliluokan johonkin piirteeseen, pitäisi eksplisiittisesti osoittaa kummasta A -tyyppisestä alioliosta on kyse.

2.2.4 Moniperinnän ongelmia

Moniperintää on tutkittu jo useita vuosia siitä lähtien, kun se on oliokieliin ominaisuutena sisällytetty. Tänä aikana on löydetty monia yleisesti tunnistettuja ongelmia, joita moniperintä tuo mukanaan. Perinnästä tosin löytyy monia ongelmia, jotka ilmenevät jo yksittäisperinnässä. Esimerkiksi käsitteellisen mallintamisen tiedostaminen ja perinnän turhan käytön välttäminen jää aina ohjelmoijan vastuulle.

Yleisimmät ongelmat moniperinnässä ovat perintähierarkian laajeneminen ja yleinen monimutkaistuminen verrattuna yksittäisperintään. Moniperinnässä syntyy myös mahdollisuus hankaliin *nimikonflikteihin* (Koskimies 2000, s. 84). Moniperinnän käytössä mahdollisesti syntyvä haarautuva moniperintä aiheuttaa myös yleensä ongelmia ja sen käyttöä olisi vältettävä (Sakkinen 1992, s. 107). Hankala tilanne syntyy varsinkin, jos yliluokka on sen jostakin perillisestä tiettyä perintäpolkua pitkin saavutettavissa, mutta toista polkua pitkin saavuttamattomissa.

Kuitenkin jo moniperinnän yksinkertaisemmassa ilmentymismuodossa eli riippumattomassa moniperinnässä syntyy ongelmia. Yleisin ongelma riippumattomassa moniperinnässä on horisontaaliset nimikonfliktit. Tällaisessa tilanteessa yliluokan piirteeseen on viitattava eksplisiittisesti, muuten ei voida tietää mitä samannimisistä piirteistä tarkoitetaan. Käsitteellisen mallintamisen kannalta tilanne on myös hankala. Esimerkiksi virtuaalisten operaatioiden syrjäyttämisessä voidaan aliluokan samannimisellä operaatiolla syrjäyttää useita yliluokkien samannimisiä operaatioita, vaikka näiden toiminnallisuudella tai käyttötarkoituksella ei olisi mitään yhteistä.

Moniperinnän käyttömahdollisuuden tuomasta perinnän käytön kynnyksen madaltumisesta voi seurata turhaa perinnän käyttöä. Myös tämä on asia, jossa valta jää ohjelmoijalle. Perintähierarkian laajeneminen voi siis johtua ohjelmoijasta, joka voi olla herkempi käyttämään perintää, jos sitä ei ole millään tavalla rajoitettu. Yksittäisperinnässä perinnän käyttöä voi jonkin verran rajoittaa sen yksi ja ainoa käyttömahdollisuus luokkaa kohden.

Perintähierarkian solmujen ja välien määrän kasvaessa kasvaa myös riski edellä mainittuihin nimikonflikteihin. Kun luokka perii useaa eri periytymispolkua pitkin samalla nimellä varustetut eri yliluokkien piirteet, ei voida välttämättä määrittää yksiselitteisesti mikä näistä on voimassa luokan sisällä (Koskimies 2000, s. 84). Tällaisessa moniperintätilanteessa yliluokkien luokkaan periytymisjärjestyksellä voisi olla merkitystä. Tällaista ominaisuuksia on pohdittu mm. jo aiemmin mainitussa Colnet'n ym. kirjassa (Colnet 1991).

2.3 Yhteenveto

Perintä ja eteenkin moniperintä on mielenkiintoinen olio-ominaisuus. Moniperintää onkin vuosien saatossa kritisoitu ja siitä on kiistelty. Useat tutkijat ovat sitä mieltä, että moniperinnän hyödyt eivät riitä korvaamaan siitä aiheutuvia haittoja (Koskimies 2000, s. 80). Toiset pitävät sitä taas ehdottomasti vaivansa arvoisena, varsinkin hallitusti käytettynä (Sakkinen 1989). Näyttäisi myös siltä, että moniperintä soveltuu paremmin ohjelmakoodin uudelleenkäyttöä tukevaan perinnän käyttöön. Varsinaiseen käsitteelliseen mallintamiseen yksittäisperintä on usein aivan riittävä väline. Joskus moniperinnän käyttö on myös mahdollista korvata yksittäisperinnän ja koostamisen yhteiskäytöllä (Koskimies 2000, s. 85). Yksittäisperinnän puuhierarkian ja moniperinnän verkkohierarkian vertaaminen tekee heti selväksi sen, että moniperintä tuo luokkien välisiin suhteisiin huomattavasti lisää kompleksisuutta, koska luokkien välisten suhteiden lukumäärä voi kasvaa valtavasti. Tämä asettaa lisää vaatimuksia moniperinnän käyttämiselle ja sen hallinnalle eli siis hallitulle käytölle olio-ohjelmoinnissa. Ohjelmointikielet eivät aseta valitettavasti juuri mitään rajoitteita sen käytölle, joten vastuu jää täysin kielen käyttäjälle.

Tässä luvussa käsittelimme keskeisintä perintään liittyvää termistöä ja erilaisia perintään liittyviä näkökulmia. Luvussa käsitellyt asia tulevat antamaan tukea seuraavien lukujen perinnän käsittelylle tutkielman kohteena olevissa kolmessa ohjelmointikielessä. Esittelimme mm. perinnän tarkastelua helpottavan alioliomallin ja moniperinnän yksinkertaisemman ilmenemismuodon eli riippumattoman moniperinnän sekä monimutkaisemman haarautuvan moniperinnän. Tärkeimpinä esiteltyinä perinnän käytön eri näkökulmina voidaan pitää perinnän käyttöä käsitteelliseen mallintamiseen ja toisaalta sen käyttöä puhtaasti ohjelmistokoodin uudelleenkäyttöön.

3 C++:N MONIPERINTÄ

Tässä luvussa käsittelemme moniperintää C++:ssa. Tulemme selvittämään, millä lailla moniperintää voidaan käyttää C++:ssa ja minkälaisia erilaisia ominaisuuksia kieli tarjoaa sen käytölle. Käsittelemme mahdollisia ongelmia, joita ilmenee moniperinnän käytössä C++:ssa. Ongelmien esittelyn yhteydessä esitämme myös joitakin keinoja, joilla näitä ongelmia voisi välttää. Tämä tapahtuu pääasiallisesti rajoittamalla moniperinnän varsin vapaata käyttöä C++:ssa.

3.1 Yleistä

3.1.1 Periytymistapa

C++-kielen moniperintä on hyvinkin rajoittamatonta. Ainoa rajoite on se, ettei kahden solmun (luokan) välillä ei voi olla kuin yksi suora väli (Sakkinen 1992, s. 99). Luokka voi periä vapaasti useita eri yliluokkia, mutta myös saman luokan voi periä useasti. *Periytymistapa* voi olla joko *virtuaalinen* tai *ei-virtuaalinen*. C++:ssa myös suojamääreet vaikuttavat periytymiseen, mutta tästä enemmän seuraavassa kohdassa. Periytymistavan mahdollinen virtuaalisuus vaikuttaa perittyjen (yliluokkien) aliolioiden määrään. Virtuaalisessa periytymistavassa useaa eri perintäpolkua pitkin virtuaalisesti peritty yliluokka nähdään yhtenä alioliona. Toisin sanoen jokainen yksittäinen virtuaalisesti aliluokkaan peritty yliluokka sisältyy tähän aliluokkaan yhtenä yliluokan tyyppisenä alioliona (ISO/IEC 14882, s. 164). Toisaalta mikäli luokka perii yliluokan ei-virtuaalisesti, se saa peritystä yliluokasta aina oman aliolion. Siis aina kun luokka perii yliluokan virtuaalisesti, se voi joutua jakamaan tämän aliolion jonkin toisen luokan kanssa. Termi virtuaalisen periytymistavan sijasta voisi ollakin jakava periytyminen (Sakkinen 1992, s. 85). Seuraavassa esimerkissä luokalla `D` on siis vain yksi `A` -aliolio.

Esimerkki 3.1:

```
class A { /*...*/ };
class B : virtual public A { /*...*/ };
```

```
class C : virtual public A { /*...*/ };
class D : public B, public C { /*...*/ };
```

Vastaavanlainen tilanne ei-virtuaalisessa perinnässä kuten esimerkissä 2.1 aiheuttaa sen, että useaa perintäpolkua pitkin perityt yliluokat sisältyvät luokkaan useana erillisenä alioliona. Siten luokka D:llä olisikin siis kaksi A -tyyppistä alioliota. On huomioitava, että **virtual** -avainsanalla C++:ssa on operaatioiden yhteydessä aivan eri merkitys verrattuna periytymistapaan. Virtuaaliset operaatiot ovat syrjäytettävissä aliluokissa eli ne ovat siis monimuotoisia. Ei-virtuaaliset operaatiot ovat *yksimuotoisia* (monomorphic) ja näitä ei voi syrjäyttää, mutta samalla nimellä operaation voi *piilottaa* samoin kuin yliluokan attribuutinkin. C++:n moniperinnässä yliluokkaa ei voida periä toistuvasti eli luokkaa ei voi määritellä jonkin luokan välittömäksi yliluokaksi kuin ainoastaan kerran (ISO/IEC 14882, s. 164). Luokka voi tosin toimia välittömänä ja ei-välittömänä yliluokkana tai useana ei-välittömänä yliluokkana kuten haarautuvassa moniperinnässä.

3.1.2 Suojamääreet periytymistavassa

C++:ssa virtuaalisuuden lisäksi myös suojamääreet vaikuttavat perintään. Tämä onkin yksi kielen periytymistä eniten monimutkaistavia asioita. C++:ssa suojamääre voi olla joko julkinen (public), suojattu (protected) tai yksityinen (private). Suojamääre yhdessä periytymistavan mahdollisen virtuaalisuuden kanssa luo peräti kuusi erilaista periytymistavan varianttia. Nämä on esitelty seuraavassa taulukossa (TAULUKKO 1):

TAULUKKO 1: Mahdolliset periytymistavan yhdistelmät

periytymistapa	julkinen	suojattu	yksityinen
virtuaalinen	X	X	X
ei-virtuaalinen	X	X	X

Periytymistapa määrää tavan, jolla yliluokka eli siis tarkemmin ottaen sen piirteet peritään. Periytymistapa vaikuttaa yliluokasta perittyjen piirteiden näkyvyyteen aliluokassa. Luokan esittelyn yhteydessä luokan määrittelyssä käytetään suojamäärettä perittävän yliluokan nimen kanssa. Periytymistapa voi ainoastaan

tiukentaa piirteen suojamäärettä, ei heikentää sitä. Esimerkki kirjasta selventää tätä (Stroustrup 1997, 406). Ajatellaan kahta luokkaa, jotka ovat luokka C ja sen välittömänä yliluokkana luokka B.

Jos luokka B:n periytymistapa C:hen on:

- Yksityinen, B:n julkiset ja suojatut piirteet näkyvät vain luokka C:ssä ja sen ystävissä.
- Suojattu, B:n julkiset ja suojatut piirteet näkyvät luokka C:ssä, sen ystävissä ja aliluokissa.
- Julkinen, sen julkiset piirteet näkyvät kaikkialle ja suojatut piirteet luokka C:ssä, sen ystävissä ja aliluokissa.

Suojamääreitä oikein käyttämällä voidaan tarjota halutunlaisia palvelurajapintoja luokan asiakkaille, julkinen rajapinta kaikille ja suojattu rajapinta aliluokille. Pääsääntöisesti luokan tietosisältö eli attribuutit on pyrittävä pitämään mahdollisimman suojattuina.

C++:ssa periytymistapa monimutkaistaa jo yksittäisperintää. Jos näitä kuutta erilaista varianttia käytetään moniperinnän yhteydessä, tilanne monimutkaistuu edelleen. Teoriassa yhdellä luokalla voi moniperinnässä olla kuudella eri periytymistavalla perittyjä yliluokkia!

3.2 Ongelmakohtia

3.2.1 Yksityinen periytymistapa

Yksityistä periytymistä käytetään yleensä toiminnallisuuden perimiseen. Siinä perittävän ja perivän luokan välille ei muodostu alityyppi (*is-a)* suhdetta. Yksityinen periytymisen käyttö C++:ssa voi aiheuttaa ongelmia, jotka ilmenevät siis jo yksittäisperiytymisessä. Eräs yksityisen periytymisen ongelma tulee esille virtuaalisten operaatioiden syrjäyttämisen yhteydessä (Sakkinen 1992, s. 80). Yksityisessä periytymisessä luokka voi syrjäyttää jonkin yliluokkansa virtuaalisen operaation, jota se ei voi kutsua eli jota se ei näe. Tällaisessa tapauksessa perintäpolku

luokasta sen yliluokkaan on yksityisen periytymisen johdosta siis *läpikäymätön* (intransitive) (Sakkinen 1992, s. 77). Läpikäymättömässä periytymisessä jokainen luokka pääsee siis käsiksi korkeintaan välittömien vanhempiensa piirteisiin. Näin tällä läpikäymättömällä periytymispolulla on aina siis vähintään kolme luokkaa (solmua). Julkista tai suojattua periytymistapaa C++:ssa kutsutaan taas vastaavasti *läpikäytäväksi* (transitive). Selvitetään läpikäymätöntä tilannetta hieman alkuperäisestä muokatulla esimerkillä yksityisestä periytymisestä (Sakkinen 1992, s. 82):

Esimerkki 3.2:

```
class Perus {
public:
      virtual void m();
/*...*/
};
class Kätkijä : private Perus {
/*...*/
};
class Väli : public Kätkijä {
/*...*/
};
class Erikois : public Väli {
public:
      virtual void m();
/*...*/
};
```

Tässä tapauksessa `Erikois` -luokan `m` -operaatio syrjäyttää `Perus` -luokan vastaavan, mutta siellä näkymättömän operaation. Näin `m` -operaation kutsu on `Erikois` -luokan piiristä sallittu, tämän mahdollistaa siis operaation syrjäytys. `Erikois` -luokan `Väli` -yliluokasta ei kuitenkaan voi kutsua `Perus` -luokan `m` -operaatiota, koska sen piirteet ovat tälle yksityisiä ja `Väli` -luokassa ei myöskään syrjäytetä `m` -operaatiota. Luokkien välissä oleva luokka `Kätkijä` perii yksityisesti `Perus` -luokan ja sen piirteet muuttuvat näkymättömiksi kaikille `Kätkijä` -luokan jälkeläisille. Lisäksi `Kätkijä` -luokan operaatioista tehdyt `m` -operaation kutsut `Erikois` -tyyppiselle oliolle kohdentuvat `Erikois` -luokan `m` -operaatioon. Yksityinen periytyminen välissä ei siis eristä näitä luokkia toisiltaan (Sakkinen 1992, s. 81).

C++:n yksityistä periytymistä ei pidetäkään kielen kannalta tärkeänä tai välttämättömänä ominaisuutena verrattuna julkiseen periytymistapaan. Onpa jopa ehdotettu sen ja suojatun periytymisen poistamista kielestä (Sakkinen 1992, s.78). Oikeastaan vain julkinen periytymistapa voidaan nähdä merkityksellisenä periytymisessä käsitteellisen mallintamisen kannalta. Vain julkinen tai suojattu periytyminen on alityypitystä eli se tekee luokasta sen perityn yliluokan alityypin. Yksityistä periytymistä on tarkoitus käyttää vain ohjelmakoodin eli siis toteutuksen perintään (Stroustrup 1997, s. 405).

3.2.2 Nimikonfliktit

Moniperinnässä syntyy helposti *horisontaalisia nimikonflikteja*. Nimikonflikteja voidaankin pitää keskeisimpinä moniperinnän ongelmista (Sakkinen 1992, s. 89). C++:n ei ole mitään kätevää mekanismia moniperinnässä syntyvien nimikonfliktien korjaamiseksi perivässä luokassa. Nimikonfliktit ovat ongelmallisia eteenkin virtuaalisten operaatioiden kohdalla. Jos luokkaan periytyy useampi kuin yksi samanniminen piirre, yksinkertaisin keino *monikäsitteisyyden* (ambiguity) poistamiseksi on *luokkatarkentimen* (scope resolution operator) käyttö tätä kutsuttaessa. Kahdesta samannimisestä peritystä piirteestä on siis valittava eksplisiittisesti haluttu piirre luokkatarkentimen avulla (Stroustrup 1997, s. 394). Luokkatarkenninta käytettäessä operaatiokutsuissa näiden mahdollinen virtuaalisuus peruuntuu. Samannimisiin ei-virtuaalisiin operaatioihin tai attribuutteihin viitatessa luokkatarkentimen käytöstä ei luonnollisestikaan ole vastaavaa haittaa. Toinen mahdollisuus on käyttää *tyyppimuunnos* (type casting) operaatiota. Sen käyttö ei vie operaatiolta sen mahdollista virtuaalisuutta, koska tyyppimuunnos voidaan suorittaa dynaamisesti. Otetaan esimerkiksi seuraavanlainen tilanne riippumattomasta moniperinnästä Stroustrupin kirjasta (Stroustrup 1997, s. 779).

Esimerkki 3.3:

```
class Window {
public:
      virtual void draw();
```

```
/*...*/
};
class Cowboy {
public:
      virtual void draw();
/*...*/
};
class CW : public Cowboy, public Window {
/*...*/
};
```

Tässä luokka CW perii sekä Cowboy -luokan että Window -luokan. Jos CW -luokasta nyt haluttaisiin kutsua perittyä draw -operaatiota täytyisi tällöin käyttää C++:n luokkatarkenninta. Halutun yliluokan virtuaalista operaatiota kutsuttaisiin joko Window::draw() tai Cowboy::draw() monikäsitteisen kutsun välttämiseksi. Mutta miksi kukaan haluasi määritellä virtuaalisia operaatioita ja kutsua sitten niitä luokkatarkentimen kautta (Stroustrup 1999, s. 8)? Jos draw -operaatio haluttaisiin määritellä uudelleen CW –luokassa, se syrjäyttäisi kummankin yliluokan vastaavan samannimisen operaation, huolimatta näiden ilmeisestä toiminnallisuuden erilaisuudesta. Mikäli aliluokassa kuitenkin haluttaisiin kummankin yliluokan operaation toiminnallisuus voitaisiin CW -luokan syrjäyttävässä operaatiossa kutsua näitä molempia seuraavasti:

Esimerkki 3.4:

```
class CW : public Cowboy, public Window {
public:
      void draw()
      {
            Cowboy::draw();
            Window::draw();
      }
/*...*/
};
```

Tässä luokkatarkentimen avulla tehty operaation kutsu vie draw -operaation mahdollisen virtuaalisuuden ja mahdollisesti haittaa näin tulevaa luokkahierarkian jatkokehitystä (Sakkinen 1992, s. 89) Myös Stroustrup pitää tätä vääränä, koska

yhteisestä nimestä huolimatta operaatiot eivät liity toisiinsa ja niitä ei saisi syrjäyttää yhteisellä operaatiolla (Stroustrup 1997, s. 779). CW -luokasta voitaisiin kutsua haluttua draw -operaatiota myös käyttäen tyypinmuunnosta seuraavasti:

```
((Cowboy*)this)->draw();
```

Vaihtoehtoisesti operaatiota voitaisiin kutsua myös tällä tavalla:

```
*(&Cowboy::draw)();
```

Ensimmäisessä vaihtoehdossa **this** -olioviitettä käytetään osoittamaan Cowboy -aliolioon. Tyypinmuunnoksen käyttöä luokkahierarkiassa alaspäin eli aliluokkaan tarkentavasti olisi vältettävä, koska se ei ole aina täysin tyyppiturvallista. Edellisessä tyypinmuunnos tehdään kuitenkin ylöspäin eli täysin turvallisesti. Jälkimmäinen ratkaisuista on toiminnaltaan vastaava kuin ensimmäinen, mutta syntaksiltaan monimutkaisempi ja näin sen semantiikka voi jäädä hämäräksi. Otetaan esimerkki Stroustrupin kirjasta, jossa samannimiset operaatiot syrjäytetään uusissa luokissa (Stroustrup 1997, s. 779):

Esimerkki 3.5:

```
class Wwindow : public Window {
public:
      virtual void win_draw() = 0;
      void draw() { win_draw(); };
/*...*/
};
class Ccowboy : public Cowboy {
public:
      virtual void cow_draw() = 0;
      void draw() { cow_draw(); };
/*...*/
};
class CW : public CCowboy, public WWindow {
public:
      void win_draw();
      void cow_draw();
/*...*/
};
```

Näin apuluokat WWindow ja CCowboy jäävät abstrakteiksi. Käsitellään vielä hieman suoraviivaisempi ratkaisu samaan Stroustrupin esittämään ongelmaan samannimisten operaatioiden syrjäyttämisestä apuluokkien avulla (Sakkinen 1992, s. 92):

```
Class Wwindow : public Window {
public:
      Virtual void win_draw() { Window::draw(); };
      void draw() { /*...*/ }
/*...*/
};
class Ccowboy : public Cowboy {
public:
      virtual void cow_draw() { Cowboy::draw(); };
      void draw() { /*...*/ }
/*...*/
};
class CW : public CCowboy, public WWindow {
public:
      void win_draw();
      void cow_draw();
/*...*/
};
```

Myös tässä tapauksessa luodaan kaksi täysin uutta apuluokkaa WWindow ja CCowboy, joissa toteutetaan draw –operaatiot ja määritellään kaksi uutta virtuaalista apuoperaatiota, joista kutsutaan sitten draw -operaatioita. Sitten nämä kaksi uutta luokka peritään CW -luokkaan, jossa kaksi erinimistä virtuaalista operaatiota voidaan halutessa syrjäyttää ilman nimikonfliktia. Näin nimikonflikti olisi ratkaistu ja operaatioiden virtuaalisuus olisi samalla säilynyt. Nimikonfliktitilanteen ratkaisu on riittävä, mutta kohtuullisen suuritöinen ainakin alkuperäisen ongelman pienuuteen nähden. Edellä olevassa CW -luokassa ei ole edes pakko määritellä molempia operaatioita.

Entäpä jos uudessa ratkaisussa käytettäisiin yksityistä periytymistapaa kahteen uuteen luokkaan? Näin alkuperäiset draw -operaatiot saataisiin kätevästi piilotettua CW -luokalta, koska niitä ei ole enää tarkoitus kutsua sieltä. Valitettavasti piilottaminen ei

kuitenkaan estä niiden syrjäyttämistä `CW` –luokasta. Samalla myös periytymisen merkitys katoaisi, koska yksityinen periytymistapa estäisi alityyppisuhteen syntymisen.

Sakkinen mainitsee artikkelissaan vielä yhden mahdollisen keinon korjata edellä mainitun kaltainen nimikonflikti (Sakkinen 1992, s. 92). Tässä ehdotetussa tavassa C++-kieleen lisättäisiin uusi mekanismi *tittelit* (titles). Titteli muistuttaisi C++:n osoitinta jäsenfunktioon, jollaisia kielessä on jo käytössä. Lisätään edelliseen esimerkkiin vielä ratkaisu ehdotettuja titteleitä käyttäen:

Esimerkki 3.6:

```
class CW : public Cowboy, public Window {
public:
      virtual void Cowboy..draw();
      virtual void Window..draw();
/*...*/
};
```

Näin esiteltäisiin kaksi yliluokkien `draw` -operaatiot syrjäyttävää ja aliluokassa operaatioita eli niiden toimintaa tarkentavaa tittelillä. Näin aliluokassa määritelty `Cowboy..draw()` tarkoittaisi `Cowboy::draw()` -operaation siellä tarkentavaa ja syrjäyttävää operaatiota.

Esimerkin 3.5 ratkaisusta, joka on C++:ssa toteutettavissa voimme huomata, että luokkahierarkia monimutkaistuu huomattavasti ja siitä tulee myös helposti yllättävän sekava. Helpoin ja suoraviivaisin ratkaisu tähän – ehkä jopa hivenen keinotekoiseen – ongelmaan olisi esimerkiksi `Window` -luokan `draw` -operaation uudelleennimeäminen sen perivässä luokassa vaikkapa nimellä `refresh`. C++:ssa perityn piirteen varsinainen uudelleennimeäminen ei ole mahdollista, mutta epäsuorasti luokkahierarkiaa kasvattamalla edellä näytetyllä tavalla voidaan luoda keinotekoinen ratkaisu tilanteeseen. C++:ssa ei voi myöskään millään – ei siis edes yksityisellä periytymistavalla – estää perittyjen virtuaalisten `draw` -operaatioiden syrjäyttämistä. `Ccowboy` ja `Wwindow` -luokissa, jotka perivät vanhat `draw` -operaatiot pitäisi pystyä estämään aliluokilta niiden syrjäytys tai käyttö. Olisihan mahdollisten jälkeläisten

nimeen omaan tarkoitus käyttää Ccowboy ja Wwindow -luokissa määriteltyjä uusia operaatioita. Toisaalta, jos ohjelmiston lähdekoodit olisivat käytettävissä ja muokattavissa, konfliktissa olevia nimiä voitaisiin muuttaa, mutta tällöin ohjelmiston yhteensopivuus kärsisi huomattavasti näistä nimimuutoksista. Kyseistä periytymiseen perustuvaa uudelleennimeämistä käsitellään myös Ellisin ja Stroustrupin kirjassa (Ellis 1997, s. 237). Kirjassa esitellyssä ratkaisussa C++:n syntaksia jouduttaisiin hieman laajentamaan. Tässä esimerkissä nimet on muutettu vastaamaan edellistä esimerkkiä:

Esimerkki 3.7:

```
class CW : public CCowboy, public WWindow {
public:
      virtual void win_draw() = Wwindow::draw;
      virtual void cow_draw() = Ccowboy::draw;
/*...*/
};
```

Tätä siistiä ja hyvin C++:n syntaksityylin mukaista ratkaisua Stroustrup ei kuitenkaan pidä yhtä hyvänä kuin esimerkin 3.5 ratkaisua, joka on itse asiassa paljon monimutkaisempi ja työläämpi. Jokaiselle syrjäytettävälle virtuaaliselle operaatiolle on siis luotava oma luokkansa, jossa nämä sitten voidaan nimetä uudelleen. Edellisestä esimerkistä jää myös hivenen epäselväksi, onko CW -luokassa tarkoitus tehdä ainoastaan operaatioiden uudelleennimeäminen vaiko myös näiden syrjäytys. Lisäksi nimikonfliktissa olevien operaatioiden kutsumuotoihin ei oteta kantaa, edellisessähän on esitelty ainoastaan yliluokkien nimikonfliktissa olevien piirteiden nimet. Entäpä jos kummassakin yliluokassa olisi esitelty useita draw -operaatioita, joilla olisi eri kutsumuodot eli siis erilaiset parametrilistat?

3.2.3 Haarautuva moniperintä

Haarautuvaa moniperintää voidaan pitää moniperinnän hankalimpana tapauksena. C++:n haarautuvassa moniperinnässä voi nimikonfliktien lisäksi tulla esille muita ongelmia. Esimerkiksi sekä julkisen että yksityisen periytymistavan käyttäminen yhtäaikaa luokkien välillä voi aiheuttaa ongelmallisia tilanteita. Jokin nimi (piirre) voi

olla saavutettavissa tiettyä periytymispolkua pitkin, mutta samalla se ei ole saavutettavissa toista näiden luokkien välistä polkua pitkin. C++:ssa saavutettavuus on määritelty seuraavasti: jos tietty nimi tai yliluokka on usean eri polun päässä moniperinnässä se on saavutettavissa, mikäli se on saavutettavissa jonkin polun kautta (Stroustrup 1997, s. 406). Jos nimi samasta yliluokasta on taas saavutettavissa useaa eri periytymispolkua pitkin, syntyy riski nimikonflikteihin, mikäli yliluokkaa ei ole peritty virtuaalisesti. Näin siis myös mahdollinen periytymistavan virtuaalisuus vaikuttaa nimikonfliktien lisääntymiseen haarautuvassa moniperinnässä.

Virtuaalinen periytyminen haarautuvassa moniperinnässä voi myös aiheuttaa ongelmia. Mikäli yliluokka peritään virtuaalisesti ja sen virtuaalisia operaatioita syrjäytetään eri periytymispolulla, voi operaatioita kutsuttaessa syntyä yllättäviä tilanteita. Otetaan esimerkiksi seuraavanlainen tilanne haarautuvasta perinnästä (Stroustrup 1999, s. 12):

Esimerkki 3.8:

```
class W {
public:
      virtual void f();
      virtual void g();
      virtual void h();
      virtual void k();
};
class AW : public virtual W { void g(); };
class BW : public virtual W { void f(); };
class CW : public AW , public BW { void h(); };

CW* pcw = new CW;
pcw->f(); // BW::f();
pcw->g(); // AW::g();
pcw->h(); // CW::h();

((AW*)pcw)->f(); // BW::f();
```

Esimerkissä on siis yksinkertaisin haarautuvan moniperinnän tilanne. Luokka CW perii suoraan kaksi luokkaa ja näiden kautta jaetun W -luokan. Esimerkissä kaikki menee hyvin, mutta entäpä jos myös BW -luokassa syrjäytettäisiin g -operaatio? Tällöin

operaation kutsu osoittimen kautta `pcw->g()` ei olisi enää yksiselitteinen ja ratkaisuksi jouduttaisiin tekemään uusi syrjäytys `CW` -luokassa monikäsitteisyyden poistamiseksi. Virtuaalisten operaatioiden dynaamisen sidonnan mukaisesti kohdeolion tyyppi määrää kutsuttavan operaation. Näin vaikka esimerkissä kutsutaan `f` -operaatiota `AW` -tyyppisen osoittimen kautta, kutsu menee silti toisella periytymispolulla olevalle `BW` -luokassa olevalle syrjäytetylle `f` -operaatiolle, eikä `W` -luokkaan saakka. Virtuaalisten yliluokkien virtuaalisten operaatioiden syrjäyttämisessä ja kutsumisessa päteekin seuraava hiukan alkuperäisestä muodostaan (Ellis 1997, s. 235) korjattu sääntö (Sakkinen 1992, s. 101):

> *Jos virtuaalisen yliluokan virtuaalinen operaatio on syrjäytetty useammalla kuin yhdellä periytymispolulla läpi periytymishierarkian, (hierarkiassa) on oltava yksi muita hallitseva syrjäytys.*
>
> *if a virtual function from a virtual base class is redefined on more than one path through the inheritance structure, there must be one redefinition that dominates all others.*

C++:n referenssiteoksessa todetaankin, että virtuaalisia yliluokkia käytettäessä useampi kuin yksi operaatio – tai muu nimi – voi olla saavutettavissa periytymispolkuja pitkin. Tällaisessa tilanteessa monikäsitteisyyden välttämiseksi yhden saavutettavista samoista nimistä on oltava hallitseva muihin nähden. (Ellis 1997, s. 204). Tämä *hallitsevuussääntö* (dominance rule) onkin välttämätön virtuaalisille operaatioille, koska se määrää mitä niistä kutsutaan dynaamisesti. On huomioitava, että C++:ssa yliluokasta piirrettä kutsuttaessa mahdollinen monikäsitteisyys tarkistetaan ennen perittyjen piirteiden suojamääreiden tarkistusta (Ellis 1997, s. 202). Näin jopa (näkymätön) yksityinen piirre voi aiheuttaa sen, että julkisen samannimisen piirteen kutsusta tuleekin – ehkä hieman yllättäen – monikäsitteinen.

Myös yksityisen periytymistavan käyttö yhdessä julkisen tai suojatun periytymisen kanssa samassa haarautuvassa moniperinnässä voi johtaa hankaliin tilanteisiin. Tällöin on mahdollista syrjäyttää tiettyä periytymispolkua pitkin saavuttamattomia

piirteitä. Otetaan esimerkki kahden periytymistavan yhdistelmästä (Sakkinen 1992, s. 99):

Esimerkki 3.9:

```
class A { /*...*/ };
class B : public virtual A { /*...*/ };
class C : private B, public virtual A { /*...*/ };
class D : public virtual C { /*...*/ };
```

Tässä luokat A ja C ovat saavutettavissa D -luokasta. Luokka B on taas saavutettavissa C -luokasta, mutta ei enää D -luokasta. D -luokka voi kuitenkin syrjäyttää A -luokasta perittyjä virtuaalisia operaatioita ja nämä ovat myös voimassa julkisesti perityssä C -luokassa. Koska B -luokka on saavutettavissa C -luokasta, vaikuttavat muutokset myös D -luokasta saavuttamattomissa olevaan B -luokkaan (Sakkinen 1992, s. 99).

Haarautuvassa moniperinnässä käsitteellisen mallintamisen kannalta ei-virtuaalinen – siis oman aliolion – periminen ei ole läheskään niin merkityksellistä kun virtuaalinen eli jakava perintä. Tuskin voidaan kuvitella minkään luokan olevan jotain yliluokka -tyyppiä useasti (toistuva *is-a* –suhde luokkien välillä). Sen sijaan ohjelmistokoodin uudelleenkäytön kannalta usean aliolion periminen voi olla jossain tilanteissa hyödyllistä. Otetaan esimerkiksi esimerkki Stroustrupin kirjasta, jossa käsitelty tilanne on ei-virtuaalinen haarautuva moniperintä (Stroustrup 1997, s. 394). Esimerkin `Satellite` -luokalla olisi kaksi `Link` -alioliota. Tässä tilanteessa käsitteellisen mallintamisen kannalta `Link` -luokka voitaisiin periä virtuaalisesti, tällöin `Satellite` -luokalla olisi vain yksi `link` -tyyppinen aliolio ja näin *is-a* suhdekin toteutuisi. Tässä esimerkissä onkin lähdetty siitä, että `link` -tyyppistä alioliota on toteutuksen kannalta hyvä käyttää kummassakin sen perineessä luokassa. Esimerkkiään Stroustrup kutsuu kirjassaan yksinkertaisimmaksi ja ilmeisimmäksi moniperinnän käytöksi, missä luokkaan liitetään yhteen ja käytetään hyväksi kahden toisiinsa millään lailla liittymättömän yliluokan toiminnallisuutta (Stroustrup 1997, s. 399).

3.3 Korjausehdotuksia

C++:n moniperintää voisi selkiyttää ja parantaa monella tapaa. Joitakin parannusehdotuksia C++:n perintää koskien onkin jo esitetty (Sakkinen 1992). C++:n moniperinnän pahimpina ongelmina voidaan pitää pääasiallisesti seuraavia: horisontaalisia nimikonflikteja, haarautuvan moniperinnän monimutkaisuutta sekä vääriä periytymistavan yhdistelmiä. Haarautuvassa moniperinnässä voi ilmetä kaikki edellä mainitut ongelmat, mutta riippumattoman moniperinnän pääasiallinen ongelma on nimikonfliktit. C++:n haarautuvaa moniperintää voisi rajoittaa tai ainakin sen hallintaa olisi parannettava. Sen voisi jopa kieltää kokonaan niin, että kieli sallisi vain riippumattoman – siis yksinkertaisemman – moniperinnän muodon. Näin huomattava osa tässäkin luvussa esitetyistä C++:n ongelmista olisi ratkaistu. Tosin horisontaalisia nimikonflikteja syntyisi yhä ja myös osa yksityisen periytymisen tuomista ongelmista säilyisi kielessä.

Myös C++:n periytymistapaa olisi hyvä korjata. Yksityisen periytymistavan voisi poistaa tai sitä voisi rajoittaa jonkin verran. Esimerkiksi niin, ettei virtuaalisia operaatioita voisi tehdä näkymättömiksi tai saavuttamattomia virtuaalisia operaatioita ei saisi syrjäyttää (Sakkinen 1992, s. 81). Yksityisten virtuaalisten operaatioiden käyttötarve on yleensäkin erittäin harvinaisia (Sakkinen 1992, s. 83). Yksi hyvä mahdollisuus olisi tehdä haarautuvasta jakavasta eli virtuaalisesta moniperinnästä implisiittisesti julkista ja näin ainoastaan ei-virtuaalinen haarautuva moniperintä voisi olla yksityistä. Tällöin ei pääsisi syntymään tilanteita, missä yliluokka olisi jotain periytymispolkua pitkin saavutettavissa, mutta toista periytymispolkua pitkin ei. Sallitut periytymistavan yhdistelmät olisivat kuten seuraavassa taulukossa (TAULUKKO 2) on esitetty:

TAULUKKO 2: Sallitut periytymistavan yhdistelmät

periytymistapa	julkinen	suojattu	yksityinen
virtuaalinen	X	X	
ei-virtuaalinen			X

Tällöin haarautuvassa moniperinnässä aliolioiden lukumäärä eli niiden mahdollinen jakaminen määräytyisi saavutettavuuden mukaan. Julkisessa periytymistavassa jaettaisiin samat alioliot ja yksityisessä periytymisessä luokka perisi aina oman aliolion (Sakkinen 1992, s. 98). Näin periytymistapa olisi myös paremmin suhteessa käsitteellisen mallintamisen kanssa. Julkisessa periytymisessä muodostuu alityyppisuhde ja tällöin luokalla olisikin vain yksi yliluokka -tyyppinen aliolio, luokkahan ei voi yleensä olla mitään tyyppiä useasti. Vastaavasti yksityisessä periytymisessä, jossa on kyse pääosin toiminnallisuuden eli ohjelmakoodin perinnästä, jokaista perittyä yliluokkaa vastaisi aina oma jakamaton aliolio. Näin esimerkiksi edellä käsitellyn `Satellite` -luokkaa käsittelevän esimerkin tilanne korjaantuisi paremmaksi.

Myös nimikonfliktien välttämiseksi C++:ssa olisi hyvä olla jokin mekanismi, esimerkiksi piirteiden uudelleennimeäminen. C++:ssa ei myöskään ole mitään mekanismia, jolla voisi estää luokan käyttöä perinnässä. Tällainen mahdollisuushan on joissain muissa oliokielissä. Myös tällaisella ominaisuudella olisi mahdollista rajoittaa periytymistä ja siten myös perinnän turhaa käyttöä.

3.4 Yhteenveto

Tässä luvussa käsittelimme moniperintää C++:ssa. Ensimmäisenä esittelimme periytymistavan, jolla C++:ssa vaikutetaan eniten perintään luokkien välillä. Periytymistapa voi olla julkinen, suojattu tai yksityinen sekä myös mahdollisesti virtuaalinen. Periytymistavassa suojamääreet vaikuttavat piirteiden näkyvyyteen ja mahdollinen virtuaalisuus voi johtaa aliolioiden jakamiseen. Seuraavaksi käsittelimme joitakin keskeisimpiä C++:n moniperinnän ongelmia, joita ovat muiden muassa yksityinen periytymistapa, nimikonfliktit ja liiallinen haarautuva periytyminen. Lopuksi käsittelimme korjausehdotuksia, joilla C++:n moniperintää voisi selkiyttää ja hallita. Erityisesti C++:n perintään liittyvää periytymistapaa olisi korjattava, sillä tällä hetkellä se jättää liian paljon vapaita mahdollisuuksia käyttäjälleen. Periytymistavan suojamääreet voisivat olla sidoksissa mahdolliseen virtuaalisuuteen: yksityisen periytymisen olisi oltava aina ei-virtuaalista ja julkinen sekä suojattu periytyminen olisi aina virtuaalista. Näitä selkeyttäviä sääntöjä onkin jo ehdotettu alan kirjallisuudessa (Sakkinen 1992).

4 EIFFELIN MONIPERINTÄ

Tässä luvussa tulemme käsittelemään yleisesti perintää ja tarkemmin moniperintää Eiffel-ohjelmointikielessä. Erityisesti tulemme paneutumaan moniperinnän käyttömahdollisuuksiin ja käsittelemme myös siinä olevia rajoitteita ja puutteita. Tulemme käsittelemään myös potentiaalisia ongelmia Eiffelin moniperinnässä ja tämän käytössä. Lopuksi tarjoamme joitakin yleisiä parannusehdotuksia kielen moniperinnän käyttöön tai jopa sen toteutukseen.

4.1 Yleistä

4.1.1 Perintälauseke

Eiffelin luokkien välinen perintätapa määräytyy *perintälausekkeessa* (inheritance clause) (Meyer 1997, s. 462). Aivan kuten C++:ssa luokan esittelyn yhteydessä, Eiffelin perintälausekkeessa luetellaan perittävät yliluokat. Varsinaisia suojamääreitä ei ole ja näin yliluokkien perintä on periaatteessa aina julkista eli siis näkyvää (Meyer 1997, s. 548). Luokkien piirteet voidaan tehdä eksplisiittisesti julkisiksi esittelemällä ne seuraavasti: `feature {ANY}`, mikä onkin Eiffelissä oletusarvo. Piirteitä voidaan myös piilottaa muilta luokilta eli asiakasluokilta, mutta ei kuitenkaan koskaan luokan jälkeläisluokilta. Tästä seuraakin se, ettei Eiffelissä ole koskaan mahdollista luoda periytymishierarkiaan läpikäymätöntä periytymispolkua luokkien välille. Piirteet voidaan piilottaa kaikilta asiakkailta esittelemällä piirre seuraavasti: `feature {}` tai `feature {NONE}`. Näin esiteltynä piirteestä tulee suojattu. Piirteet voidaan myös tehdä julkisiksi halutuille asiakasluokille luettelemalla näiden nimet aaltosulkujen sisällä seuraavasti: `feature {A,B,C}`. Piirteiden suojaus Eiffelissä on oliokohtainen eli suojattu piirre näkyy vain oliossa itsessään ja yliolioissa (aliluokissa). Luokkakohtainen suojaus on mahdollista luoda luokalle käyttämällä sen nimeä piirteen esittelyssä, esimerkiksi luokassa `E` seuraavasti: `feature {E}`. Tällä tavoin siis saadaan aikaan piirteen luokkakohtainen suojaus oliokohtaisen sijaan `E` -luokassa. Perintälausekkeessa luokka voi lisäksi piilottaa yliluokilta perittyjä piirteitä omilta asiakasluokiltaan käyttäen `export` -avainsanaa.

Haarautuvassa moniperinnässä jaetaan aina mahdolliset useasti perityt vanhemmat, joten luokalle ei tule alioliomallin mukaisesti useita samantyyppisiä aliolioita. Toistuva perintä on myös mahdollista ja tähän pätee sama jakamisperiaate: luokka voi periä toisen luokan suoraan useaan kertaan eli luokalla voi olla sama luokka useana välittömänä vanhempanaan – tai välittömänä ja jonkin luokan kautta perittynä välillisenä yliluokkana – ja silti luokalle tulee tätä yliluokkaa vastaamaan vain yksi aliolio.

Eiffelin perintälausekkeessa voidaan myös luetella perittyjä piirteitä, joita halutaan uudelleennimetä tai uudelleenmääritellä eli syrjäyttää kyseisessä luokassa. Otetaan vielä edellisen luvun esimerkissä 3.5 käsitelty tilanne riippumattomasta moniperinnästä, jossa syntyy nimikonflikti. Tässä esimerkissä perittiin kaksi samannimistä, mutta merkitykseltään hyvinkin erilaista piirrettä luokkaan kahdelta vanhemmalta.

Esimerkki 4.1:

```
class COWBOY
      feature draw
... end

class WINDOW
      feature draw
... end

class CW
      inherit
      COWBOY
            rename draw as cow_draw
            redefine cow_draw
      ... end
      WINDOW
            rename draw as win_draw
            redefine win_draw
      ... end
... end
```

Näin perityt nimikonfliktissa olevat `draw` -operaatiot nimetään uudelleen edellisen luvun esimerkin lailla. Tämän jälkeen uudet operaatiot voidaan myös syrjäyttää

luokassa. Tämä voisi olla myös operaation viivästyttämistä, joka saadaan aikaan `undefine` -avainsanalla. Näin konkreetista yliluokan operaatiosta tehdään abstrakti ja näin myös luokasta itsestään tulee abstrakti. Viivästyttämistä ei luonnollisestikaan voi tehdä attribuuteille. Nimikonfliktien välttämiseksi perintälausekkeessa voidaan käyttää vielä valintalausetta `select` -avainsanalla, jolla valitaan haluttu piirre perityistä erinimisistä piirteistä. Tätä Eiffelin hieman erikoisempaa ominaisuutta käsittelemme vielä myöhemmin tässä luvussa. Edellä esitetty Eiffelin ratkaisu kahden nimen nimikonfliktitilanteeseen on kaiken kaikkiaan hyvin yksinkertainen ja siisti.

4.1.2 Nimet

Kaikilla luokkien piirteillä Eiffelissä on *viimeinen nimi* (final name). Piirteen viimeinen nimi luokassa määräytyy seuraavasti (Meyer 1997, s. 549):

- Luokassa esitellyille (välittömille) piirteille se on piirteen nimi.
- Perityille uudelleennimeämättömille piirteille se on piirteen nimi yliluokassa, josta se on peritty (piirre voi tietenkin olla uudelleen nimetty yliluokassa).
- Uudelleen nimetylle peritylle piirteelle viimeinen nimi on sen uusi nimi luokassa.

Viimeisten nimien välillä luokassa ei saa olla nimikonfliktia. Nimikonflikti syntyy, jos luokalla on kaksi *voimassaolevaa* (effective) eri piirrettä, joilla kummallakin on samat nimet. Eri piirteillä tarkoitetaan sitä, että samasta esivanhemmasta toistuvasti peritty samanniminen piirre – siis haarautuvassa moniperinnässä – ei aiheuta nimikonfliktia. Tällöin on kyse siis samasta piirteestä ja piirteet samaistetaan (Meyer 1997, s. 550). Nimikonflikti luokassa kahden viimeisen nimen välillä sallitaan ainoastaan seuraavissa tapauksissa (Meyer 1997, s. 562):

- Kumpikin samanniminen piirre on peritty yhteisestä vanhemmasta ja kumpaakaan näistä ei ole syrjäytetty.
- Perityt piirteet omaavat samanlaiset kutsumuodot ja korkeintaan yksi niistä on voimassaoleva.

- Perityt piirteet omaavat samanlaiset kutsumuodot ja ne kumpikin syrjäytetään luokassa.

Viimeisessä tapauksessa kaksi perittyä eri piirrettä siis yhdistetään yhdeksi ainoaksi piirteeksi luokassa.

4.1.3 Syrjäytys

Operaatiot – joita Eiffelissä kutsutaan *rutiineiksi* (routine) – ovat automaattisesti virtuaalisia. Rutiinit voidaan Eiffelin terminologiassa jakaa vielä *funktioihin* (function), joilla on jokin paluuarvo ja *proseduureihin* (procedure), joilla ei ole paluuarvoa. Kutsumme näitä kaikkia yksinkertaisuuden vuoksi yleisesti operaatioksi. Parametriton operaatio eli Eiffelissä funktio, joka palauttaa tietyn tyypin, voidaan samaistaa samannimiseen – ja operaation palautustyypin kanssa saman tyyppiseen – attribuuttiin. Näin aliluokassa on mahdollista syrjäyttää yliluokan operaatio oikean nimisellä ja tyyppisellä attribuutilla (Meyer 1997, s. 491). Päinvastainen ei ole mahdollista eli yliluokan attribuuttia ei voida syrjäyttää aliluokassa operaatiolla. Kaikkien luokassa syrjäytettävien piirteiden nimet on myös lueteltava luokan perintälausekkeessa. Kaikkien operaatiokutsujen ollessa automaattisesti virtuaalisia on hyvä olla olemassa mekanismi, jolla voidaan estää syrjäytys tarvittaessa. Tämä tapahtuu *jäädyttämällä* piirre (frozen). Jos luokan piirre on jäädytetty, sitä ei voi syrjäyttää (Meyer 1997, s. 583). Viivästettyä piirrettä ei myöskään voida jäädyttää. Jos aliluokan syrjäyttävässä operaatiossa halutaan kutsua yliluokan alkuperäistä operaatiota voidaan käyttää `Precursor` -avainsanaa, jotta operaation kutsu saadaan menemään alkuperäiseen operaatioon. Tämän avainsanan käyttö on sallittua ainoastaan operaation uudelleenmäärittelevän operaation sisällä. Sen yhteydessä on myös mahdollista käyttää halutun yliluokan nimeä, jos syrjäytetään useita samannimisiä yliluokkien operaatioita (Meyer 1997, s. 494). Piirteen syrjäytyksessä pätee Eiffelin *kovarianttisääntö* (covariance rule):

> *Piirteen syrjäytyksessä mahdollisen paluuarvon ja operaation parametrien tyyppien täytyy olla yhteensopivia alkuperäisen syrjäytettävän piirteen tyyppien kanssa.*

> *In a feature redeclaration, both the result type if the argument is a query (attribute or function) and the type of any argument if it is a routine (procedure or function) must conform to the original type as declared in the precursor version.*

Näin siis vain kovariantilla piirteellä voidaan syrjäyttää, muutoin tilanne ei ole sallittu. Operaatioiden kuormitus ei ole mahdollista Eiffelissä. Tyyppien sopivuus tarkoittaa sitä, että tietty luokka D on sopiva luokan C kanssa, mikäli luokka D on luokka C:n jälkeläinen eli jokin aliluokka, luokat itse mukaan lukien.

4.2 Ongelmakohtia

Eiffelin moniperintä on monimutkainen ja laaja kokonaisuus. Moniperinnän hallitsemiseen on tarjottu monipuolisia mekanismeja, mutta näillä on myös mahdollista saada aikaan ylimääräistä monimutkaisuutta ja hankaliakin tilanteita. Näitä käsittelemme tarkemmin seuraavissa kohdissa.

4.2.1 Uudelleennimeäminen

Yksi Eiffel-ohjelmointikielen moniperinnän hallintaan tarjoama väline on piirteiden uudelleennimeäminen. Uudelleennimeämisellä, niin kuin myös valinnalla ja piirteiden viivästyttämisellä on mahdollista poistaa nimikonflikteja, mutta myös saada aikaan ylimääräistä monimutkaisuutta perintähierarkiaan. Monimutkaisuus kasvaa yllättävän helposti varsinkin, jos perintähierarkia on kooltaan suuri. Tällöin alkuperäinen piirre voidaan esimerkiksi nimetä uudelleen useita kertoja jollain perintäpolulla luokasta toiseen ja samaan aikaan sama piirre voidaan periä jotain toista kautta täysin eri nimellä. Näin alkuperäisen piirteen kutsumuoto voi vaihdella luokasta toiseen ja tästä seuraa ylimääräistä sekavuutta sekä monimutkaisuutta. Lisäksi tämä voi johtaa tilanteisiin, jotka joudutaan ratkaisemaan valintamekanismilla. Tästä valinnan käytöstä käsitellään esimerkki seuraavassa kohdassa. Uudelleennimeämisen pahin ongelma on kuitenkin ehkä sen rajoittamattomuus: Eiffelissä ei ole minkäänlaisia rajoitteita tai sääntöjä uudelleennimeämiselle, eikä uudelleennimeämisen kohteina

oleville piirteille. Otetaan seuraavaksi suhteellisen yksinkertainen esimerkki, jossa uudelleennimeäminen saa aikaan jo lievän ongelman:

Esimerkki 4.2:

```
class A
      feature f
... end

class B
      feature f
... end

class C
      inherit A end
      inherit B rename f as g end
... end

class E
      feature g
... end

class F
      inherit C end
      inherit E rename g as h end
-- Tässä luokassa f on g ja g onkin h...
... end
```

Näin F -luokassa joudutaan nimeämään E -luokasta peritty g –piirre, koska C -luokan kautta peritty luokkahierarkia sisältää samannimisen piirteen. Tämä johtuu C -luokan uudelleennimeämisestä, joka vaikuttaa sen jälkeläisiin. Jos C -luokassa oleva uudelleen nimetty nimi muutettaisiin tai, jos kumpikin C -luokkaan peritty piirre syrjäytettäisiin (kumpikin yhdellä operaatiolla), niin tämä poistaisi nimikonfliktin ja F -luokkaan jäisi tällöin turha piirteen nimeäminen turhaan monimutkaistamaan perintähierarkiaa. Tässä tapauksessa F -luokan jälkeläiset perisivät yliluokkiensa piirteet hyvin harhaanjohtavilla nimillä.

4.2.2 Piirteiden valinta

Piirteen valinnalla on tarkoitus ratkoa moniperinnässä syntyviä nimikonflikteja. Näin *valintasääntö* (select rule) on määritelty kirjassa (Meyer 1997, s. 555):

> *Luokan joka perii toistuvalta yliluokalta useamman kuin yhden voimassa olevan eri version samasta piirteestä eikä itse syrjäytä piirteitä, on näistä piirteistä valittava yksi käyttäen valintalausetta.*
>
> *A class that inherits two or more different effective versions of a feature from a repeated ancestor, and does not redefine both, must include exactly on of them in a select clause.*

Eri versio tarkoittaa edellisessä sitä, että jokin toistuvasti peritty piirre on syrjäytetty tai uudelleennimetty jossakin yliluokassa. Näin näitä toistuvan yliluokan piirteitä ei voida enää samaistaa – syrjäytetty piirre on luonnollisesti eri piirre kuin alkuperäinen – ja uudelleennimeäminen aiheuttaa piirteen *monistumisen* (replicate). Käsitellään seuraavaksi yksi valinnan käytön esimerkkitapaus Meyerin kirjasta (Meyer 1997, s. 554):

Esimerkki 4.3:

```
class A
      feature f
... end

class B inherit
      A
      rename f as bf
      redefine bf end
... end

class C inherit
      A
... end

class D inherit
      B
```

```
        Select bf end
        C
        end
... end
```

Mutta D -luokka voitaisiin esitellä myös seuraavalla tavalla valiten C -yliluokasta f -operaatio:

```
class D inherit
        B
        end
        C
        Select f end
... end
```

Tässä D -luokka perii A -luokan piirteen f toistuvasti sekä B että C -luokkien kautta. B -luokassa tapahtuvan uudelleennimeämisen ja syrjäytyksen takia piirteitä ei samaistetakaan D -luokassa samaksi, vaan siellä on kaksi perittyä erinimistä piirrettä. Tässä tapauksessa peritty piirre siis monistuu eli replikoituu. Näin uudelleennimeämisen ansiosta D -luokassa vältetään nimikonflikti. Meyer toteaa myös, että sama lopputulos syntyisi jos B -luokka ei nimeäisikään uudelleen piirrettä, mutta sen sijaan se tehtäisiin D -luokassa (Meyer 1997, s. 554). Tämä pitääkin luonnollisesti paikkansa. Tässä tapauksessa käytetään valintaa, jolla haluttu piirre valitaan D -luokkaan.

Meyer toteaa, että dynaamisesti A -tyyppisen viitteen kautta D -tyyppiselle oliolle kutsuttaessa f -piirrettä ei voitaisi päätellä, kumpaan piirteeseen kutsun pitäisi mennä: perittyyn f -operaatioon A -luokassa vai B -luokan syrjäyttämään, jonka *paikallinen nimi* (local name) on bf (Meyer 1997, s. 554). Tämä aiheuttaa valinnan käytön pakollisuuden kyseisessä tilanteessa. Myös Joyner toteaa kirjassaan tämän valinnan käytön pakon: kun peritään kaksi samannimistä piirrettä, joista toinen on syrjäytetty ja nimetty uudelleen *staattisen monikäsitteisyyden* (static ambiguity) poistamiseksi, jäljelle jää vielä *dynaaminen monikäsitteisyys* (dynamic ambiguity). Tämä on ratkaistava perivässä luokassa käyttäen valintaa (Joyner 1999, s. 112). On huomattava vielä, ettei valinta koske ainoastaan operaatioita vaan myös attribuuteille voidaan joutua tekemään valinta.

Jos esimerkin tilannetta muutetaan niin, että B -luokassa ei ole uudelleennimetty f -operaatiota, mutta se on edelleen syrjäytetty, tilanne muuttuu tavanomaisemmaksi nimikonfliktiksi. Ilman uudelleennimeämistä toistuvan yliluokan A -piirre, joka on syrjäytetty B -yliluokassa luo kaksi samaa viimeistä nimeä D -luokkaan ja näin syntyy nimikonfliktitilanne. Tällaisessa tilanteessa valintamekanismia ei tarvitse käyttää, jos käytetään syrjäytystä. Valintamekanismin käytön pakko syntyykin edellä mainitusta syrjäytyksen ja uudelleennimeämisen myötä syntyvästä dynaamisesta monikäsitteisyydestä. Kahta eri kautta peritty esivanhemman piirre monistuu uudelleennimeämisestä johtuen ja valinnalla on valittava haluttu piirre (Meyer 1997, s. 552).

Yksi mahdollinen tapa poistaa edellä mainitun kaltainen tilanne olisi piirteen viivästys. Viivästys ei tosin ole käyttökelpoinen ratkaisuvaihtoehto läheskään kaikissa tilanteissa, tuleehan myös viivästetyn piirteen omaavasta luokasta tällöin automaattisesti abstrakti.

4.2.3 Kovariantti syrjäytystapa

Eiffelissä virtuaalisen operaation voi syrjäyttää myös jos syrjäyttävä operaatio on kovariantti syrjäytettävän kanssa. Tämä periaate yhdessä muuttujien monimuotoisuuden eli polymorfismin kanssa voi aiheuttaa ongelmallisia tilanteita. Tällöin on mahdollista saada aikaan tilanne, jossa dynaamisesti kutsutaan kovarianttia operaatiota jossa parametrien tyypit eivät kuitenkaan täsmää kohdeoliossa oikealla tavalla. Otetaan tästä esimerkki Joynerin kirjasta (Joyner 1999, s. 235):

Esimerkki 4.4:

```
class A
feature
      r (p: I) is
      end
... end

class B inherit A
```

```
        redefine r end
feature
        r (p: J) is
        end
... end

class C inherit A
        redefine r end
feature
        r (p: K) is
        end
... end
```

Lisäksi oletamme vielä, että operaatioiden parametreina käytetyt luokat `J` ja `K` ovat `I` -luokasta perittyjä. Sekä luokan `B` että luokan `C` `r` -operaation syrjäytykset ovat siis täysin kovariantteja, koska `J` ja `K` ovat `I` -luokan jälkeläisiä. Tässä tapauksessa syntyy ongelmia, jos esimerkiksi `A` -tyyppisen viitteen kautta käsitellään `B`- tai `C` -tyyppistä oliota ja kutsutaan tälle `r` -operaatiota, jolle annetaan parametriksi vastaavasti joko `k`- tai `j` -tyyppinen olio eli siis:

- operaatiokutsu `a.r(k)` aiheuttaa mahdollisen virheen, jos `a` viittaa `b` -tyyppiseen olioon.
- vastaavasti operaatiokutsu `a.r(j)` aiheuttaa mahdollisen virheen, jos `a` viittaa `c` -tyyppiseen olioon.

`B` -luokassa kovariantti `r` -operaatio ei voi käsitellä `k` -tyyppistä parametria, eikä vastaavasti `C` -luokassa oleva `r` -operaatio voi käsitellä `j` -tyyppistä parametria. Meyer käsittelee tätä ongelmaa kirjassaan ja on nimennyt tämän *virhekutsuksi* (catcall). Englanninkielinen nimi tulee lyhenteestä CAT (changing availability or type) (Meyer 1997, s. 638). Operaatiota voidaan pitää tällaisena, jos jokin syrjäytys muuttaa sen parametrien tyyppiä. Tällaisia ovat siis kaikki kovariantit syrjäytykset, jotka eivät ole operaation kutsumuodoltaan täysin identtisiä. Edellisestä on hyvä mainita, että Eiffelin kovariantti syrjäytysmekanismi aiheuttaa ongelmia jo yksittäisperinnässä.

4.3 Korjausehdotuksia

Kuten myös aikaisemmin käsittelemämme C++:n, myös Eiffelin moniperintä on suhteellisen monimutkainen kokonaisuus. Monimutkaisuutta lisää esimerkiksi vapaa uudelleennimeämisen mahdollisuus varsinkin laajemmissa luokkahierarkioissa. Tätä olisikin hyvä rajoittaa jollain tavalla, vaikkapa sallimalla se ainoastaan tiettyjen sääntöjen puitteissa. Uudelleennimeäminen voisi olla sallittua esimerkiksi vain nimikonfliktitilanteessa konfliktissa oleville piirteille. Uudet nimet voisi johtaa automaattisesti konfliktissa olevien piirteiden ja näiden luokkien nimistä, jotta vältyttäisiin täysin vapaan nimeämiskäytännön tuomista mahdollisista ongelmista. Jos esimerkiksi luokka `C` on perinyt kaksi luokkaa, `A` ja `B` -luokan ja näissä kummassakin on `p` -piirre, niin nämä nimettäisiin automaattisesti nimikonfliktitilanteessa `C` -luokassa `a_p` ja `b_p` nimisiksi piirteiksi. Tällaista mekanismia onkin käytetty mm. joissakin tietokantakielissä. Jos luokka `C` olisikin perinyt toistuvasti `A` -luokan, tällöin sen `p` -piirteet samaistetaan ja näin niiden välillä ei ole nimikonfliktia eikä niiden uudelleennimeämistäkään tarvita. Jos luokka `C` olisi perinyt samanaikaisesti vielä `D` -luokan, jossa olisi myös `a_p` niminen piirre tämä nimettäisiin `C` -luokassa automaattisesti `d_a_p` nimiseksi. Piirteiden uudelleennimeämisen mahdollisuus yhdessä syrjäyttämisen kanssa aiheuttaa myös tarpeen valintamekanismille. Tämä lisää myös osaltaan monimutkaisuutta perintähierarkiaan. Myös valintamekanismin käytöstä käsittelimme esimerkin tässä luvussa.

Piirteiden syrjäytyksessä aiheutuvat mahdolliset virhetilanteet olisi vältettävissä kovariantista syrjäytystavasta luopumisella. Ainoastaan identtisellä operaation kutsumuodolla voisi syrjäyttää yliluokan operaation. Toisaalta kovariantti syrjäytys tuo monipuolisuutta kielen käyttömahdollisuuksiin, koska tällöin ei olla sidottu tavanomaiseen identtiseen kutsumuotoon syrjäytyksessä. Myös piirteiden uudelleennimeäminen ja sitä seuraava valinta yhdessä voivat aiheuttaa mahdollisesti hankalia ja usein monimutkaisia ratkaisuja. Esimerkiksi nimikonfliktitilanteissa olisi hyvä olla jokin yhtenäinen tapa, jolla ratkaistaan tilanteet. Nyt tilanteen voi ratkaista usealla eri tavalla kuten valintaesimerkissä aiemmin, aiheuttaen hyvinkin sekavia tilanteita ja turhaa monimutkaisuutta. Kaiken kaikkiaan Eiffelissä on kuitenkin pyritty hakemaan selkeitä ratkaisuja useisiin moniperinnän ongelmakohtiin ja -tilanteisiin, joihin tutkielmamme muissa kielissä ei ole otettu juurikaan kantaa.

4.4 Yhteenveto

Tässä luvussa käsittelimme moniperintää Eiffel-ohjelmointikielessä. Aluksi esittelimme perinnän, moniperinnän ja keskeisimpiä näihin liittyviä termejä kielessä. Ehkäpä mielenkiintoisimpina Eiffelin ominaisuuksina esittelimme piirteiden uudelleennimeämisen mahdollisuuden ja valintamekanismin. Seuraavaksi esittelimme esille tulleita perinnän ongelmakohtia Eiffel-ohjelmointikielessä. Totesimme, että täysin vapaa uudelleennimeämisen mahdollisuus sekä myös tämän ja syrjäytyksen mukanaan tuoma valinnan käytön tarve, ja osittain myös kovarianttiperiaatteella toimiva syrjäytysmekanismi ovat keskeisimmät ongelmien ja monimutkaisuuden aiheuttajat Eiffelin moniperinnässä.

5 JAVAN MONIPERINTÄ

Tässä luvussa käsittelemme perintää ja erityisesti moniperintää Java-ohjelmointikielessä. Javan moniperintä on omalla tavallaan hyvin suppea ja poikkeava verrattuna tutkielmamme kahteen muuhun kieleen, ja näitä sen poikkeavia piirteitä tulemmekin käsittelemään tarkemmin. Aluksi esittelemme yleisiä termejä liittyen perintään ja sen käyttöön Javassa. Sen jälkeen käsittelemme tarkemmin Javan moniperintää, sen keskeisimpiä ongelmia ja siinä havaittuja selkeitä puutteita.

5.1 Yleistä

5.1.1 Perintä

Javassa yliluokka voidaan periä eli laventaa, kuten luokan esittelyssä ilmaistaankin `extends` -avainsanalla. Tavallisia luokkia voi periä suoraan ainoastaan yhden. Tavanomaisten luokkien lisäksi kielessä on *liittymäluokkia* (interface) eli niin sanottuja puhtaita abstrakteja luokkia. Nämä liittymäluokat voivat pitää sisällään ainoastaan Javan metodien esittelyjä (abstrakteja operaatioita), mahdollisia vakioarvoja, *sisäkkäisiä* (inner, nested) luokkia tai sisäkkäisiä liittymäluokkia (Gosling 1997, s. 91). Liittymäluokka voidaan *toteuttaa* (implement) luokassa. Tämä ilmaistaan luokan esittelyssä käyttäen `implements` -avainsanaa. Luokka voi toteuttaa – siis toisin sanoen moniperiä – useita liittymäluokkia vapaasti (Gosling 1997, s. 94). Myös liittymäluokka itse voi moniperiä useita liittymäluokkia. Javassa kaikki kielen tavalliset luokat perivät `Object` -luokan implisiittisesti (Gosling 1997, s. 59). Sen perintä tapahtuu siis automaattisesti – suoraan tai mahdollisen yliluokan kautta – eikä sen näin periminen vie luonnollisestikaan luokalta sen ainoata tavallisen luokan perimismahdollisuutta. Tämä `Object` -luokka onkin Javan luokkahierarkian yleinen juuriluokka.

Perinnän hallintaan ja rajoittamiseen Java tarjoaa joitakin työkaluja. Esimerkiksi `final` -avainsanalla voidaan luokan käyttö yliluokkana estää eli tehdä luokasta näin luokkahierarkian lehtiluokka. Samalla avainsanalla voidaan vaikuttaa myös luokan

piirteisiin eli siis operaatioiden kohdalla estää näiden syrjäytys ja näin myös monimuotoisuus sekä attribuuttien kohdalla estää näiden arvon muuttaminen. Näin piirteestä tulee siis *viimeistelty* (final).

Javan suojamääreet vaikuttavat piirteiden saavutettavuuteen. Perintä Javassa on aina julkista eli piirteiden näkyvyyttä aliluokkiin ei voida tiukentaa. Julkiset piirteet ovat saavutettavissa kaikkialta, yksityiset vain luokan – siis sen ilmentymien – sisältä eli suojaus on siis luokkakohtainen. Paketti suojausmääre, joka on piirteiden oletusarvo, tekee piirteet näkyviksi vain luokassa ja luokan oman paketin muissa luokissa. Suojatut piirteet näkyvät luokkaan, sen kaikkiin perillisiin – myös muissa paketeissa – sekä saman paketin luokkiin (Gosling 1997, s. 31). Näistä suojattu määre on käsiteltävä vielä hiukan tarkemmin. Suojattu piirre on saavutettavissa luokassa olioviitteen kautta, jos viite on tämän luokan tyyppiä tai jotain sen alityyppiä. Otetaan tästä esimerkki kirjasta (Gosling 1997, s. 63):

Esimerkki 5.1:

```
public class A {
      protected int f;
      /* ... */
};
public class B extends A { /*...*/ };
public class C extends A { /*...*/ };
public class D extends B { /*...*/ };
```

Näin suojattu `f` -attribuutti `A` -luokassa on saavutettavissa sen jälkeläisissä normaalisti. Mutta `B` -luokassa siihen ei voida viitata `C` -tyyppisen olioviitteen kautta, koska `C` -luokka on periytymishierarkian eri haarassa. Luonnollisesti `D` -tyyppisen olioviitteen kautta ja suoraan viittaaminen onnistuisi tässä tapauksessa. Tätä samaa periaatetta noudatetaan myös C++:n suojattujen piirteiden yhteydessä.

5.1.2 Syrjäytys

Syrjäytettäessä operaatiota Javassa on operaation kutsumuodon oltava identtinen syrjäytettävään nähden, muuten kyseessä on piirteen kuormitus. Syrjäytys tarkoittaa

yliluokan toteutuksen syrjäyttämistä omalla luokan operaation toteutuksella. Ainoastaan saavutettavissa olevia (siis näkyviä) ei-staattisia eli oliokohtaisia ja ei-viimeisteltyjä piirteitä voi syrjäyttää (Gosling 1997, s. 67). Syrjäyttäjän on oltava ei-staattinen (se voi olla kuitenkin viimeistelty) operaatio, jonka näkyvyys on sama tai parempi kuin syrjäytettävällä operaatiolla (Gosling 1997, s. 68). Näistä säännöistä seuraa se, että ainoat sallitut näkyvyysmuutokset syrjäytyksessä ovat seuraavat (TAULUKKO 3):

TAULUKKO 3: Mahdolliset suojamääreiden muutokset syrjäytyksessä

piirteiden suojaus	ei muutu	heikkenee	ei muutu	heikkenee	heikkenee	ei muutu
syrjäytettävä	julkinen	paketti	paketti	paketti	suojattu	suojattu
syrjäyttäjä	julkinen	julkinen	paketti	suojattu	julkinen	suojattu

Javassa voi syrjäyttää ainoastaan operaatioita ja niitäkin vain toisilla operaatioilla. Attribuuttien syrjäytys tai niillä syrjäyttäminen ei siis ole mahdollista. Mikäli aliluokassa esitellään jonkin yliluokan attribuutin kanssa samanniminen attribuutti, tämä piilottaa yliluokan vastaavan nimen (Gosling 1997, s. 68). Yliluokassa esiteltyihin piirteisiin voidaan viitata käyttäen `super` -avainsanaa. Liittymäluokkiin viitataan hieman eri tavoin ja tästä enemmän seuraavassa kohdassa.

5.1.3 Liittymäluokkien moniperintä

Javassa perintä on siis rajoitettu tavanomaisten luokkien osalta yksittäisperintään. Näin moniperintä rajoittuu siis koskemaan vain liittymäluokkia. Liittymäluokkien moniperinnällä on sanottu saavutettavan kaikki halutut ominaisuudet (Joyner 1999, s. 121). Moniperinnän rajoittamista on myös perusteltu mm. sillä, että näin Javassa vältetään nimikonfliktit ja ongelmat yliluokkien piirteisiin viitattaessa (Gosling 1997, s. 94). Edellinen ei tosin pidä täysin paikkaansa ja käsittelemmekin tätä asiaa tarkemmin myöhemmin tässä luvussa.

Javan suurin ero aikaisemmin käsiteltyihin kieliin nähden on juuri moniperinnän voimakas rajoittaminen: Javassa liittymäluokat eivät sisällä toteutusta vaan ainoastaan

toteutettavien operaatioiden kutsumuodot, sisäkkäisiä luokkia ja mahdollisia vakioita. Liittymäluokan perivien luokkien on tarjottava toteutus näille liittymäluokassa esitellyille operaatioille. Vaikka luokka perii useampia liittymäluokkia, joissa on samanniminen operaation nimi ei varsinaista nimikonfliktia synny, koska luokka voi toteuttaa ne kaikki saman kutsumuodon omaavilla operaatiolla. Tässä tilanteessa nimet siis samaistetaan automaattisesti. Jos luokalla on suora yliluokka ja liittymäluokka, joissa kummassakin on saman kutsumuodon omaava operaatio, tilanne on mielenkiintoisempi. Tällöin toteutusta ei tarvitse tarjota, vaan yliluokan perityn operaation katsotaan tarjoavan liittymäluokan vaativan toteutuksen. Otetaan tällaisesta tilanteesta lyhyt esimerkki:

Esimerkki 5.2:

```
public class A {
      public final void tulosta() { /*...*/ }
};

public interface B {
      public void tulosta();
};

public class C extends A implements B {
// syrjäytystä ei tarvita, eikä sitä tässä edes voisi tehdä.
/* ... */
};
```

Nyt A -luokassa oleva operaatio on määritelty `final` -avainsanalla ja sitä ei siis voi Javassa syrjäyttää. Tästä seuraakin, että C -luokassa ei voida toteuttaa perityn liittymäluokan B samannimistä operaatioita, vaikka näin pitäisikin tehdä. Nyt toteutusta ei tarvitse tarjota, koska yliluokan A piirre tarjoaa toteutuksen tälle nimelle C -luokassa.

Mikäli edellisen kaltaisessa tilanteessa halutaan säästää luokan ainoa perintämahdollisuus ja yliluokan A perimistä ei pidetä täysin välttämättömänä – ei siis ole olemassa ilmeistä alityyppisuhdetta, voidaan A -luokan perintä korvata kätevästi koostamisella. Näin C -luokka olisikin seuraavanlainen:

```
public class C implements B {

      public void tulosta() { a.tulosta(); }

      private A a = new A();  // alustus
};
```

Nyt A -luokkaa ei peritäkään, mutta se tarjoaa silti toteutuksen liittymäluokasta B peritylle abstraktille operaatiolle.

5.2 Ongelmakohtia

5.2.1 Nimikonfliktit

Liittymäluokille sallittu moniperintä mahdollistaa myös Javassakin nimikonfliktien synnyn. Kaksi perittyä liittymäluokkaa voi joko omata useita samannimisiä operaation esittelyjä tai useita samannimisiä esiteltyjä vakioattribuutteja. Usean liittymäluokan samannimisten operaatioiden toteutus menee seuraavasti (Gosling 1997, s. 96):

- Jos operaatioiden kutsumuodot ovat identtiset, ne voidaan toteuttaa yhdellä operaatiolla.
- Jos operaatioiden parametrityypit tai lukumäärä on poikkeava, luokassa voidaan toteuttaa nämä usealla vastaavalla operaatiolla (tällöin kuormitetaan nimeä normaalisti).
- Jos operaatioiden kutsumuodot eroavat ainoastaan paluuarvojen tyypeiltä, ei operaatioita voi yhdessä luokassa toteuttaa ja näin näitä liittymäluokkia ei voi periä.
- Jos operaatiot eroavat ainoastaan niissä *nostettavien* (throws) *poikkeusten* (exception) suhteen, on perivässä luokassa toteutettava nämä operaatiot yhdellä operaatiolla. Useita identtisen kutsumuodon omaavia operaatiota, jotka eroavat ainoastaan näiden nostavien poikkeusten osalta ei samassa liittymäluokassa (tai luokassa) luonnollisestikaan saa olla.

Jos nimikonfliktissa on liittymäluokissa esitellyt vakioarvot, on tilanne huomattavasti yksinkertaisempi: piirteeseen pitää viitata liittymäluokan nimeä käyttäen. Otetaan seuraavaksi lyhyt esimerkki, jossa käsitellään liittymäluokkien piirteiden nimikonfliktitilannetta:

Esimerkki 5.3:

```
public interface S {
      int CONSTANT = 500; // vakioarvo, aina public static final
      void print();
};
public interface A extends S {
      int MAX = 100;
      public int f(String s, int a);
      public void g() throws AExceptiton;

      public class Inner implements S { // sisäkkäinen luokka
            void print() { /*...*/ } // toteuttaa rajapinnan
      };
};
public interface B extends S {
      int MAX = 200;
      public boolean f(int a);
      public void g() throws BException;

      public class Inner implements S, B { // sisäkkäinen luokka
            void print() { /*...*/ }
            public boolean f(int a) { /*...*/ }
            public void g() { /*...*/ }
      };
};
public Class C implements A, B {
      public int f(String s, int a) { /*...*/ }
      public boolean f(int a) { /*...*/ }
      void print() { /*...*/ }
      public void g() { /*...*/ }
};
```

Näin C -luokassa voidaan periä nämä kaksi liittymäluokkaa, joiden f -operaatiot ovat erotettavissa toisistaan poikkeavien parametrilistojen ansiosta. Pelkät paluutyypin eroavaisuudet eivät ole Javassa sallittuja. Operaatio g täytyykin toteuttaa yhdellä ainoalla operaatiolla. Javassa syrjäyttävä (tai toteuttava) operaatio voi nostaa

vähemmän poikkeuksia kuin sen ylityyppi on alunperin määritellyt (Gosling 1997, s. 97). Operaatio, joka syrjäyttää tai toteuttaa – mahdollisesti jopa useita – perittyjä operaatioita ei voi nostaa poikkeuksia, joita nämä toteutettavat eivät nosta (Gosling 2000, s. 176). Näin edellisessä `C` -luokan `g` -operaatio ei voi nostaa kumpaakaan poikkeusta, koska sen toteuttamat liittymäluokat eivät mahdollista sitä: `B` -liittymäluokassa oleva `g` -operaatio ei nosta `AException` -poikkeusta ja vastaavasti `A` -liittymäluokassa oleva `g` -operaatio ei nosta `BException` -poikkeusta. Yliluokan poikkeusten nostamatta jättäminen ei luonnollisestikaan ole välttämättä mahdollista tilanteissa, joissa yliluokan syrjäytettävää operaatiota (siis toiminnallisuutta) kutsutaan syrjäyttävän sisältä käyttäen **`super`** -avainsanaa. Syrjäytettävän operaation kutsuhan aiheuttaisi automaattisesti mahdollisuuden sen kaikkien nostavien poikkeusten syntyyn ja nostettavat poikkeukset on Javassa aina käsiteltävä – lukuun ottamatta `RuntimeException` ja `Error` -luokkien sekä näiden jälkeläisten ilmentymiä.

Mikäli esimerkin `C` -luokassa haluttaisiin vielä käyttää liittymäluokasta perittyjä vakioarvoja, voitaisiin niihin viitata yksinkertaisesti luokan nimeä käyttäen `B.MAX` tai `A.MAX`. Esimerkin liittymäluokat perivät vielä yhteisen *yliliittymäluokan* (superinterface), josta perityt piirteet samaistetaan mikäli ne peritään johonkin useampaan kertaan (Gosling 2000, s. 205).

Edellisessä esimerkissä liittymäluokat perivät siis yhteisen `S` -yliliittymäluokan. Mielenkiintoisinta esimerkissä on liittymäluokkien sisäkkäiset luokat, jotka aiheuttavat perivään luokkaan nimikonfliktitilanteen. Kumpikin `Inner` -niminen sisäkkäinen luokka toteuttaa `S` -liittymäluokan `print` -operaation ja toinen vielä sen oman *ympäröivän* (enclosing) liittymäluokan operaatiot. Näin liittymäluokan sisäkkäisen luokan avulla voidaan saada liittymäluokan sisään toiminnallisuutta ja näin myös kiertää – tosin hieman keinotekoisesti – moniperinnän rajoittuneisuutta Javassa. Liittymäluokan sisäkkäinen luokka – mahdollisesti myös liittymäluokka – on aina automaattisesti julkinen ja staattinen eli luokkakohtainen (Gosling 1997, s. 93). Näin siihen pitääkin viitata aina liittymäluokan ja sisäkkäisen luokan nimellä. Yksi mahdollinen tapa käyttää operaatiota kyseisessä tilanteessa on uusien sisäkkäisten luokkien luonti ja perinnän käyttö. Tästä tavasta on esitetty esimerkki seuraavaksi,

jossa toteutamme kolmannessa luvussa esitetyn CowboyWindow -esimerkin tapaisen tilanteen Javaa käyttäen.

Esimerkki 5.4:

```
public interface Cowboy {

	public void draw();
	public class Inner implements Cowboy {

		public void draw() { /*...*/ } // toteutus
	}
};

public interface Window {

	public void draw();
	public class Inner implements Window {

		public void draw() { /*...*/ } // toteutus
	}
};

public class CowboyWindow implements Cowboy, Window {

	private CCowboy cc = new CCowboy();
	private WWindow ww = new WWindow();

	public class CCowboy extends Cowboy.Inner {

		public void cow_draw() {

			super.draw();
		}
	}
	public class WWindow extends Window.Inner {

		public void win_draw() {

			super.draw();
		}
	}
	public void draw() {
```

```
            cc.cow_draw();
            ww.win_draw();
      }
};
```

Kuten edellisistä esimerkeistä hyvin näemme, Javan perintä on – vaikka siinä onkin moniperintää rajoittettu – suhteellisen monimutkainen ja osittain jopa sekava toteutukseltaan. Javassa on päädytty rajoittamaan moniperintää, mutta sen vuoksi siihen on jouduttu ottamaan mukaan sitä korvaamaan vielä jopa monimutkaisempia rakenteita, kuten esimerkiksi luokkien ja liittymäluokkien sisäkkäiset luokat (Joyner 1999, s. 137).

5.2.2 Rajoittuneisuus

Javassa ei siis sallita tavanomaisille luokille moniperintää ja tämä rajoittaakin luonnollisesti jonkin verran toteutusmahdollisuuksia. Liittymäluokkia moniperittäessä luokka (tai liittymäluokka) perii sopimuksia eli rajapintojen operaatioiden kutsumuotoja, jotka on luokassa toteutettava. Näin luokka samalla on myös tyypiltään näitä perittyjä liittymäluokkia. Liittymäluokkien käyttö Javassa jääkin yleensä pääasiallisesti tyypinperinnäksi. Varsinainen ohjelmistokoodin uudelleenkäyttöön perustuva toiminnallisuuden perintä ei ole yleensäkään Javassa kovinkaan järkevää, koska toimintaa voidaan periä vain yhdeltä yliluokalta. Toiminnan moniperinnän rajoittamista voidaan kuitenkin halutessa hieman kiertää mm. koostamalla ja sisäkkäisillä luokilla. Otetaan seuraavaksi koostamisesimerkki, jossa luodaan keinotekoisesti haarautuvaa moniperintää vastaava tilanne ja jossa saadaan aikaan vieläpä nimikonfliktitilanne:

Esimerkki 5.5:

```
public class A {
      public void r() {/* toiminta */}

};
public class B extends A {
      public void r() {/* syrjäytys */}

};
```

```
public class C extends A {
      public void r() {/* syrjäytys */}

};
public interface E {
      public static final A a = new B();

};
public interface F {
      public static final A a = new C();

};
public class D implements E, F {
      public void r() {

            E.a.r(); // tai ((E)a).r();
            F.a.r(); // tai ((F)a).r();
      }
};
```

Näin liittymäluokissa on esitelty vakiomuuttujina kaksi oliota, joihin viitataan samannimisten viitteiden kautta. Nämä ovat liittymäluokkakohtaisia muuttujia johtuen **static** -avainsanasta – vakiomuuttujien esittelythän olivat liittymäluokissa sallittuja. Näin saadaan siis aikaan keinotekoisesti tilanne, jossa moniperitään kahden luokan toimintaa ja saadaan vieläpä aikaan tahallinen nimikonflikti. Nimikonfliktin ratkaisemiseksi on `r` -operaatiota kutsuttava käyttäen liittymäluokan nimeä kutsun yhteydessä, kuten esimerkissä onkin tehty. Sen sijaan **super** -avainsanaa ei voida käyttää, koska tällä viitataan aina luokan (yliluokkaan) aliolioon. Otetaan **super** -avainsanan käytöstä vielä lyhyt esimerkki:

Esimerkki 5.6:

```
public class A {
      public void r() {}

};
public class B extends A {
      public void r() {}

};
public class C extends B {
      public void r()
```

```
    {
        super.r(); // A.r() vai B.r() ??
    }
};
```

Yllä oleva kutsu `super.r()` menee `C` luokan välittömään yliluokkaan `B` ja näin `A` -luokassa olevaa `r` -operaatiota ei voida kutsua `C` -luokasta suoraan.

5.3 Korjausehdotuksia

Javan moniperintä on suhteellisen suppea ja rajoittunut. Mitään monimutkaisuuden mukanaan tuomia ongelmia ei juurikaan ilmene – nimikonfliktit ovat tosin mahdollisia – niin kuin aikaisemmin esitettiin. Oikeastaan rajoittuneisuus – nimikonfliktien ohella – onkin Javan moniperinnän suurin ongelma. Moniperinnän salliminen ainoastaan abstrakteille liittymäluokille rajoittaa toteutuksen perintää ja perinnän käyttöä yleensäkin. Tosin pieni perinnän käytön rajoittaminen ei välttämättä ole huono asia, mutta Javaan valittu rajoittaminen on suhteellisen voimakasta ja sen perintä on silti suhteellisen sekava, johtuen mm. sisäkkäisistä luokista. Toisten luokkien toteutusta voidaan tosin ottaa käyttöön käyttämällä esimerkiksi koostamista tai jopa niin kuten edellisen kohdan esimerkissä oli tehty. Näin liittymäluokkien moniperinnällä sekä koostamisella voidaan kiertää hiukan luokkien yksittäisperintä rajoitetta ja saada aikaiseksi samantapainen haarautuvan moniperinnän tilanne kuin tavanomaisten luokkien moniperinnällä. Kaiken kaikkiaan moniperinnän salliminen Javassa ainoastaan liittymäluokille tuntuu oudolta ratkaisulta, ottaen huomioon ettei näiden käytössä vältetä edes moniperinnän yleistä ongelmaa eli nimikonflikteja.

Mikäli Javan moniperintä kuitenkin laajennettaisiin vapaammaksi koskemaan kaikkia luokkia, tulisi sen mukana myös tukku perinteisiä moniperinnän ongelmia, jotka olisi vastaavasti ratkaistava – tai olla ratkaisematta, niin kuin useissa moniperinnän sallivissa kielissä onkin valitettavasti tehty. Mutta niin kuin jo aiemmin totesimme, valitettavasti rajoitetun moniperinnän liittymäluokat voivat aiheuttaa nimikonflikteja tietyissä tilanteissa, vaikka niistä ei saa edes täysiä moniperinnän hyötyjä mm. ohjelmistokoodin perinnän suhteen. Mitään muuta välitöntä parannettavaa tai

korjattavaa – lukuun ottamatta sen liiallista rajoittuneisuutta – ei Javan moniperinnästä juuri löydykään.

5.4 Yhteenveto

Tässä kappaleessa käsittelimme moniperintää Java-ohjelmointikielessä. Ensimmäisenä esittelimme perinnän käyttömahdollisuuksia ja rajoituksia. Tämän jälkeen käsittelimme Javan moniperintää, joka on sallittu ainoastaan liittymäluokille. Tavallisille luokille on mahdollista vain yksittäisperintä. Liittymäluokat ovat puhtaita abstrakteja luokkia, jotka eivät voi pitää sisällään toteutusta vaan ainoastaan operaatioiden esittelyjä. Seuraavaksi pohdimme Javan moniperinnän ongelmia, joista selkein on kielen tiukka rajoittuneisuus moniperinnän – erityisesti koodin perinnän – käytön suhteen. Lopuksi kerroimme myös mahdollisia keinoja parantaa Javan moniperintää. Nämä parantavat keinot rajoittuvat pääasiassa moniperinnän sallimiseen myös tavanomaisille luokille. Rajoittamaton moniperintä tosin toisi Javaan mukanaan muista kielistä tuttuja moniperinnälle ominaisia ongelmia ja lisää monimutkaisuutta, jotka sitten olisi ratkaistava jollain tavalla.

6 KIELTEN VERTAILU

Tässä luvussa tulemme vertailemaan aiemmissa luvuissa esiteltyjä kieliä: C++:aa, Eiffeliä ja Javaa. Tulemme käymään läpi kielten moniperintään liittyviä ominaisuuksia ja myös vertailemaan näitä keskenään. Pääasiallinen vastakkainasettelu tulee muodostumaan C++:n ja Eiffelin ominaisuuksien välille, koska näiden kahden kielen monipuoliset, mutta myös monimutkaisemmat ominaisuudet ovat luonnollisesti paremmin vertailtavissa. Javan moniperintä onkin näitä kahta kieltä huomattavasti suppeampi toteutukseltaan. Lopuksi voimme vielä todeta, ettei kieliä pyritä asettamaan varsinaisesti paremmuusjärjestykseen, vaikkakin kielistä on havaittavissa selkeitäkin eroja.

6.1 Periytymistapa

6.1.1 Yleistä periytymisen vertailusta

Periytymistavan voidaan katsoa määräävään, miten yliluokka eli sen piirteet peritään luokkaan. Kaikissa tutkielmamme kielissä periytyminen tapahtuu eksplisiittisesti, ilmaisemalla perittävien luokkien nimet perivän luokan esittelyssä eli perintä perustuu jo toisessa luvussa mainittuun nimiyhtäläisyyteen. Periytymisen käyttöä vertailtavissa kielissä ei ole rajoitettu juuri millään lailla, tosin Javassa tavanomaisten luokkien perintä on rajoitettu yksittäisperintään. Käytännössä tätä ei voidakaan juuri valvoa ja vastuu periytymisen järkevästä ja asianmukaisesta käytöstä jää aina viime kädessä ohjelmoijalle. Esimerkiksi Meyerin kirjassaan mainitseman autonomistajaesimerkin kaltainen periytymisen väärinkäyttö on aina mahdollista, eikä tällaista voi mitenkään ohjelmointikielessä estää (Meyer 1997, s. 810). Kyseinen esimerkki on eräästä ohjelmistoalan oppikirjasta otettu. Siinä on esitelty moniperintää ja sen käyttöä esittelemällä autonomistaja -luokka, joka perii sitten kaksi yliluokkaa: auton ja henkilön. Tässä kohtaa esimerkin kirjoittaja ei ole täysin ymmärtänyt periytymisen ja koostamisen välistä eroa, eikä periytymisen käyttötapaa tai edes sen merkitystä.

Seuraavassa periytymistavan vertailu jaetaan erikseen periytymisen suojaukseen ja sen lukumääräsuhteisiin. Tarkastellaan ensiksi suojauksia vertailtavissa kielissä ja tämän jälkeen perinnän lukumääräsuhteita.

6.1.2 Suojamääreet

Tutkielmamme kielistä ainoastaan C++:ssa voidaan periä yliluokan piirteitä yksityisesti tai suojatusti. Yksityinen periytymistapahan todettiin joissain tilanteissa hyvinkin ongelmalliseksi, johtuen mm. yksityisen periytymistavan tuomasta mahdollisuudesta syrjäyttää saavuttamattomia operaatioita. Tosin yksityinen periytymistapa tuo puhtaassa toiminnallisuuden perinnässä yhden selkeän edun julkiseen ja suojattuun (alityyppi) periytymistapaan verrattuna. Otetaan tästä seuraavanlainen esimerkki:

Esimerkki 6.1:

```
class Maakulkuneuvo {
public:
      virtual void lastaa_kuorma() { /*...*/ }
/*...*/
};
class Laiva {
public:
      virtual void laske_syvays() = 0;
/*...*/
};
class Rahtilaiva : public virtual Laiva, private Maakulkuneuvo {
public:
      virtual void laske_syvays() { /*...*/ }
/*...*/
};
class Satama_alue {
public:
      void f() {

            list<Maakulkuneuvo*> autohalli;
            Laiva l_ptr = new Rahtilaiva; // Ok.
            Maakulkuneuvo m_ptr = new Rahtilaiva; // Ei käy.
            autohalli.push_back(l_ptr); // Ei käy.
      }
```

```
/*...*/
};
```

Näin yksityisellä periytymistavalla voidaan kätevästi estää sopimattomat sijoitukset yliluokka -tyyppisiin osoittimiin. C++:n yksityisellä periytymistavalla voidaan siis periä haluttua toiminnallisuutta yliluokilta, jotka eivät kuitenkaan sovellu millään lailla perivän luokan ylityypiksi. Eihän rahtilaivaakaan saa sijoittaa autohallin suojiin. Vastaavanlaista suojaa vääriä sijoituksia vastaan ei ole mahdollista saada aikaan Eiffelissä eikä Javassa. Tosin Javassa tällainen toiminnan perintä ei ole edes järkevää, johtuen moniperinnän rajoituksista. Edellä mainitun kaltainen suoja puuttuu esimerkiksi luvussa kaksi mainitulta Eiffelin luokkakirjaston ARRAYED_STACK -luokalta, jonka ilmentymän voi sijoittaa sekä ARRAY että STACK -tyyppiseen olioviitteeseen.

Javassa periytyminen on aina julkista – yliluokan yksityiset piirteet eivät tietenkään näy aliluokille, eikä niitä voi syrjäyttää. Myös Eiffelissä piirteet peritään aina julkisesti, eikä luokille ole edes mahdollista piilottaa piirteitä jälkeläisiltä. Näin Eiffelissä ei siis ole edes olemassa koko yksityisen piirteen käsitettä. Kielten suojamääreistä voidaan vielä todeta yksi pieni, mutta kuitenkin merkityksellinen ero: C++:n ja Javan suojaus on luokkakohtainen, kun taas Eiffelissä suojaus on oliokohtainen. Mutta niin kuin Eiffeliä käsittelevässä luvussa aiemmin näytettiin, voidaan sen piirteiden suojaus muuttaa halutessa luokkakohtaiseksi. Lisäksi C++:ssa ja Javassa suojattua piirrettä voidaan käsitellä ainoastaan periytymishierarkian samassa haarassa (kuin luokka itse) olevan aliluokka -tyyppisen viitteen tai osoittimen kautta. Tätä rajoitetta ei ole Eiffelissä.

6.1.3 Lukumääräsuhteet

Moniperinnän periytymistapa voi olla lukumääräsuhteellisesti yliluokat jakavaa tai jakamatonta. C++:ssa on mahdollista määrittää perintä halutusti joko virtuaaliseksi eli jakavaksi tai jakamattomaksi. Näin on mahdollisuus valita, peritäänkö yliluokka omana alioliona jota ei jaeta (suotavaa yksityisen suojamääreen kanssa), vai peritäänkö yliluokka virtuaalisesti eli jakavasti (suotavaa julkisen suojamääreen kanssa). Javassa moniperityt liittymäluokat samaistetaan automaattisesti eli niiden

perintä on aina jakavaa. Myös Eiffelissä moniperintä on jakavaa ja siten haarautuvassa moniperinnässä jokainen useaa kautta peritty yliluokka samaistetaan aina samaksi alioliokси. Eiffelissä ei saada edes toistuvalla perinnällä perittyä useita samantyyppisiä aliolioita, vaan toistuvasti peritty yliluokka samaistetaan aina yhdeksi samaksi luokaksi. Mielenkiintoisena poikkeuksena edelliseen on hieman poikkeava tilanne, joka saadaan Eiffelissä aikaan uudelleennimeämisellä. Tällä mekanismilla on mahdollisuus aiheuttaa perittyjen piirteiden monistuminen ja samalla myös tilanteen monimutkaistuminen – uudelleennimeäminenhän voi johtaa pakolliseen valinnan käyttöön dynaamisen moninimisyyden ratkaisemiseksi. Uudelleennimeämisessä ei kuitenkaan synny kokonaisia uusia aliolioita, vaan ainoastaan yksittäiset uudelleennimetyt attribuutit jakautuvat eli monistuvat. Näin Eiffelin perinnän voidaankin katsoa tällaisessa tapauksessa tapahtuvan ennemminkin attribuuttikeskeisesti, eikä alioliomallin mukaisesti. Otetaan seuraavaksi esimerkki Eiffelin toistuvalla moniperinnällä syntyvästä rakenteesta, jota ei ole mahdollista luoda C++:ssa eikä Javassa perinnän avulla.

Esimerkki 6.2:

```
class SOLU
      feature f
... end

class KAKSISOLUKKO inherit
      SOLU rename f as f1
      SOLU rename f as f2 select f2 end
... end

class KOLMISOLUKKO inherit
      SOLU rename f as f1
      SOLU rename f as f2
      SOLU rename f as f3 select f3 end
... end

-- NELISOLUKKO, VIISISOLUKKO jne...

class SOLUKENNO
feature {NONE}
      ARRAY[SOLU] // voi sisältää kaikkia em.
... end
```

Näin voidaan luoda luokka SOLUKENNO, joka koostuu toistuvasti perityistä soluista. Toistuvasti perittävän SOLU -luokan halutut piirteet (attribuutit) voitaisiin uudelleen nimetä ja näin ne saataisiin monistettuina perivään luokkaan.

Selvästi vaihtelevin ja monimutkaisin periytymistapavalikoima löytyy siis C++-kielestä. Siinä voidaan periä kolmella eri suojaustasolla yliluokkia – ja samanaikaisesti vieläpä joko virtuaalisesti tai ei-virtuaalisesti eli yhteensä kuudella eri tavalla. Eiffelissä ja Javassa periytyminen on C++:aan verrattuna aina periaatteessa virtuaalista ja julkista. Näin esimerkiksi mahdollisia hankaluuksia aiheuttava yksityinen periytymistapa jää kokonaan pois. Eiffelin ja Javan moniperintä onkin periytymistavan osalta huomattavasti yksinkertaisempi ja suoraviivaisemmin toteutettu. C++:n tarjoamista eri variaatiosta voi muodostua ongelmia, kuten esimerkiksi saavuttamattoman virtuaalisen operaation syrjäytys. C++:n kaikkien eri periytymisvariaatioiden käyttömahdollisuudelle ei ole edes mitään varsinaista tarvetta, joten käytännössä kannattaisikin suosia ainoastaan julkista ja virtuaalista tai yksityistä ja ei-virtuaalista periytymistapayhdistelmää, kuten aiemmin jo C++:aa käsittelevässä luvussa todettiin.

6.2 Syrjäytys

6.2.1 Yleisiä eroja

Operaatioiden syrjäytys on mahdollista kaikissa tutkielmamme kielissä. Kuitenkin vain C++:ssa operaatio on ensimmäistä kertaa esiteltäessä esiteltävä virtuaalisena, jotta se voitaisiin myöhemmin syrjäyttää. Sen sijaan Javassa ja Eiffelissä operaatiot ovat automaattisesti syrjäytettävissä, mutta näissä kuitenkin voidaan estää syrjäytys erikseen määrätyllä avainsanalla. Javassa voidaan vieläpä estää luokan käyttö periytymiseen eli tehdä luokasta hierarkian lehtiluokka. Vastaavaan ei ole mahdollisuutta C++:ssa eikä Eiffelissä.

Yksi Eiffelin syrjäytykseen liittyvä erikoisuus kahteen muuhun kieleen nähden on mahdollisuus syrjäyttää operaatio samannimisellä attribuutilla. Toinen Eiffelin syrjäytyksen merkittävä ero C++:aan ja Javaan nähden on kovariantin syrjäyttämisen

mahdollisuus. Eiffelissä syrjäyttäminen on siis hivenen vapaampaa kuin näissä kahdessa muussa kielessä. Kovariantista syrjäytyksestä voi kuitenkin syntyä omat riskinsä, niin kuin aiemmin neljännessä luvussa totesimme.

Missään tutkielmamme kolmesta kielestä ei ole juuri minkäänlaista kontrollia operaation syrjäyttämiselle. Ainoastaan syrjäyttävän operaatio kutsumuodolle asetetut vaatimukset on toteutettava eli kutsumuodon on oltava identtinen (tai Eiffelissä kovariantti) syrjäytettävään nähden. Eiffelissä syrjäyttävän operaation on vielä noudatettava mahdollisia syrjäytettävän operaation *esiehtoja* (preconditions) sekä *jälkiehtoja* (postconditions). Syrjäyttävän operaation esiehdon on oltava sama tai väljempi ja jälkiehdon sama tai tiukempi kuin syrjäytettävässä yliluokan operaatiossa (Meyer 1997, s. 573). Perusperiaate esi- ja jälkiehdoissa on seuraavanlainen: operaation kutsujan on sitouduttava noudattamaan operaation esiehtoa ja mikäli näin on, operaation (toteuttajan) on noudatettava sen suorituksen jälkeistä jälkiehtoa. Mikäli esi- tai jälkiehto ei kuitenkaan jostain syystä toteudu, järjestelmä nostaa tilanteesta poikkeuksen (Meyer 1997, s. 413).

Missään tutkielmamme kielessä syrjäyttävän operaation varsinaisen toiminnan ei tarvitse olla millään lailla johdonmukaista syrjäytettävään verrattuna. Esimerkiksi Beta-ohjelmointikielessä on noudatettava *käyttäytymisen yhdenmukaisuutta* (behavioral consistency) syrjäyttävässä operaatiossa (Bracha 1990, s. 1). Näin syrjäyttävästä operaatiosta kutsuttaisiin yliluokkien operaatioita. Esimerkiksi Eiffelissä tällainen saataisiin aikaan halutessa käyttämällä `Precursor` -avainsanaa ja mahdollista luokan nimeä, mikäli moniperinnässä syrjäytetään useita yliluokkien operaatioita. Vastaavasti Javassa sama voitaisiin tehdä `super` -avainsanalla ja operaation nimellä – toteutustahan voidaan ainoastaan yksittäisperiä Javassa. C++:ssa yliluokkien syrjäytettäviä operaatioita olisi kutsuttava käyttäen apuna luokkatarkenninta tai tyyppimuunnosoperaatiota `this` -avainsanan kanssa.

6.2.2 Syrjäytyksen määräytyminen

Mikäli haarautuvassa moniperinnässä tehdään syrjäytyksiä usealla eri periytymispolulla, voi operaatioiden kutsujen kohdistaminen oikeaan luokkaan tulla

monimutkaisemmaksi ja hankalammiksi. Javan kohdalla tällaisessa tilanteessa ei ole ongelmaa, koska moniperintä koskee käytännössä vain liittymäluokkia ja varsinaista toiminnallisuutta voidaan periä ainoastaan yhdeltä yliluokalta. Näin Javassa ei siis tarvitse koskaan tehdä valintaa mahdollisten eri toteutuksien välillä – toisin siinä voidaan sisällyttää toimintaa liittymäluokkiin sisäkkäisillä luokilla, mutta varsinaista virtuaalisten operaatioiden konfliktitilannetta sen moniperinnässä ei voi koskaan syntyä. C++:ssa tällaisissa haarautuvan moniperinnän tilanteissa, joissa operaatio on syrjäytetty usealla periytymispolulla, vallitsee hallitsevuussääntö. Periytymishierarkiassa on siis oltava yksi hallitseva syrjäytys, mikäli jokin piirre on syrjäytetty usealla periytymispolulla. Eiffelissä ei tällaista hallitsevuussääntöä ole ja siinä tällaiset tilanteet onkin ratkaistava pääasiallisesti sen omalla erityismekanismilla eli valinnalla. Näin ohjelmoijan on eksplisiittisesti tehtävä mahdollisesti tarvittava valinta syrjäytysten välillä sen sijaan, että ohjelman kääntäjä pyrkisi päättelemään automaattisesti periytymishierarkiasta hallitsevan operaation syrjäytyksen. Tässä onkin yksi mielenkiintoinen ero näiden kahden kielen moniperinnän välillä: C++:ssa pyritään automaattisesti löytämään (päättelemään) kutsuttava operaatio, kun taas Eiffelissä ei jätetä mitään epäselväksi ja ohjelmoijan onkin eksplisiittisesti valittava aina haluamansa toteutus perintähierarkiasta.

6.3 Moniperinnän nimikonfliktit

Kaikkien tutkielmamme kielten moniperinnässä on mahdollisuus nimikonfliktien syntyyn. Javassa nimikonflikteja voi syntyä liittymäluokkien moniperinnässä ja C++:ssa luokkien moniperinnässä. C++:n nimikonfliktit voivat syntyä siis toiminnallisuutta sisältävien operaatioiden välille, kun taas Javassa ainoastaan liittymäluokkien abstraktien operaatioesittelyjen – myös mahdollisten vakioarvojen sekä sisäkkäisten luokkien – välille. Myös Eiffelin moniperinnässä syntyy nimikonflikteja. Eiffel on kuitenkin tutkielmamme ainoa kieli, joka tarjoaa selkeän ratkaisun nimikonfliktien estämiseen. Tämä on piirteiden uudelleennimeäminen. Näin Eiffel onkin ainoa tutkielmamme kielistä, jossa on pyritty ratkaisemaan nimikonfliktitilanteet edes jollain tavalla. C++:ssa tilanteet joudutaan pahimmassa tapauksessa ratkaisemaan keinotekoisesti uusilla apuluokilla, kuten kolmannessa luvussa näytettiin.

Uudelleennimeämisestä on tosin hyvä todeta, että rajoittamaton uudelleennimeäminen tuo mukanaan lisää monimutkaisuutta, kuten aikaisemmin neljännessä luvussa todettiin. Perintähierarkiasta tulee vaikeammin tulkittava ja siinä voi tapahtua myös piirteiden monistumista, joka on hankalimmissa tapauksessa ratkaistava valintamekanismilla. Valintamekanismihan on jouduttu tuomaan apuvälineeksi Eiffelin moniperintään piirteiden uudelleennimeämisen sekä syrjäytyksen johdosta ja tämä mekanismi omalta osaltaan monimutkaistaa moniperintää ja perintähierarkiaa kielessä.

6.4 Yhteenveto

Tässä luvussa vertailimme keskeisimpiä eroja C++:n, Eiffelin ja Javan moniperinnässä. Aluksi käsittelimme periytymistapaa, josta on jo havaittavissa C++:n huomattavasti Eiffeliä ja Javaa monimutkaisempi ja sekavampi toteutus: C++:sta periytymistavan mahdollisia kombinaatioita löytyy kaiken kaikkiaan kuusi kappaletta. Eiffelissä ja Javassa sen sijaan on mahdollista vain yhdenlainen periytymistapa. Seuraavaksi vertailimme syrjäytystä. Tästä löytyykin kielten ehkä mielenkiintoisin ero moniperinnässä: C++:n haarautuvassa moniperinnässä hallitsevuussääntö ratkaisee voimassaolevan syrjäytyksen, mikäli jokin peritty piirre on syrjäytetty usealla periytymispolulla. Eiffelissä vastaavanlaisissa tilanteissa ei ole samankaltaista hallitsevuussääntöä, vaan nämä tilanteet on ratkaistava usein sen omalla erityismekanismilla eli valinnalla. Tässä ohjelmoijan on eksplisiittisesti valittava haluttu perityistä piirteistä. Javassa ei edellisen kaltaista ongelmaa ole, koska sen moniperintä koskee vain abstrakteja liittymäluokkia, eikä siis periaatteessa lainkaan toteutusta. Seuraavaksi käsittelimme näiden kolmen kielen yhteistä – ehkäpä kaikkein yleisintä – moniperinnän ongelmaa eli nimikonflikteja. Näistä kielistä ainoastaan Eiffelissä on pyritty tarjoamaan ratkaisu nimikonflikteihin: Eiffelissä nimikonfliktit voidaankin ratkaista piirteiden uudelleennimeämisellä, joka on hyvin selkeä ja suoraviivainen tapa ratkaista nämä ongelmat. C++:ssa ja Javassa ei ole mitään näin siistiä tapaa ratkaista nimikonflikteja, vaan niissä on haluttaessa viitattava konfliktissa olevaan nimeen esimerkiksi luokkatarkentimella. Näin olemme vertailleet moniperintää ja sen käyttämistä C++-, Eiffel- ja Java-ohjelmointikielissä. Kielistä on selkeästi havaittavissa suuria eroja moniperinnän käyttömahdollisuuksien ja toteutuksen suhteen. C++:n ja Eiffelin ollessa suhteellisen monimutkaisia – on tosin

huomattava, että Eiffelin moniperintää on selkeästi ajateltu pisimmälle kieltä kehitettäessä. Java on taas suhteellisen suppea toteutukseltaan ja sen käyttömahdollisuudet jättävätkin paljon toivomisen varaa. Tosin Javan perintä on moniperinnän rajoituksista huolimatta silti suhteellisen monimutkainen. Tämä johtuu mm. sisäkkäisten luokkien hyvin vapaista käyttömahdollisuuksista.

7 YHTEENVETO

Tässä tutkielmassa käsittelimme ja tutkimme mielenkiintoista, mutta myös kiisteltyä olio-ominaisuutta eli moniperintää ja sen käyttötapoja C++-, Eiffel- ja Java-ohjelmointikielissä. Aluksi tutkielman toisessa luvussa käsittelimme yleisesti perintää sekä siihen ja olio-ohjelmointiin yleensä liittyviä keskeisiä termejä. Tämän jälkeen käsittelimme tarkemmin juuri moniperintää sekä tämän teoriaa ja yleisiä ongelmia. Kolmannessa, neljännessä ja viidennessä luvussa käsittelimme tarkemmin moniperintää sekä sen puutteita ja ongelmia tutkielman kohteina olevissa kielissä, C++:ssa, Eiffelissä ja Javassa. Lopuksi kuudennessa luvussa vertailimme esiteltyjen kielten moniperinnän toteutuksia ja niissä havaittuja eroja, puutteita ja ongelmia.

Tutkielman toisessa luvussa käsittelimme yleisiä termejä liittyen perintään olio-ohjelmoinnissa. Ensi kertaa perintä esiteltiin SIMULA 67 -ohjelmointikielessä, josta se on melko suoraan kopioitu mm. C++:aan. Seuraavaksi esittelimme perinnän erilaisia jakotapoja, joista keskeisin on ehkä jako käsitteelliseen mallintamiseen ja muuhun eli puhtaaseen ohjelmistokoodin perintään. Perintä voidaan nähdä jossain määrin koostamisen vastakohtana olio-ohjelmoinnissa, mutta kummassakin on periaatteessa kyse valmiina olevan koodin hyväksikäytöstä ohjelmistojen kehitystyössä. Seuraavaksi syvennyimme käsittelemään moniperintää. Moniperinnän yhteydessä esittelimme kolme erilaista perinnän tarkastelumallia. Alioliomallissa perityt yliluokat nähdään luokan aliolioina ja perivä aliluokka ylioliona. Alioliomalli helpottaa varsinkin monimutkaisimpien moniperintähierarkioiden tarkastelua. Alioliomallille vastakkaisena voidaan nähdä attribuuttikeskeinen malli, jossa yliluokat voidaan katsoa perittävän attribuutti kerrallaan, eikä kokonaisina olioina. Kolmas esitelty malli oli lineaarinen malli. Tässä yliluokkien perintäjärjestys voi vaikuttaa niiden piirteiden periytymiseen luokkaan. Näiden mallien jälkeen esittelimme moniperinnästä kaksi eri ilmenemismuotoa. Yksinkertaisempi muoto on riippumaton moniperintä, jossa luokka perii useita eri yliluokkia, mutta ei mitään näistä useasti. Haarautuvassa moniperinnässä luokka taas perii jonkin yliluokan useaan kertaan, esimerkiksi kahden eri vanhemman kautta. Toisen luvun lopuksi käsittelimme moniperinnän ongelmia. Yleisin ongelma on horisontaaliset nimikonfliktit, joita voi syntyä jo yksinkertaisessa riippumattomassa moniperinnässä. Riippumatonta

moniperintää monimutkaisempi haarautuva moniperintä aiheuttaa myös omia ongelmiaan, jotka useasti ovat hankalia ratkaista, johtuen mm. perintähierarkian voimakkaasta monimutkaistumisesta.

Kolmannessa luvussa käsittelimme moniperintää C++:ssa. C++:ssa periytymistapa määrää miten yliluokkia peritään luokkaan. Tässä havaitsimme ensimmäisen ongelman kielessä: C++:ssa on peräti kuusi erilaista periytymistavan yhdistelmää, joista ainoastaan kolme olisi tarpeellisia. Hyviä perintätapayhdistelmiä ovat vain julkinen tai suojattu suojamääre virtuaalisen periytymistavan kanssa sekä yksityinen suojamääre ei-virtuaalisen periytymistavan kanssa. Toinen C++:n periytymisen ongelma on yksityisen periytymisen mukanaan tuoma läpikäymättömyys perintäpolkuun. C++:ssa on siis mahdollista syrjäyttää saavuttamattomissa oleva virtuaalinen operaatio, jota ei voi syrjäyttävästä luokasta kutsua. Toinen ongelma on moniperinnän nimikonfliktit, joihin ei ole mitään ratkaisumekanismia kielessä. Tämä voi johtaa hyvinkin hankaliin ratkaisuihin, niin kuin kolmannessa luvussa esitimme. Myös haarautuva moniperintä aiheuttaa ongelmia C++:ssa: jokin piirre voi olla saavutettavissa toista periytymispolkua pitkin, mutta toista polkua pitkin sitä ei voi kutsua. Haarautuvassa moniperinnässä C++:n hallitsevuussääntö määrää, mitä virtuaalista operaatiota kutsutaan, mikäli tämä operaatio on syrjäytetty usealla periytymispolulla.

Neljännessä luvussa käsittelimme moniperintää Eiffel-ohjelmointikielessä. Eiffelissä luokkien perintä määrätään perintälausekkeessa. Perintä on aina periaatteessa julkista ja virtuaalista. Näin Eiffelissä ei ole mahdollista piilottaa piirteitä luokan jälkeläisluokilta. Eiffelissä horisontaaliset nimikonfliktit voidaan ratkaista kätevästi uudelleennimeämisellä. Tällöin kahden samalla nimellä perityn yliluokan piirteen välinen nimikonflikti voidaan ratkaista helposti. Valitettavasti uudelleennimeäminen voi aiheuttaa lisää monimutkaisuutta perintähierarkiaan ja johtaa jopa valintamekanismin käyttöön dynaamisen monikäsitteisyyden poistamiseksi. Uudelleennimeäminen voi myös aiheuttaa haarautuvassa moniperinnässä perityn attribuutin monistumisen ja näin myös lisätä perintähierarkian monimutkaisuutta. Eiffelissä on lisäksi kovariantti syrjäytystapa. Näin operaatiot voidaan syrjäyttää operaatiolla, jonka kutsumuodon ei tarvitse olla identtinen syrjäytettävään nähden. Myös tämän todettiin aiheuttavan perinnässä ongelmatilanteita.

Viidennessä luvussa käsittelimme moniperintää Javassa. Javan moniperintä poikkeaakin huomattavasti kahdesta edellä esitellystä kielestä, johtuen sen hyvin tiukoista rajoitteista. Javassa moniperintä ei ole sallittua tavallisille luokille. Javassa vain puhtaita abstrakteja luokkia eli liittymäluokkia voidaan moniperiä. Tämä voimakas koodin perinnän rajoittaminen on johtanut Javassa jopa tavanomaista moniperintää monimutkaisempiin ratkaisuihin, esimerkiksi mahdollisuuteen käyttää rajoittamattomasti sisäkkäisiä luokkia, jopa moniperittävien liittymäluokkien sisällä. Näin Javassa voidaankin kiertää hieman yksittäisperiytymiseen rajoittunutta toiminnallisuuden eli ohjelmakoodin perintää. Javan liittymäluokkiin rajoittunut moniperintä aiheuttaa tavanomaisia moniperinnän ongelmia eli esimerkiksi nimikonflikteja liittymäluokkien ja näissä esiteltyjen piirteiden välille.

Tutkielman kuudennessa luvussa vertailimme moniperintää sekä sen ominaisuuksia ja ongelmia tutkielmamme kohteina olevissa kolmessa kielessä. Ensin vertailimme periytymistapaa, jolla yliluokat peritään kyseisessä kielessä luokkiin. C++:ssa havaittiin selkeästi monimutkaisin periytymistapavalikoima, jossa todettiin olevan jopa turhia periytymistapayhdistelmiä. C++ on myös ainoa vertailumme kieli, jossa on mahdollisuus yksityiseen periytymistapaan, joka havaittiin jossain tilanteissa – haitoistaan huolimatta – hyödylliseksi. Seuraavaksi käsiteltiin syrjäytystä, jonka mahdollistamiseksi C++:ssa operaatiot on määriteltävä eksplisiittisesti virtuaalisiksi. Eiffelissä ja Javassa virtuaalisuus on vastaavasti eksplisiittisesti estettävissä. C++:ssa ja Eiffelissä, joissa toteutusta voi moniperiä useilta välittömiltä yliluokilta, syntyy haarautuvassa moniperinnässä tilanteita, joissa operaation määräävä syrjäytys on ratkaistava jotenkin. Tässä tilanteessa käsiteltiin näiden kahden välinen ero: C++:ssa vallitsee hallitsevuussääntö, joka määrää hallitsevan syrjäytyksen. Eiffelissä on vastaavasti määrättävä eksplisiittisesti piirre valintamekanismilla. Moniperinnän nimikonflikteista todettiin, että vain Eiffelissä on tarjottu selkeä mekanismi nimikonfliktien ratkaisuksi: piirteiden uudelleennimeäminen. C++:ssa ja Javassa nimikonflikteihin ei ole tarjottu minkäänlaista varsinaista ratkaisukeinoa. Samanlaiset ongelmat vaivaavat siis näiden kummankin moniperintää, vaikka Javassa moniperinnän rajoittamisella onkin pyritty näitä ongelmia välttämään.

LÄHDELUETTELO

- Bracha, G., Cook, W. 1990. Mixin-based Inheritance. ACM OOPSLA Proceedings, October 1990. [online], [viitattu 27.12.2001]. Saatavilla www-muodossa <http://delivery.acm.org/10.1145/100000/97982/p303-bracha.pdf?key1=97982&key2=4116979001&coll=portal&dl=ACM&CFID=1119931&CFTOKEN=30718714>
- Cardelli, L. 1988. A Semantics of Multiple Inheritance. Information and Computation 76. 1988, 138 – 164.
- Colnet, D., Leonard, D., Masini, G., Napoli, A., Tombre, K. 1991. Object Oriented Languages. London: Academic Press Limited.
- Ellis, M., Stoustrup, B. 1997. The Annotated C++ Reference Manual. Reading, Massachusetts: Addison-Wesley.
- Gosling, J., Joy, B., Steele, G., Bracha, G. 2000. The Java Language Specification, 2nd ed. Reading, Massachusetts: Addison-Wesley.
- Gosling, J., K. Arnold. 1997. The Java Programming Language, 2nd ed. Reading, Massachusetts: Addison-Wesley.
- ISO/IEC 14882. 1998. Information Technology, Programming Languages – C++, ISO/IEC, 1998.
- Joyner, I. 1999. Objects Unencapsulated. Upper Saddle River, New Jersey: Prentice-Hall.
- Koskimies, K. 2000. Oliokirja. Jyväskylä: Gummerus Kirjapaino Oy.
- Meyer, B. 1996. The many faces of inheritance: A taxonomy of taxonomy. IEEE Computer, Vol. 29, No. 5, May 1996, 105 – 108.
- Meyer, B. 1997. Object-oriented software construction, 2nd ed. Upper Saddle River, New Jersey: Prentice-Hall.
- Sakkinen, M. 1989. Disciplined Inheritance. ECOOP'89, July 1989. Cambridge University Press, Cambridge, UK, 39-56.
- Sakkinen, M. 1992. A Critique of the Inheritance Principles of C++. USENIX Computing Systems, Vol. 5, No. 1, Winter 1992. 69 – 110.
- Snyder, A. 1986. Encapsulation and inheritance in object-oriented programming languages. ACM OOPSLA Proceedings, October 1986 [online],

[viitattu 4.5.2001]. Saatavilla www-muodossa <http://www.acm.org/pubs/articles/proceedings/oops/28697/p38-snyder/p38-snyder.pdf>

- Stroustrup, B. 1997. The C++ Programming Language, 3rd ed. Reading, Massachusetts: Addison-Wesley.
- Taivalsaari, A. 1996. On the notion of inheritance. ACM Computing Surveys, Vol. 28, No. 3, September 1996. 438 – 479.
- Wegner, P. 1987. Dimensions of object-based language design. ACM OOPSLA Proceedings, October 1987 [online], [viitattu 4.5.2001]. Saatavilla www-muodossa <http://www.acm.org/pubs/articles/proceedings/oops/38765/p168-wegner/p168-wegner.pdf>