Olli Kalmari

Moniperintä ja tyypitys olio-ohjelmointikielissä

Tietotekniikan
pro gradu -tutkielma
12. toukokuuta 2008

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Olli Kalmari
Yhteystiedot: olkakalm@jyu.fi
Työn nimi: Moniperintä ja tyypitys olio-ohjelmointikielissä
Title in English: Multiple Inheritance and Typing in Object-oriented Programming Languages
Työ: Tietotekniikan pro gradu -tutkielma
Sivumäärä: 58
Tiivistelmä: Moniperintä on merkittävä ongelmien aiheuttaja olio-ohjelmointikielissä. Tässä tutkielmassa tarkastellaan erityisesti moniperintään ja tyypitysmekanismeihin liittyviä vaikeuksia. Tutkimus on toteutettu vertailemalla viittä erilaista olio-ohjelmointikieltä.
English abstract: Multiple inheritance is a notable source of problems in object-oriented programming languages. This thesis discusses difficulties that are related specifically to multiple inheritance and typing mechanisms. The means of research is a comparison of five different programming languages.
Avainsanat: ohjelmointi, ohjelmointikielet, oliokeskeisyys, Common Lisp, Eiffel, Java, JavaScript, Smalltalk
Keywords: programming, programming languages, object-orientation, Common Lisp, Eiffel, Java, JavaScript, Smalltalk

Sisältö

1 Johdanto

Ohjelmoinnissa on esiintynyt vuosien varrella monenlaisia toimintatapoja ja paradigmoja, jotka ovat vaikuttaneet myös ohjelmointikielten kehitykseen. Tällä hetkellä ehkä suosituin ohjelmointimalli on niin kutsuttu oliokeskeinen ohjelmointi eli lyhyemmin olio-ohjelmointi. Sen avulla on onnistuttu helpottamaan tosimaailman rakenteiden mallintamista tietokoneohjelmissa ja parantamaan koodin uudelleenkäytettävyyttä.

Olio-ohjelmoinnissa on erittäin merkittävässä asemassa perinnäksi kutsuttu tekniikka, jossa tietorakenne ”perii” ominaisuuksia joltakin aiemmin määritellyltä tietorakenteelta. Eräs olio-ohjelmoinnin vaikeimmista ongelmista on moniperintä eli perinnän erikoistapaus, jossa ominaisuuksia peritään rinnakkain useammasta eri lähteestä. Tässä tutkielmassa tarkastellaan moniperintää tyypityksen näkökulmasta. Tyypityksellä tarkoitetaan tapaa, jolla ohjelmointikielessä käsitellään erilaisia tietotyyppejä. Tutkimuksen lähtökohtana on kysymys ”Onko dynaaminen tyypitys moniperinnän kannalta parempi kuin staattinen tyypitys?”

Tutkimus on toteutettu vertailemalla keskenään viittä eri olio-ohjelmointikieltä; nämä ovat Common Lisp Object System, Eiffel, Java, JavaScript ja Smalltalk. Jokaisessa kielessä on hieman erilainen lähestymistapa olio-ohjelmointiin, ja jokaisessa on myös erilainen perinnän ja tyypitysjärjestelmän yhdistelmä. Tutkielmassa näytetään, mitä hyviä ja huonoja puolia liittyy kuhunkin kokonaisuuteen.

Luvut 2 ja 3 toimivat perehdytyksenä aihealueeseen. Luvussa 2 kerrotaan olio-ohjelmoinnista yleisellä tasolla, ja luvussa 3 tutustutaan tarkemmin tyypityksen ja perinnän periaatteisiin. Tutkielman tärkein osa on luku 4, jossa tutkitaan tyypityksen ja perinnän yhdistelmiä vertailukielissä. Luvussa 5 esitetään vertailusta saatuja johtopäätöksiä.

2 Olio-ohjelmointi

Tässä luvussa määritellään olio-ohjelmointi ja esitellään joitakin avainkohtia sen historiasta. Lopuksi kerrotaan lyhyesti kustakin käsiteltävästä oliokielestä ja annetaan esimerkkejä niiden syntaksista.

2.1 Määritelmä

Olio-ohjelmoinnille on paljon hieman toisistaan poikkeavia määritelmiä. Yksi yleisimmistä on malli, jonka muun muassa Timothy Budd [4] esittää. Sen mukaan oliopohjaisessa ohjelmoinnissa ohjelma nähdään kokoelmana toisiinsa löyhästi kytkeytyneitä agentteja, joita sanotaan olioiksi. Jokainen olio on vastuussa jostakin tietystä tehtävästä, ja ohjelman suoritus perustuu olioiden väliseen kommunikointiin. Olio on abstrakti tietotyyppi, joka koostuu datasta ja dataa käsittelevistä operaatioista. Datamuuttujia kutsutaan yleisesti olion attribuuteiksi ja operaatioita sen metodeiksi.

Tämä määritelmä ei vielä riitä erottamaan olio-ohjelmointia modulaarisesta ohjelmoinnista. Esimerkiksi Ada, Modula ja CLU ovat kieliä, jotka tukevat tiedon abstrahointia mutta joita ei yleisesti pidetä oliokielinä. Peter Wegnerin esittämässä määritelmässä [32] oliokieliltä vaaditaan, että niiden tietoabstraktioilla on oltava tyypit ja että abstraktioita pystytään koostamaan muista tyypeistä perimällä[1]. Määritelmä voidaan esittää kaavana:

$$\text{oliosuuntautuneisuus} = \text{tiedon abstrahointi} + \text{abstraktit tietotyypit} + \text{tyyppien perintä}$$

Tämäkään määritelmä ei ole täydellinen. Esimerkiksi CLOS:n oliot eivät täysin vastaa abstraktin tietotyypin käsitettä eikä Selfissä tunneta varsinaista olioiden perintää (ks. esim. [8] ja [31]). Antero Taivalsaari esittää väitöskirjassaan [30] seuraavanlaisen, vähemmän rajoittavan määritelmän:

$$\text{olio-ohjelmointi} = \text{oliot} + \text{inkrementaalinen muokkaus}$$

Inkrementaalinen muokkaus on Taivalsaaren mukaan vähemmän rajoittunut perinnän muoto, tai toisin sanoen perintä on vain yksi inkrementaalisen muokkauksen

[1] Myös Budd on sitä mieltä, että juuri perintä erottaa oliokielet muista kielistä.

muoto. Olio puolestaan tarkoittaa tässä kokonaisuutta, johon kuuluu identiteetti, tila (muuttujat) ja käyttäytyminen (operaatiot).

Tässä tutkielmassa olio-ohjelmoinnilla tarkoitetaan ohjelmointimallia, jossa perintähierarkioita muodostavat oliot kommunikoivat keskenään ja muodostavat siten toiminnallisen järjestelmän. Peter Wegnerin määritelmä toimii tutkielman formaalina pohjana. Niissä kohdissa, joihin se ei sovi, turvaudutaan Taivalsaaren versioon.

2.2 Olio-ohjelmoinnin historia

2.2.1 1960-luku: Simula

Norjalaisten Ole-Johan Dahlin ja Kristen Nygaardin kehittämää Simula 67 -kieltä voidaan pitää ensimmäisenä olio-ohjelmointikielenä. Se oli tiettävästi ensimmäinen kieli, joka tuki luokkamäärittelyjä ja niiden perusteella luotavia olioita sekä luokkien perintää (josta käytettiin termiä *prefixing*). Simula 67 oli paranneltu ja yleiskäyttöisempi versio Dahlin ja Nygaardin Simula I -kielestä, joka oli nimensä mukaisesti tarkoitettu simulointiin[2]. Dahl ja Nygaard kertovat kielen historiasta vuoden 1978 artikkelissaan [24].

Simula I:n kehitys lähti liikkeelle vuonna 1961 Kristen Nygaardin aloitteesta. Nygaard työskenteli operaatioanalyysin[3] parissa ja havaitsi, että simulointi on ainoa käyttökelpoinen keino analysoida riittävän todenmukaisia toimintamalleja. Mallien kuvaamiseen ei kuitenkaan ollut sopivaa käsitteistöä eikä kieltä, eikä simulaatio-ohjelmien tekoon ollut sopivia työkaluja. Niinpä Nygaard päätti luoda uuden kielen, joka ratkaisisi mainitut ongelmat. Kielen suunnittelussa ja kääntäjän toteutuksessa häntä avusti Ole-Johan Dahl. Simula I julkaistiin vuonna 1965.

Simula I oli Algol 60 -kielen laajennus. Algol-ohjelma koostui lokaalia dataa käsittelevän operaatiosarjan sekä itse datan rakenteen määrittelyistä. Simula I:ssä voitiin koota yhteen monta tällaista kokonaisuutta eli *prosessia* ja saada ne toimimaan näennäisesti rinnakkain. Prosessin määrittelyosuutta kutsuttiin *aktiviteetiksi*. Prosessi suoritti operaationsa aktiivisessa vaiheessa (*active phase*), ja yhden prosessin siirtyessä aktiivisesta vaiheesta toiseen saattoi toteutua mielivaltaisen monta muiden prosessien aktiivista vaihetta. Aktiivisten vaiheiden suoritushetki ei siis ollut riippuvainen järjestelmän ajasta kuten Algol-ohjelmissa. Dahl ja Nygaard selvittävät

[2]Nimi Simula on lyhenne sanoista SIMUlation LAnguage.

[3]Matemaattisia menetelmiä, joilla tutkitaan mm. taloudellista, hallinnollista ja sotilaallista toimintaa.

asiaa Simula I -artikkelissaan [9].

Dahl ja Nygaard eivät olleet täysin tyytyväisiä Simula I:een ja päättivät tehdä kielestä uuden version. Rakenteista tehtiin yleiskäyttöisempiä ja kieleen lisättiin mahdollisuus käyttää vanhoja tietorakenteita uusien pohjana. Aktiviteetit nimettiin uudelleen luokiksi (*classes*) ja prosessit olioiksi (*objects*). Simula 67:n määrittely julkaistiin vuonna 1967, ja ensimmäinen toteutus valmistui vuonna 1969.

2.2.2 1960-luku ja 1970-luku: Alan Kay ja Smalltalk

Eräs tärkeä vaihe olio-ohjelmoinnin historiassa oli Smalltalk-kielen synty. Smalltalkin kehityksen aloitti amerikkalainen Alan Kay, joka keksi termin olio-ohjelmointi (*object-oriented programming*) ja esitteli ensimmäisenä uuden, olioiden väliseen vuorovaikutukseen perustuvan ohjelmointimallin. Kay selvittää kielen syntyyn johtaneita tapahtumia artikkelissaan [16]. Hänen mukaansa uuden mallin taustalla oli kaksi keskeistä tarvetta: oli löydettävä monimutkaisille järjestelmille parempi modulaarinen rakenne, joka mahdollistaisi yksityiskohtien piilottamisen, sekä tapa tehdä sijoituksia joustavammin ja lopulta päästä niistä kokonaan eroon.

Kay sai idean olio-ohjelmoinnista vuonna 1966, kun hän tutustui Simula I:een. Hän oli jo aiemmin törmännyt monta kertaa periaatteeseen, jossa yhdestä muotista luodaan monia itsenäisiä, yksilöllisiä ilmentymiä, mutta nyt hän näki sen ensimmäistä kertaa toteutettuna ohjelmointikielessä. Kay tiesi ennestään, että rekursiivisen suunnittelun periaatteena on tehdä osasista yhtä voimakkaita kuin kokonaisuudesta, ja hän oivalsi, että periaatetta voisi soveltaa ohjelmoinnissa uudella tavalla. Tietokonetta ei välttämättä kannattaisi jakaa heikompiin osakokonaisuuksiin — tietorakenteisiin ja proseduureihin — vaan moniin pieniin tietokoneisiin, jotka kaikki simuloisivat jotakin käyttökelpoista rakennetta. Nämä pienet tietokoneet voitaisiin luoda yhteisestä muotista samaan tapaan, kuin eri tehtäviin erikoistuneet solut luonnossa muodostuvat samanlaisista kantasoluista.

1960-luvun lähestyessä loppuaan Kay tuli siihen tulokseen, että ennen pitkää tietokoneista tulee ihmisten henkilökohtaisia tallennusvälineitä. Hän piti siksi tärkeänä, että tietokoneiden käyttöä alettaisiin opettaa lapsille jo melko varhaisessa vaiheessa. Aloitettuaan vuonna 1970 työt Xeroxin tutkimuskeskuksessa Palo Altossa hän alkoi suunnitella pientä tietokonetta ja siihen ohjelmointikieltä, jota lapset osaisivat käyttää. Hän antoi kielelle nimeksi Smalltalk.

Kayn työryhmä kehitti Smalltalkia vuodesta 1972 vuoteen 1980. Kieli kävi läpi monia muodonmuutoksia, ja ajatus lapsille sopivasta ohjelmointikielestä jäi vähitellen taka-alalle. Smalltalk sai vaikutteita muun muassa LISP-kielestä ja perin-

nän osalta myös Simula 67:stä. Kielen ympärille rakennettiin graafinen, ikkunoihin perustuva käyttöliittymä, jonka periaatteita myöhemmin hyödynnettiin Applen Macintoshissa. Smalltalk julkaistiin virallisesti vuonna 1983.

2.2.3 1980-luku: oliokielten lisääntyminen

1980-luvulla ilmestyi useita olio-ohjelmointikieliä, joista ehkä suosituimmaksi nousi C++. Sen kehitti tanskalainen Bjarne Stroustrup Bellin tutkimuskeskuksessa New Jerseyssä. Stroustrup tutki hajautettuja järjestelmiä ja piti Simulan ominaisuuksia erittäin sopivina hajautetun ohjelmiston simulointiin. Simula ei kuitenkaan ollut riittävän suorituskykyinen suurissa järjestelmissä. Vuonna 1979 Stroustrup alkoi suunnitella uutta työkalua, jolla voisi analysoida UNIX-ytimen hajautusmahdollisuuksia lähiverkossa. Hänen ratkaisunsa oli lisätä järjestelmäohjelmoinnissa suosittuun C-ohjelmointikieleen Simulan luokat. Tästä kehittyi uusi ohjelmointikieli nimeltä C with Classes, joka uudelleensuunnittelun jälkeen sai nimen C++. Ensimmäinen versio C++:sta ilmestyi vuonna 1983. Stroustrup kertoo kielen syntyvaiheista artikkelissaan [28].

Muita merkittäviä 1980-luvulla syntyneitä oliokieliä olivat muun muassa Self, CLOS ja Eiffel. Self otti vahvoja vaikutteita Smalltalkista (ks. esim. [31]), mutta se esitteli uusina piirteinä luokattomuuden ja prototyyppipohjaisen perinnän. CLOS eli Common Lisp Object System puolestaan lisäsi LISP-kielestä kehittyneeseen Common Lisp -murteeseen tuen olio-ohjelmoinnille (ks. [3]). Eiffelin suunnittelussa painotettiin oliosuuntautuneisuuden lisäksi ohjelmistotekniikan keskeisiä periaatteita, minkä johdosta kieleen sisältyi tuki esimerkiksi tiedon piilottamiselle, ohjelmien ominaisuuksien formaalille määrittelylle ja tiukoille nimeämiskäytännöille (lähteinä [21] ja [22]).

2.2.4 1990-luku ja 2000-luku: olio-ohjelmoinnin läpimurto

Olio-ohjelmointi nousi 1990-luvulla yhdeksi suosituimmista ohjelmointisuuntauksista. Internet ja graafiset käyttöliittymät saivat entistä tärkeämmän roolin ohjelmoinnin — myös olio-ohjelmoinnin — kehityksessä. Uusista kielistä ehkä suurimman suosion saavutti vuonna 1995 julkaistu Java. Se sai alkunsa Sun Microsystemsin sulautettuja järjestelmiä käsittelevässä tutkimusprojektissa, jossa tarvittiin ohjelmointiympäristö erilaisilla prosessoriarkkitehtuureilla toimivien laitteiden yhteistoimintaan verkossa. Kielen alkuperäinen suunnittelija, amerikkalainen James Gosling, tuli siihen tulokseen, että mikään olemassa olevista kielistä ei tarjonnut sopivaa nopeuden ja luotettavuuden yhdistelmää, ja päätti luoda uuden kielen. Tutkimus-

projektin ajatus kauko-ohjattavista kodinkoneista ei herättänyt kiinnostusta, mutta projektiryhmä havaitsi, että kehitetty teknologia soveltuisi hyvin myös Internetiin. Javan ansiosta staattisille HTML-sivuille pystyttiin nyt liittämään dynaamisia sovelmia. Myöhemmin Javan toiminta laajeni myös muille sovellusalueille. Javan synnystä kertovat muun muassa David Bank [2] ja Jon Byous [5].

World Wide Webin kehittymisen myötä merkittävään asemaan nousivat myös niin kutsutut komentosarjakielet (*scripting languages*), joiden käyttö yleistyi HTML-sivuilla. Monet niistä ovat oliosuuntautuneita, kuten esimerkiksi Selfin tyyliä seuraava prototyyppipohjainen JavaScript [11].

2.3 Vertailukielistä

2.3.1 Common Lisp Object System (CLOS)

CLOS on Common Lisp -kielen laajennus, joka lisää kieleen tuen olio-ohjelmoinnille. Common Lisp on Lisp-kielen murre, joka kehitettiin standardisoimaan sitä edeltävien Lisp-murteiden ominaisuudet. Se ei ole Lispin toteutus vaan määritelmä, jolle on useita toteutuksia. Nykyään CLOS on osa Common Lisp -standardia.

Common Lisp on yleiskäyttöinen ohjelmointikieli, joka tukee olio-ohjelmoinnin lisäksi myös funktionaalista ja imperatiivista ohjelmointiparadigmaa. Se on dynaamisesti tyypitetty, eli muuttujien tyypit ovat tiedossa vasta ajon aikana (ks. alaluku 3.1), mutta muuttuja voidaan myös asettaa vain yhden tyypin edustajaksi. Toteutuksesta riippuen Common Lisp voi olla joko tulkattava tai käännettävä kieli.

CLOS on luokkapohjainen järjestelmä. Seuraavassa esimerkissä luodaan **defclass**-makron avulla yksinkertainen luokka nimeltä person, jolla on attribuutit **name** ja **id**. Molemmille attribuuteille on määritetty argumentti, jolla käyttäjä voi alustaa attribuutin haluamaansa arvoon, sekä oletusarvo, joka asetetaan oliolle alustusargumentin puuttuessa. Ensimmäiselle attribuutille on määritetty lisäksi **:accessor**-määritteellä luku- ja kirjoitusfunktiot, joiden kautta attribuutin arvoon pääsee käsiksi.

```
(defclass person ()
  ((name
    :initarg :name
    :accessor name
    :initform "Unknown")
   (id
    :initarg :id
    :initform "010108")))
```

Seuraavassa asetetaan **p1**-muuttujan arvoksi uusi person-luokan ilmentymä, joka luodaan **make-instance**-makrolla:

```
(setf p1 (make-instance 'person :name "John" :id "190582"))
```

Yleensä luokkapohjaisissa oliokielissä metodikutsu osoitetaan aina tietylle oliolle, ja suoritettava koodi päätellään tuon olion luokasta. Tästä mallista käytetään usein nimeä viestinvälitys (*message passing*), koska toiminta voidaan ymmärtää niin, että oliot lähettävät toisilleen viestejä eli metodikutsuja. CLOS:ssa olioiden käyttäytyminen kuitenkin toteutetaan geneeristen funktioiden avulla [8]. Geneerinen funktio on abstrakti operaatio, jolla on nimi ja parametrilista muttei konkreettista toteutusta. Toteutus määritellään metodeissa. Jokainen metodi tarjoaa geneerisen funktion toteutuksen tietyille luokille, jotka esiintyvät funktiokutsun parametrilistassa. Suoritettavan koodin valinnasta ei siis ole vastuussa mikään luokka vaan geneerinen funktio. Seuraavassa määritellään geneerinen funktio **print** ja sen sisälle metodi, joka ottaa yhden **person**-tyyppisen parametrin.

```
(defgeneric print (printable-object)
  (:method ((printable-object person)) ... ))
```

Metodia kutsutaan antamalla haluttu olio sen parametriksi:

```
(print p1)
```

Tämä esittely perustuu CLOS:n standardiin [8] sekä Sonya E. Keenen kirjaan [17].

2.3.2 Eiffel

Eiffel on sekä ohjelmistonkehitysmenetelmä että ohjelmointikieli, joka kokonaisuudessaan on tarkoitettu ohjelmistojen analyysiin, suunnitteluun, toteutukseen ja ylläpitoon. Kieli on luokkapohjainen oliokieli, jonka suunnittelussa on pyritty siihen, että ohjelmistot olisivat uudelleenkäytettäviä, laajennettavia ja luotettavia. Uudelleenkäytettävyyttä haetaan laajoilla kirjastoilla ja uusien kirjastojen luontituella. Laajennettavuuden parantamiseksi kielessä on pyritty tekemään ohjelmistojen muokkaus mahdollisimman helpoksi. Luotettavuus on tarkoitus saavuttaa eri tekniikoilla, joihin kuuluvat staattinen tyypitys, tarkistusväittämät (*assertions*), kurinalainen poikkeusten käsittely ja automaattinen roskienkeruu. Tarkistusväittämillä tuetaan niin kutsuttua sopimuspohjaista suunnittelua (*design by contract*), jolla tarkoitetaan sitä, että ohjelmiston komponentit noudattavat tiettyjä ennalta sovittuja ehtoja.

Eiffelissä kaikki koodi kirjoitetaan luokkiin. Luokka koostuu piirteistä (*features*), jotka voivat olla joko attribuutteja tai metodeja eli rutiineja. Seuraavassa määritellään pankkitiliä kuvaava luokka ACCOUNT. Sillä on attribuutit **balance**, **owner** ja

minimum_balance sekä kuusi metodia. Viimeisten metodien edellä oleva määrite **NONE** ilmaisee, että metodit eivät näy olion ulkopuolelle.

```
class ACCOUNT create
  make
feature
  balance: INTEGER
  owner: PERSON
  minimum_balance: INTEGER

  open (who: PERSON)
      -- Asetetaan tilin omistajaksi who.
    do
      owner := who
    end

  deposit (sum: INTEGER)
      -- Talletetaan tilille sum-argumentin verran rahaa.
    require
      positive: sum > 0
    do
      add (sum)
    ensure
      deposited: balance = old balance + sum
    end

  withdraw (sum: INTEGER)
      -- Nostetaan tililtä sum-argumentin verran rahaa.
    require
      positive: sum > 0
      sufficient_funds: sum <= balance - minimum_balance
    do
      add (-sum)
    ensure
      withdrawn: balance = old balance - sum
    end

  may_withdraw (sum: INTEGER): BOOLEAN
      -- Onko tilillä tarpeeksi rahaa
      -- halutun summan nostamiseksi?
    do
      Result := (balance >= sum + minimum_balance)
    end

feature {NONE}
```

```
    add (sum: INTEGER)
      -- Lisätään sum-argumentti tilin saldoon.
    do
      balance := balance + sum
    end

  make (initial, min: INTEGER)
      -- Alustetaan tili alkusaldolla initial ja minimisaldolla min.
    require
      not_under_minimum: initial >= min
    do
      minimum_balance := min
      balance := initial
    ensure
      balance_initialized: balance = initial
      minimum_initialized: minimum_balance = min
    end

invariant
  sufficient_balance: balance >= minimum_balance

end -- ACCOUNT
```

Eiffelin tarkistusväittämät voivat olla metodien esi- tai jälkiehtoja tai luokkien invariantteja. Esiehdot määritetään avainsanalla **require**, jälkiehdot avainsanalla **ensure** ja invariantit avainsanalla **invariant**. Jos metodille on määritetty esiehto, jokaisen metodia kutsuva asiakasolion on täytettävä se. Jälkiehto puolestaan takaa kutsujille, että tietyt olosuhteet ovat voimassa kutsun jälkeen, jos esiehto täyttyi. Invariantti on ehto, jonka on oltava voimassa aina, kun luokan ilmentymää käsitellään ulkopuolelta. Tarkistusväittämien asettamien ehtojen rikkominen aiheuttaa ajonaikaisen virheen.

Uuden ACCOUNT-olion voi luoda seuraavasti:

```
create acc1.make (5500, 1000)
```

Metodi **make** on yksityinen, mutta **create**-kutsussa se sallitaan, koska kyse on olion alustuksesta. Kutsu onnistuu, koska se täyttää **make**-metodin esiehdon.

Eiffelin periaatteet selviävät muun muassa ECMA-standardista [12] ja Bertrand Meyerin kirjasta [22]. Koodiesimerkki on peräisin standardidokumentista.

2.3.3 Java

Java on Eiffelin tapaan yleiskäyttöinen, luokkapohjainen olio-ohjelmointikieli, jossa kaikki koodi kirjoitetaan luokkiin. Sen suunnittelussa on pyritty mahdollisimman hyvään alustariippumattomuuteen; tavoitteena on, että Java-ohjelma tarvitsee kirjoittaa vain kerran ja sen jälkeen sitä voi ajaa kaikkialla Internetissä. Yleensä Java-ohjelma käännetään tavukoodiksi, jota voidaan ajaa eri alustoille tehdyissä virtuaalikoneissa.

Edellinen pankkitililuokka voisi olla Javassa seuraavanlainen:

```
public class Account {

  private int balance;
  private Person owner;

  public Account(int initial) {
    if (initial >= 0)
      balance = initial;
  }

  public void open(Person who) {
    owner = who;
  }

  public void deposit(int sum) {
    if (sum >= 0)
      add(sum);
  }

  public void withdraw(int sum) {
    if (sum >= 0 && sum <= balance)
      add(-sum);
  }

  private void add(int sum) {
    balance = balance + sum;
  }

}
```

Uusi Account-olio luodaan seuraavasti:

```
Account myAccount = new Account();
```

Tämän kuvauksen lähteenä on Javan määrittely [15].

2.3.4 JavaScript

JavaScript on oliosuuntautunut komentosarjakieli (*scripting language*). Komentosarjakielellä tarkoitetaan kieltä, jota käytetään jonkin valmiin järjestelmän ominaisuuksien muokkaukseen ja automatisointiin. Tällaisissa järjestelmissä toiminnallisuuteen päästään käsiksi käyttöliittymän kautta, ja komentosarjakielen avulla tuota toiminnallisuutta voidaan käyttää ohjelmista käsin. Tällä tavoin järjestelmä tarjoaa komentosarjakielelle isäntäympäristön (*host environment*), jossa tämä voi toimia. JavaScriptille tyypillinen isäntäympäristö on WWW-selain.

JavaScript on Selfin tapaan oliokieli, jossa ei ole luokkia. Uusien olioiden luomiseen käytetään erityisiä konstruktorifunktioita (*constructors*), jotka varaavat olioille muistitilan ja alustavat niiden muuttujat. Kun olion jokin muuttuja asetetaan viittaamaan johonkin irralliseen funktioon, tuosta funktiosta tulee kyseisen olion metodi, jota voidaan siis kutsua olion kautta. Myös funktiot ovat JavaScriptissä olioita.

Seuraavassa määritellään konstruktorifunktio, jolla luodaan yksinkertainen laskuriolio.

```
function Counter() {
  this.count = 0;

  this.add = function() {
    this.count++;
  };

  this.reset = function() {
    this.count = 0;
  };

  this.toString = function() {
    return this.count;
  };
}
```

Uusi olio luodaan kutsumalla konstruktoria, minkä jälkeen oliolle voidaan antaa metodikutsu:

```
var myCounter = new Counter();
myCounter.add();
```

Olioon voidaan luomisen jälkeen myös lisätä uusia muuttujia, jotka eivät olleet mukana konstruktorissa:

```
myCounter.name = "Cars";
myCounter.subtract = function() {this.count--;};
```

JavaScriptin periaatteet on määritelty ECMAScript-standardissa [11]. Standardi pohjautuu useaan aiemmin toteutettuun teknologiaan, joista merkittävimmät ovat Netscapen kehittämä alkuperäinen JavaScript ja Microsoftin oma JavaScript-murre JScript. JavaScript ei vastaa standardia täydellisesti, mutta erot ovat hyvin pieniä.

2.3.5 Smalltalk

Smalltalk on yleiskäyttöinen, luokkapohjainen olio-ohjelmointikieli, jolla on paljon yhteistä CLOS:n, Eiffelin ja Javan kanssa. Monissa Smalltalkin toteutuksissa kieli ja ohjelmointiympäristö ovat melko tiukasti sidoksissa toisiinsa, eli luokkien ja niiden muodostamien sovellusten hallinta onnistuu ainoastaan käyttöliittymän kautta.

Useimmiten Smalltalk-ympäristö on tulkattu. Alkuperäisen Smalltalk-80:n toteutuksessa oli kaksi pääkomponenttia: virtuaalikuva (*virtual image*) ja virtuaalikone. Virtuaalikuvassa olivat kaikki järjestelmän oliot, ja virtuaalikone sisälsi kaikki laitteiston osat sekä konekieliset rutiinit, joiden avulla virtuaalikuvan oliot saatiin toimimaan. Käyttäjän kirjoittama ohjelmakoodi käännettiin tavukoodiksi, jonka virtuaalikone tulkkasi [14]. Smalltalkin ANSI-standardissa [1] ei kuitenkaan aseteta rajoja kielen toteutukselle.

Smalltalkille on useita eri toteutuksia, joilla on pieniä keskinäisiä eroja. Tässä tutkielmassa Smalltalk-esimerkit perustuvat etupäässä Squeak-ympäristöön, joka on Smalltalk-80:een perustuva avoimen lähdekoodin projekti. Smalltalkin alkuperäiset kehittäjät Alan Kay ja Dan Ingalls ovat olleet mukana myös Squeakin kehitystyössä.

Määritellään yksinkertainen pankkitililuokka Smalltalkilla.

```
Object subclass: #BankAccount
  instanceVariableNames: 'balance'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Bank'
```

Eräs Smalltalkin perusajatuksista on: "Everything is an object." (Kaikki asiat ovat olioita.) On huomattava, että edellä oleva luokan esittely on itse asiassa Object-luokkaa edustavalle oliolle lähetetty metodikutsu **subclass**, jolle annetaan parametreina muun muassa luokan attribuutit ja kategoria, johon luokka kuuluu.

Luokkaan voidaan lisätä metodi **balance**, joka palauttaa samanimisen attribuutin arvon:

```
balance
  ↑balance
```

Smalltalkin luokissa on yleensä alustusta varten metodi nimeltä **initialize**. Seuraavassa alustetaan **balance**-attribuutti arvoon 0:

```
initialize
  balance ← 0.
```

Tilille voidaan lisätä rahaa kutsumalla **deposit**-metodia, jolle annetaan parametrina lisättävä summa:

```
deposit: amount
  balance ← balance + amount.
```

Vastaavasti rahan nostamiseen on metodi **withdraw**:

```
withdraw: amount
  amount > balance ifTrue: [
    ↑ self inform: 'Not enough funds'].
    balance ← balance − amount.
```

Uusi BankAccount-olio luodaan kutsumalla BankAccount-luokkaa edustavan olion metodia **new**:

```
myAccount ← BankAccount new.
```

3 Tyypitys ja perintä

Tässä luvussa kerrotaan yleisesti tyypityksestä ja perinnästä. Luvussa tarkastellaan kummankin käsitteen osalta myös yleisen tason toteutusperiaatteita ja merkittävimpiä ongelmakohtia.

3.1 Tyypitys

3.1.1 Staattinen ja dynaaminen tyypitys

Tyypityksellä tarkoitetaan ohjelmoinnissa sitä, että tietokoneen muistissa oleville bittijonoille määritetään rakenne, jonka mukaan niitä tulkitaan. Ensimmäisillä ohjelmointikielillä tehtiin vain numeerista laskentaa, joten niissä oli vain yksi aritmeettinen tietotyyppi. Kielten kehittyessä kuitenkin havaittiin, että arvojen jaottelu kokonaislukuihin, liukulukuihin ja totuusarvoihin selkeyttää ohjelmointia. Vähitellen mukaan tulivat taulukot, osoittimet, merkkijonot ja muut tyypit. Simula esitteli ensimmäisenä oliotietotyypin.

Luca Cardelli ja Peter Wegner luonnehtivat artikkelissaan [6] tyypitystä vaatekerrokseksi tai panssariksi, joka suojaa allaan olevaa tyypittämätöntä dataa väärinkäytöksiltä. Tyypitys piilottaa datan todellisen esitysmuodon ja rajoittaa sitä, miten tietorakenteet voivat olla keskenään vuorovaikutuksessa. Tyypittämättömässä järjestelmässä tietorakenteet ovat ”alastomia” siinä mielessä, että esitysmuoto on kaikkien nähtävillä. Tyyppijärjestelmän väärinkäytöksissä suojaava panssari poistetaan ja dataa käsitellään ilman mitään rajoituksia.

Tietyn tietotyypin edustajilla on esitysmuoto, joka tukee tyypin oletettuja ominaisuuksia. Esitysmuoto valitaan siten, että tietotyypin edustajille on helppoa suorittaa haluttuja operaatioita. Tyyppijärjestelmän rikkominen sallii datan muokkaamisen eri tavalla kuin alun perin oli tarkoitettu, mahdollisesti tuhoisin seurauksin. Cardelli ja Wegner mainitsevat esimerkkinä, että kokonaisluvun tulkitseminen osoittimeksi mahdollistaa täysin mielivaltaiset muutokset ohjelmiin ja dataan.

Tyyppirikkomuksia voidaan estää määräämällä ohjelmille staattinen tyyppirakenne. Tyyppien avulla voidaan erotella vakiot, operaattorit, muuttujat ja funktiosymbolit. Automaattista tyyppien päättelyä (*type inference*) voidaan käyttää lausekkeiden tyypin tunnistamiseen silloin, kun tyyppitietoa ei erikseen niissä mainita.

Ohjelmointikieliä, joissa jokaisen lausekkeen tyyppi voidaan päätellä ohjelman staattisesta rakenteesta, sanotaan staattisesti tyypitetyiksi. Vastaavasti kielet, joissa tyypit selviävät vasta ajonaikaisessa analyysissa, ovat dynaamisesti tyypitettyjä. Staattinen tyyppijärjestelmä hylkää käännösvaiheessa sellaiset ohjelmat, jotka eivät täytä tiettyjä syntaktisia vaatimuksia, joiden perusteella ohjelma voidaan todeta lailliseksi. Dynaaminen järjestelmä puolestaan hyväksyy kaikki ohjelmat, ja jos jonkin ohjelmassa olevan operaation argumentit eivät ole hyväksyttäviä, operaatio aiheuttaa ajonaikaisen poikkeuksen. Staattisesti tai dynaamisesti tyypitetty kieli on lisäksi *vahvasti tyypitetty*, jos sen kaikki lausekkeet ovat tyyppien suhteen ristiriidattomia. Vahvasti tyypitetyn kielen kääntäjä pystyy takaamaan, että se hyväksyy vain sellaisia ohjelmia, jotka eivät aiheuta tyyppivirheitä.

Robert Cartwright ja Mike Fagan esittelevät artikkelissaan [7] joitakin staattisen ja dynaamisen tyypityksen etuja ja haittoja. Staattisella järjestelmällä on heidän mukaansa kaksi merkittävää vahvuutta:

Virheiden havaittavuus. Kääntäjä tunnistaa ja merkitsee sellaiset lauseet, jotka todennäköisesti aiheuttavat tyyppirikkomuksen. Tämä auttaa ohjelmoijaa korjaamaan ohjelmassa olevia virheitä jo ohjelmointiprosessin alkuvaiheessa.

Optimointi. Kääntäjä pystyy tuottamaan tehokkaampaa koodia, koska suuri osa ajonaikaisista tyyppitarkistuksista voidaan jättää pois.

Näiden vahvuuksien takia kuitenkin menetetään seuraavia ominaisuuksia:

Ilmaisuvoima. Staattinen järjestelmä joutuu hylkäämään tyyppiturvallisuuden takaamiseksi sellaisia ohjelmia, jotka ovat semanttisesti laillisia mutta joiden syntaksi ei täytä tyyppijärjestelmän vaatimuksia.

Yleiskäyttöisyys. Dynaamisessa järjestelmässä on mahdollista kirjoittaa abstrakteja operaatioita, jotka toimivat usealle eri tietorakenteelle. Staattisessa järjestelmässä ohjelmoija voi joutua tekemään samasta abstraktiosta monta toteutusta, koska yleisessä versiossa ei ole tyyppitarkistusta.

Semanttinen selkeys. Tyyppijärjestelmä on laaja kokoelma syntaktisia sääntöjä, jotka ohjelmoijan on hallittava pystyäkseen kirjoittamaan toimivia ohjelmia.

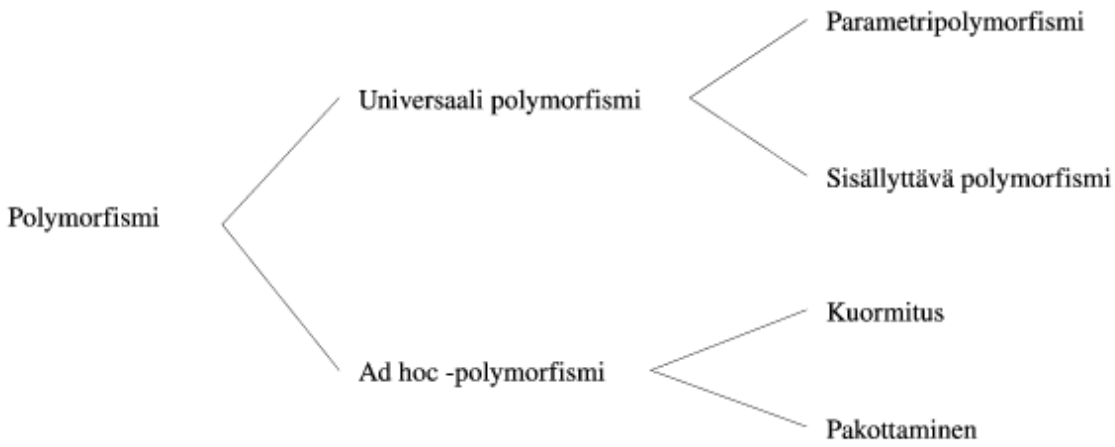
Bertrand Meyer [23] pitää myös virheiden havaittavuutta ja koodin tehokkuutta staattisen tyypityksen vahvuuksina. Lisäksi hän on sitä mieltä, että staattisesti tyypitetty kieli on ihmiselle helpompaa lukea, mikä on ohjelmiston ylläpidettävyyden kannalta tärkeä asia. Meyer kuitenkin toteaa, että mikäli luettavuutta ei tarvittaisi,

staattisen tyypityksen muut edut voitaisiin osittain saavuttaa automaattisella tyyppien päättelyllä.

3.1.2 Polymorfismi

Tyypitetyt ohjelmointikielet voidaan jakaa monomorfisiin ja polymorfisiin kieliin. Monomorfisissa kielissä funktioilla ja proseduureilla, ja siten myös niiden parametreilla, on yksikäsitteinen tyyppi, ja jokainen arvo ja muuttuja voidaan tulkita tasan yhden tyypin edustajaksi. Polymorfisissa kielissä taas joillakin arvoilla ja muuttujilla voi olla useampi kuin yksi tyyppi. Polymorfinen funktio on funktio, jonka todelliset parametrit voivat olla useaa tyyppiä. Polymorfinen tyyppi on tyyppi, jonka operaatioita voidaan soveltaa useaa eri tyyppiä edustaviin operandeihin.

Cardelli ja Wegner esittävät artikkelissaan polymorfismille luokittelun, joka perustuu Christopher Stracheyn vuonna 1967 esittämään jakoon [29]. Cardellin ja Wegnerin luokittelu on esitetty kuvassa 3.1.



Kuva 3.1: Polymorfismin eri lajit

Strachey jakoi alun perin polymorfismin kahteen ryhmään: parametripolymorfismiin (*parametric polymorphism*) ja ad hoc -polymorfismiin. Parametripolymorfismissa funktio toimii yksikäsitteisesti useille tyypeille; tyypeillä on yleensä jossakin määrin yhtenäinen rakenne. Ad hoc -polymorfismissa puolestaan funktio toimii — tai näyttää toimivan — useille tyypeille, mutta toiminta voi olla eri tyyppien tapauksessa erilaista eikä sitä pystytä päättelemään systemaattisesti funktion parametreis-

ta[1].

Cardelli ja Wegner lisäävät Stracheyn kahtiajakoon yhden uuden kategorian, sisällyttävän polymorfismin (*inclusion polymorphism*). Parametripolymorfismi ja sisällyttävä polymorfismi ovat alakategorioita yleisemmälle käsitteelle, jota Cardelli ja Wegner kutsuvat universaaliksi polymorfismiksi. Universaalisti polymorfinen funktio toimii yleensä äärettömälle määrälle tyyppejä, joilla on yhtenäinen rakenne, kun taas sen vastakohta eli ad hoc -polymorfinen funktio toimii vain rajalliselle määrälle tyyppejä, jotka eivät välttämättä liity toisiinsa millään tavalla. Universaalin polymorfismin tapauksessa voidaan sanoa varmuudella, että joillakin arvoilla (eli polymorfisilla funktioilla) on monta tyyppiä. Ad hoc -polymorfismissa tämä ei ole niin selvää, koska voidaan ajatella, että ad hoc -polymorfinen funktio on itse asiassa vain pieni joukko monomorfisia funktioita. Toteutusnäkökulmasta katsottuna universaalisti polymorfinen funktio suorittaa saman koodin mille tahansa sallittuja tyyppejä edustaville parametreille, mutta ad hoc -polymorfinen funktio voi suorittaa eri koodin eri parametrityypeille.

Universaalissa polymorfismissa arvolla voi olla monta tyyppiä pääosin kahdella eri tavalla. Parametripolymorfismissa polymorfisella funktiolla on implisiittinen tai eksplisiittinen tyyppiparametri, joka määrittää funktiolle annetun käsiteltävän parametrin tyypin jokaisessa funktiokutsussa [2]. Sisällyttävässä polymorfismissa tietyn alkion voidaan ajatella kuuluvan moneen eri joukkoon, jotka eivät välttämättä ole erillisiä. Cardelli ja Wegner toteavat, että nämä kaksi näkökulmaa ovat yhteydessä toisiinsa mutta niissä on silti sen verran eroavaisuuksia, että niille on hyvä antaa omat nimet.

Cardelli ja Wegner jakavat myös ad hoc -polymorfismin kahteen eri osaan. Kuormituksessa (*overloading*) samaa muuttujannimeä käytetään monelle eri funktiolle ja kontekstista päätellään, mitä funktiota eri kutsuissa tarkoitetaan. Kuormitus on periaatteessa pelkästään syntaktinen lyhennyskeino, koska se voidaan käsitellä ohjelman esiprosessoinnissa antamalla kuormitetuille funktioille eri nimet. Pakottaminen (*coercion*) on sen sijaan semanttinen operaatio, jolla muunnetaan funktion hyväksymään muotoon sellainen parametri, joka ilman muunnosta aiheuttaisi tyyppivirheen. Pakottamiset voidaan lisätä ohjelmaan parametrien ja funktioiden väliin käännösvaiheessa, mutta ne voidaan joutua tekemään myös ajon aikana dynaamisesti.

Kuormituksen ja pakottamisen välinen ero hämärtyy monissa tilanteissa. Tä-

[1] Ad hoc: (lat.) tätä tarkoitusta varten, tähän tehtävään, tilapäisesti.

[2] Cardelli ja Wegner käyttävät parametripolymorfismin toteuttavasta funktiosta myös nimitystä geneerinen funktio.

mä pätee erityisesti dynaamisesti tyypitettyihin ja tulkattuihin kieliin, mutta myös staattisesti tyypitetyssä kielessä esiintyy monitulkintaisuuksia. Cardelli ja Wegner esittävät seuraavan esimerkin:

```
3 + 4
3.0 + 4
3 + 4.0
3.0 + 4.0
```

Tässä yhteenlaskuoperaattori voidaan selittää kolmella eri tavalla:

- Yhteenlaskuoperaattorilla on neljä kuormitettua merkitystä, yksi kullekin parametrityyppien yhdistelmälle.
- Yhteenlaskuoperaattorilla on kaksi kuormitettua merkitystä: kokonaislukujen yhteenlasku ja reaalilukujen yhteenlasku. Jos yksi parametreista on kokonaisluku ja toinen reaaliluku, kokonaisluku pakotetaan reaaliluvuksi.
- Yhteenlaskuoperaattori on määritelty vain reaaliluvuille, ja kokonaisluvut pakotetaan aina reaaliluvuiksi.

Sama lauseke voi siis toteutuksesta riippuen sisältää kuormitusta, pakottamista tai molempia.

3.1.3 Alityypitys ja arvon jakaminen

Cardelli ja Wegner luettelevat neljä tekniikkaa, joilla perinteisisiin, monomorfisiin kieliin on lisätty polymorfisia piirteitä ja näin pyritty parantamaan käytettävyyttä:

1. Kuormitus. Kokonaislukuvakiot voivat olla sekä kokonaisluku- että reaalilukutyyppiä. Operaattorit kuten ”+” toimivat sekä kokonaisluku- että reaalilukuparametreille.
2. Pakottaminen. Kokonaislukuarvoa voidaan käyttää reaaliluvuille tarkoitetuissa operaatioissa ja päin vastoin.
3. Alityypitys. Tietyn erikoistyypin edustajat ovat myös yleistetymmän tyypin edustajia.
4. Arvon jakaminen. Pascal-kielessä **nil** on vakio, jonka kaikki osoitintyypit jakavat.

Alityypitys (*subtyping*) on sisällyttävän polymorfismin ilmentymä. Ajatus, että tyyppi voi olla toisen tyypin alityyppi, on Cardellin ja Wegnerin mukaan hyödyllinen silloin, kun käsitellään järjestettävien tyyppien osajoukkoja, kuten kokonaislukuja. He toteavat, että periaate toimii myös monimutkaisemmilla rakenteilla: voidaan kuvitella esimerkiksi tyyppi Toyota, joka on yleisemmän ajoneuvotyypin alityyppi. Jokaista alityyppiä voidaan käyttää ylityypin tavoin, mikä ajoneuvoesimerkin yhteydessä tarkoittaa, että jokainen Toyota on ajoneuvo ja Toyotalle sopivat kaikki ajoneuvojen yleiset operaatiot.

Arvon jakaminen (*value sharing*) on parametripolymorfismin erikoistapaus. Voitaisiin ajatella, että **nil**-symboli on raskaasti kuormitettu, mutta se ei ole luontevaa, koska **nil** on validi edustaja äärettömälle määrälle tyyppejä, joita ei ole edes vielä esitelty. Lisäksi kaikissa **nil**-symbolin käyttöyhteyksissä viitataan samaan arvoon, mikä ei ole yleisin tapaus kuormituksessa. Voitaisiin ajatella myös, että jokaiselle tyypille on oma **nil**, mutta kaikilla niistä on sama esitysmuoto ja identiteetti. Se, että tietorakenteella on monta tyyppiä ja sen esitysmuoto on kaikille tyypeille sama, on luonteenomaista parametripolymorfismille.

Kuormitusta ja pakottamista käsiteltiin jo alaluvussa 3.1.2.

3.2 Perintä

3.2.1 Yleistä perinnästä

Perintää (*inheritance*) pidetään yleisesti olio-ohjelmoinnin luonteenomaisimpana piirteenä ja suurimpana vahvuutena muihin ohjelmointimalleihin nähden. Perinnän perusajatus on, että uusia oliomäärittelyjä voidaan tehdä käyttäen pohjana aiempia määrittelyjä. Uutta olioluokkaa määritettäessä tarvitsee kertoa vain ne piirteet, jotka poikkeavat tiettyjen aiemmin määriteltyjen luokkien piirteistä. Kaikki vanhat piirteet liitetään automaattisesti uuteen luokkaan, eikä niitä tarvitse enää määritellä uudestaan. Halutessaan ohjelmoija voi kuitenkin määritellä uudelleen tai joskus jopa poistaa joitakin perittyjä piirteitä.

Perintää voi ajatella monesta eri näkökulmasta, ja sen voi määritellä monella eri tavalla, kuten esimerkiksi Antero Taivalsaari osoittaa väitöskirjassaan [30]. Useimmiten perintää pidetään käsitteellisenä yleistys-erikoistussuhteena, jolloin se rinnastetaan alityypitykseen (alaluku 3.1.3). Sen voi kuitenkin mieltää myös alhaisen tason mekanismiksi, jonka tavoitteena on koodin uudelleenkäytettävyyden parantaminen. Nämä kaksi ajattelutapaa saattavat varsinkin moniperinnässä olla ristiriidassa keskenään.

3.2.2 Perintä ja alioliot

Voidaan ajatella, että perinnän johdosta syntyy kerroksittaisia tai sisäkkäisiä olioita. Jos oletetaan, että luokasta A on peritty luokka B, niin A:n piirteistä muodostuu B-olion sisälle pienempi A-olio, jota kutsutaan alioliokseksi (*subobject*). Tällöin B-olio on vastaavasti A-olion yliolio (*superobject*). Aliolioita voi olla monella eri tasolla; yhden tason tapauksissa puhutaan myös välittömistä (*immediate*) ali- ja yliolioista. Markku Sakkinen [26] toteaa, että perintäpuun juuriluokan ilmentymä ei koskaan ole yliolio eikä lehtiluokan ilmentymä koskaan aliolio.

Sakkinen asettaa aliolioihin perustuvan perintämallin vastakkain toisen mallin kanssa, jota hän nimittää attribuuttikeskeiseksi. Hän havainnollistaa mallien eroja esimerkillä, jossa yli- ja aliluokassa määritellään samanniminen attribuutti. Aliolio-mallissa näiden attribuuttien samannimisyys on epäoleellista, koska ne kuuluvat eri aliolioihin ja ovat siksi täysin eri attribuutteja. Attribuuttikeskeisessä mallissa puolestaan katsotaan, että sama nimi tarkoittaa myös samaa attribuuttia. Tällöin aliluokassa esiintyvällä määrittelyllä voi olla kolme erilaista vaikutusta[3]:

1. Jos attribuutin tyyppi on aliluokassa sama kuin yliluokassakin, määrittely on merkityksetön mutta yleensä sallittu.

2. Joissakin kielissä aliluokassa annettu attribuutin tyyppi saa olla yliluokassa annetun tyypin alityyppi (jälkeläisluokka); tuloksena on tyypin uudelleenmäärittely.

3. Muissa tapauksissa määrittely on kelpaamaton.

3.2.3 Moniperintä

Kun olioluokka perii ominaisuuksia useammalta kuin yhdeltä välittömältä vanhemmalta, puhutaan moniperinnästä (*multiple inheritance*). Yksittäisperinnässä luokkahierarkia on puu, mutta moniperinnässä se laajenee suunnatuksi asykliseksi verkoksi. Moniperintä on oliokielessä tärkeä ominaisuus uudelleenkäytettävyyden ja tosimaailman rakenteiden mallintamisen kannalta, mutta sen järkevä toteutus on ongelmallista ja se tekee luokkahierarkioista sekavampia. Moniperinnästä kertovat muun muassa Ghan Bir Singh [27] ja Taivalsaari [30].

Singhin mukaan moniperinnällä on neljä pääasiallista käyttötarkoitusta:

Riippumattomien protokollien yhdistäminen. Joskus on tarvetta luoda uusi luokka, jolla on monen eri luokkaryhmän ominaisuuksia. Esimerkiksi Eiffelissä kir-

[3]Nämä vaikutukset koskevat vain staattisesti tyypitettyjä kieliä.

jastoluokalla WINDOW on kolme eri yliluokkaa: SCREENMAN, RECTANGLE ja TWO_WAY_TREE.

Kokoaminen (*mix and match*). Monta pientä yliluokkaa, jotka eivät yksinään ole käyttökelpoisia, yhdistetään moniperinnän avulla yhdeksi toimivaksi kokonaisuudeksi. Tällaisia osaluokkia kutsutaan usein sekoiteluokiksi (*mixin classes*).

Alimodulaarisuus (*submodularity*). Järjestelmää luotaessa havaitaan, että sen osaset ovat modulaarisia ja niiden yhdistämisellä voidaan parantaa järjestelmän rakennetta. Singh mainitsee esimerkkinä, että pankkisovelluksessa on järkevää yhdistää erilaiset korkojen maksuun liittyvät korkotyypit erillisessä luokassa, jonka SAVING-luokka sitten perii.

Rajapinnan ja toteutuksen erottaminen. Abstrakti luokka määrittelee oliolle rajapinnan, ja toinen, konkreettinen luokka määrittelee toteutuksen.

Moniperintää voi osittain jäljitellä koostamalla. Jos halutaan, että luokka käyttäytyy kuten kaksi erillistä luokkaa, sille voidaan asettaa kaksi attribuuttia, jotka viittaavat noihin kahteen luokkaan. Sen jälkeen luokkaan voidaan lisätä osaluokkia vastaavat metodit, joiden käsittely ohjataan sille osaoliolle, jossa kunkin metodin varsinainen toteutus on. Näin saadaan vaikutelma, että luokka olisi kahden osaluokan jälkeläinen. Ratkaisu ei kuitenkaan riitä korvaamaan puhdasta moniperintää, koska ohjelmoija joutuu kirjoittamaan ylimääräistä koodia saadakseen saman lopputuloksen. Lisäksi osaluokkien rajapinnoissa tapahtuvat muutokset joudutaan erikseen päivittämään koosteluokkaan.

3.2.4 Moniperinnän ongelmia

Singh [27] esittelee seuraavia yleisiä moniperinnän ongelmia:

Nimikonflikti. Jos kahdella tai useammalla yliluokalla on samanniminen piirre, aliluokassa ei pystytä päättelemään, mihin piirteeseen kyseisellä nimellä viitataan. Ongelma ei ole käsitteellinen, vaan se liittyy syntaksiin.

Toistuva perintä. Koska luokkahierarkia on verkko, sama yliluokka voi periytyä yhteen luokkaan kahta tai useampaa eri reittiä. Näin oliolla voi olla monta saman luokan alioliota[4].

[4]Sakkinen [25] nimittää tällaista tilannetta haarautuvaksi moniperinnäksi (*fork-join inheritance*). Toistuva perintä voi olla muunkinlaista.

Alustusmetodin valinta. Yliluokilla voi olla samannimisiä alustusmetodeja, joita kaikkia täytyy kutsua. Aliluokassa on tällöin pystyttävä erottamaan yliluokkien samannimiset metodit toisistaan.

Toteutus. Moniperinnän toteutus on käsitteellisesti vaikeaa, koska perityt luokat voidaan yhdistää monella eri tavalla. Moniperintää tukevissa kielissä on päädytty erilaisiin ratkaisuihin vaihtelevalla menestyksellä.

Väärinkäyttö. Perintä heikentää tiedon suojausta sallimalla pääsyn perittyjen luokkien yksityisiin ominaisuuksiin[5]. Moniperinnässä tämä korostuu, koska luokkia voidaan yhdistellä lähes rajattomasti. Niinpä moniperintää saatetaan käyttää pelkkänä suojauksenkiertomekanismina.

3.2.5 Moniperinnän semantiikkaa ja toteutusperiaatteita

Singhin mukaan moniperinnälle on kolme merkittävää toteutusperiaatetta, joita käytetään yleisesti oliokielissä: verkkototeutus (*graph implementation*), lineaarinen toteutus ja puutoteutus. Nimillä viitataan tapaan, jolla luokkahierarkia esitetään järjestelmässä. Näiden yleisen tason periaatteiden lisäksi kielissä on myös muita mekanismeja, joiden avulla voidaan välttää moniperinnän ongelmatilanteet.

Verkkototeutuksessa kieli mallintaa luokkahierarkiaverkon sellaisenaan. Ominaisuudet peritään ylhäältä alaspäin alkaen hierarkiassa korkeimmalla olevasta yliluokasta. Jos luokka perii samannimisen piirteen useammalta vanhemmalta, syntyy ristiriita, joka on ratkaistava jollakin tavalla. Ongelmaa ei ole, jos samannimiset piirteet ovat identtisiä, mutta muussa tapauksessa ohjelmoijan on muokattava jälkeläisluokkaa esimerkiksi nimeämällä uudelleen risteävät ominaisuudet — jos kielessä on sellainen mahdollisuus. Haarautuvan moniperinnän tapaus aiheuttaa toteutuksessa hankaluuksia, koska perittyihin attribuutteihin viitattaessa tarvitaan ajonaikaista osoitteenlaskentaa. Lisäksi on huolehdittava siitä, että tietty yhteinen aliolio alustetaan täsmälleen kerran.

Lineaarisessa toteutuksessa perintäverkko muunnetaan lineaariseksi poluksi, jolla ei ole kaksoisesiintymiä, ja lopputulosta käsitellään yksittäisperintänä. Luokat sijoitetaan polulle siinä järjestyksessä kuin ne on esitelty jälkeläisluokassa riippumatta siitä, oliko polun keskelle tulevilla luokilla vanhempia alkuperäisessä hierarkiassa. Tästä seuraa, että muutos luokkien esittelyssä saattaa muuttaa perintähierarkian perusteellisesti. On myös mahdollista, että väärän perimisjärjestyksen takia jotkin

[5]Singh ei tarkenna, onko kysymys yleisesti yksityisistä vai C++:n ja Javan tapaan yksityisistä ominaisuuksista. Ominaisuuksien näkyvyysmääritteitä käsitellään Javan yhteydessä alaluvussa 4.1.2.

tarpeelliset ominaisuudet jäävät pois jälkeläisluokasta. Hyötynä lineaarisessa toteutuksessa on toisaalta se, että toistuvaa perintää ei esiinny lainkaan.

Puutoteutuksessa on pyritty välttämään verkkototeutuksessa ja lineaarisessa toteutuksessa esiintyviä ongelmia. Perintäverkko muunnetaan puuksi monistamalla sellaiset yhteiset yliluokat, joihin voidaan päätyä montaa eri reittiä pitkin. Tuloksena on puurakenne, jossa joillakin luokilla on samoja ominaisuuksia. Toteutus on melko selkeä, ja esimerkiksi C++:ssa se on oletustapa[6], mutta se aiheuttaa ongelmia perinnän käsitteelliselle merkitykselle, erityisesti alityyppisuhteiden kannalta. Oletetaan, että luokka A on luokkien B ja C yliluokka ja että luokka D perii B:n ja C:n. Tällöin D-oliossa on alioliomallin mukaisesti kaksi erillistä A-alioliota, koska B:llä ja C:llä on kummallakin omansa. Tämä ei sovellu perinnän käsitteelliseen merkitykseen, kuten Markku Sakkinen [25], [26] osoittaa. Kun D on A:n jälkeläisluokka, D-olion pitäisi olla myös ”jonkinlainen A-olio”. Tähän ei sovi rakenne, jossa D-olio sisältää kaksi tasavertaisessa asemassa olevaa A-alioliota. Käytännön tasollakin tulee heti ongelma, kun D-oliossa pitää käsitellä A:sta perittyä attribuuttia tai kutsua A:sta perittyä metodia: ei ole selvää, kumpaa alioliota tämä koskee.

Joissakin kielissä, kuten C++:ssa ja Eiffelissä, voi yhdistellä verkko- ja puutoteutusta moniperintätilanteissa melko mielivaltaisesti. Tuloksena voi syntyä hyvin hankalia ja epäloogisia rakenteita; Sakkinen [26] esittää C++-tapauksen, jossa oliolla on kaksi keskenään samanlaista alioliota, jotka puolestaan jakavat vielä yhden aliolion. Sakkisen mukaan C++:ssa ja muissakin moniperintää tukevissa kielissä olisi parempi, jos julkista haarautuvaa moniperintää ei sallittaisi lainkaan vaan se rajoitettaisiin ainoastaan yksityiselle tasolle eli toteutukseen. Tällöin moniperinnän mahdollistama koodin uudelleenkäytettävyys säilyisi mutta hankalat alityyppisuhteet ja alioliioiden aiheuttamat toteutusongelmat vähenisivät. On kuitenkin huomattava, että kielissä, joissa kaikilla luokilla on yhteinen juuriluokka, kaikki moniperintä on haarautuvaa. Niissä haarautuvan perinnän kieltäminen saattaisi siis olla liian rajoittava ratkaisu.

[6]Jakava perintä (verkkototeutus) saadaan C++:ssa aikaan käyttämällä luokkamäärittelyissä yliluokkien yhteydessä **virtual**-avainsanaa.

4 Tyypityksen ja perinnän yhdistelmä vertailukielissä

Tässä luvussa esitellään, millainen tyypitysjärjestelmä ja perintämekanismi kussakin vertailtavaksi valitussa kielessä on, ja tarkastellaan, mitä vahvuuksia ja heikkouksia kuhunkin yhdistelmään liittyy.

4.1 Java

4.1.1 Tyypit ja alityypit

Java on staattisesti tyypitetty kieli, jossa tyypit jaetaan kahteen kategoriaan: primitiivityyppeihin ja viite- eli referenssityyppeihin. Primitiivityyppejä ovat totuusarvotyyppi **boolean** sekä numeeriset tyypit. Viitetyyppejä taas ovat luokat, liittymät ja taulukot. Tyhjille viitteille on oma tyyppi, jolla on yksi arvo, **null**. Viitetyypit voivat olla parametrisoituja, mikä tarkoittaa, että luokalla tai liittymällä voi olla tyyppiparametreja, jotka voivat poiketa toisistaan eri ilmentymissä[1]. Esimerkiksi Vector-säiliöluokka voidaan parametrilla rajoittaa sisältämään vain tietyn tyyppisiä viitteitä, kuten merkkijonoja. Primitiivityypit eivät kuitenkaan voi olla parametreja [2].

Tyypit voivat olla toisten tyyppien alityyppejä. Luokka- tai liittymätyypin C ylityyppejä ovat C:n yliluokat ja C:n toteuttamat liittymät, ja Object-luokka on kaikkien viitetyyppien ylityyppi. Luokkien ja liittymien vastaavuussuhde tyyppeihin ei ole täydellinen, koska luokat ja liittymät ladataan Javan virtuaalikoneeseen ajon aikana dynaamisesti, minkä johdosta tyyppien linkityksessä saattaa ilmetä epäjohdonmukaisuuksia.

4.1.2 Luokkien ja liittymien perintä

Javassa kaikki luokat periytyvät Object-luokasta. Jokainen luokka (paitsi Object) on tasan yhden luokan laajennus (*extension*) ja voi samalla toteuttaa useita liittymiä. Luokka voidaan määritellä abstraktiksi, jolloin siitä ei voi luoda ilmentymiä mutta se voi toimia muiden luokkien yliluokkana. Luokka voidaan myös määri-

[1] Tällaisia luokkia kutsutaan myös geneerisiksi luokiksi, mutta tässä tutkielmassa se saattaisi aiheuttaa sekaannuksia CLOS:n geneerisistä metodeista puhuttaessa.

[2] Primitiivityypeille on järjestelmässä valmiiksi määritellyt kääreluokat, joita voidaan käyttää tyyppiparametreina.

tellä **final**-avainsanalla hierarkian lehtisolmuksi, jolloin sillä ei voi olla aliluokkia. Luokan ominaisuuksien periytymistä voi ohjata näkyvyysmääritteillä, joita on kolme: julkinen (**public**), suojattu (**protected**) ja yksityinen (**private**). Jos attribuutti tai metodi on määritelty julkiseksi tai suojatuksi, luokan kaikki jälkeläiset perivät sen. Yksityiseksi määriteltyä ominaisuutta ei peritä. Jos mitään näkyvyysmääritettä ei anneta, ominaisuus näkyy kaikille samassa pakkauksessa (*package*) oleville luokille. Tällöin samassa pakkauksessa olevat jälkeläisluokat perivät ominaisuuden mutta eri pakkauksessa olevat jälkeläiset eivät. Ominaisuus voidaan määritellä **static**-avainsanalla luokkakohtaiseksi, jolloin se on yhteinen kaikille luokan ilmentymille. Yksittäisen ominaisuuden uudelleenmäärittelyn voi estää käyttämällä **final**-määritettä.

Liittymä (*interface*) on abstrakti luokka, jonka jäseninä voi olla luokkia, toisia liittymiä, vakioita ja abstrakteja metodeja. Jäsenet ovat automaattisesti näkyvyydeltään julkisia. Liittymällä ei ole toteutusta, mutta luokka voi toteuttaa liittymän toteuttamalla sen abstraktit metodit. Liittymä voi periä yhden tai useamman muun liittymän, eli liittymistä voi muodostaa perintäverkon, joka on täysin irrallaan luokkahierarkiasta. Toisistaan muuten riippumattomat luokat voivat toteuttaa saman liittymän, jolloin niitä voidaan käsitellä yhden liittymätyypin muuttujina.

Jos luokassa määritellään tietyn niminen attribuutti, kyseinen määrittely piilottaa kaikki luokan yliluokkien ja liittymien samannimiset attribuutit, joilla on julkinen, suojattu tai pakkaustason näkyvyys. Tämä pätee siinäkin tapauksessa, että päällekkäisillä attribuuteilla on eri tyypit. Luokka voi myös periä monta samannimistä (mahdollisesti erityyppistä) attribuuttia, jos esimerkiksi luokan yliluokassa ja jossakin toteutettavassa liittymässä on samanniminen attribuutti. Tällöin kuitenkin attribuuttiin viittaaminen pelkällä nimellä ilman luokka- tai liittymätarkennetta aiheuttaa käännösvirheen.

Luokka perii välittömältä yliluokaltaan ja liittymiltään kaikki metodit, joita ei ole määritelty yksityisiksi. Jos luokassa määritellään metodi, jonka kutsumuoto (*signature*; nimi ja parametrilista) on identtinen jonkin perityn metodin kutsumuodon kanssa, uusi metodi syrjäyttää perityn. Tällöin metodin paluuarvon tulee olla samaa tyyppiä kuin syrjäytetyn metodin paluuarvon. Jos paluuarvo on viite, se saa uudelleenmäärittelyssä olla myös alkuperäisen tyypin alityyppi; toisin sanoen Java sallii metodien perinnässä kovariantit paluutyypit. Jos on kyse luokkakohtaisista metodeista, määrittelyn mukaan puhutaan syrjäyttämisen sijaan piilottamisesta. Luokkakohtainen metodi ei saa piilottaa ilmentymäkohtaista metodia.

Liittymien ja parametrisoitujen tyyppien takia voi syntyä tilanne, jossa luokka perii monta kutsumuodoltaan identtistä metodia. Tällöin korkeintaan yksi meto-

deista saa olla konkreettinen; muussa tapauksessa seuraa käännösvirhe [3]. Jos jokin metodeista on konkreettinen, se syrjäyttää kaikki abstraktit kumppaninsa. Jos taas kaikki metodit ovat abstrakteja eikä luokka toteuta metodia kyseisellä kutsumuodolla, myös luokka itse on abstrakti. Ristiriitaiset paluuarvotyypit aiheuttavat kummassakin tapauksessa käännösvirheen. Virhetilanne on myös se, jos peritty konkreettinen metodi pystyy aiheuttamaan sellaisen poikkeuksen, jota ei ole määritelty syrjäytettävissä abstrakteissa metodeissa.

4.1.3 Moniperintä

Yksinkertaisimmillaan Javan moniperintää voi käyttää siten, että luodaan liittymistä käsitteellinen perintäverkko ja toteutetaan sitten tarvittavat luokat erikseen. Voidaan määritellä esimerkiksi seuraavanlaiset liittymät:

```
public interface IA {
  void a();
}

public interface IB {
  int b(int value);
}

public interface IC extends IA, IB {
  void c(int value);
}
```

Liittymässä IC on nyt oma metodi **c** ja perinnän ansiosta myös metodit **a** ja **b**. Ongelmana on, että jos tarvitaan toteutukset sekä IA:lle, IB:lle että IC:lle, samaa koodia joudutaan kopioimaan moneen eri paikkaan, koska luokkien moniperintää ei sallita. Näin ollen perinnän mahdollisuudet koodin uudelleenkäytettävyyden parantamiseksi jäävät hyödyntämättä.

Kuten edellä mainittiin, liittymien sisään voidaan määritellä konkreettisia luokkia. Liittymiin voi siis lisätä toiminnallisuutta, mikä antaa mahdollisuuden kiertää Javan moniperinnän rajoituksia. Pasi Manninen on käsitellyt asiaa pro gradu -työssään [20]. Hän on laatinut seuraavan esimerkin, joka perustuu Markku Sakkisen väitöskirjassa [25] olevaan C++-esimerkkiin.

```
public interface Cowboy {
  public void draw();
```

[3] Näin voi käydä, jos yliluokka on parametrisoitu ja sillä on kaksi metodia, jotka olivat erillisiä geneerisessä määrittelyssä mutta samaistuvat perimisen yhteydessä parametrivalinnan johdosta.

```
  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();
  }
}
```

Sisäluokkien avulla on onnistuttu sisällyttämään **draw**-metodin toteutus Cowboy- ja Window-liittymiin, joiden periaatteessa pitäisi olla abstrakteja. Kahden samannimisen metodin nimikonflikti on ratkaistu CowboyWindow-luokassa määrittelemällä kaksi uutta sisäluokkaa, joiden avulla voidaan kutsua haluttua **draw**-metodia. Moniperintä siis onnistuu, mutta ratkaisu on melko monimutkainen ja sekava, kuten Manninenkin toteaa. Lisäksi Javan staattinen tyypitys asettaa rajoituksia perittäville liittymille. Oletetaan, että Cowboy- ja Window-liittymät olisikin määritelty näin:

```
public interface Cowboy {
  public boolean draw();

  // ...
}

public interface Window {
  public int draw();

  // ...
}
```

Tällöin Mannisen esimerkin CowboyWindow-luokka ei olisi mahdollinen, koska Cowboy- ja Window-liittymien **draw**-metodit ovat kutsumuodoltaan identtiset mutta niiden paluuarvot eivät ole samaa tyyppiä. Pieni lievennys paluuarvon tyyppirajoitukseen on, että toinen kahdesta paluuarvotyypistä saa olla peritty toisesta. Tämäkin on kuitenkin huomioitava erikseen siinä luokassa, joka toteuttaa molemmat liittymät. Oletetaan, että luokka B on peritty luokasta A, ja muokataan vielä edellistä esimerkkiä näin:

```
public interface Cowboy {
  public A draw();

  // ...
}

public interface Window {
  public B draw();

  // ...
}
```

Tällöin CowboyWindow-luokan **draw**-metodin paluuarvoksi kelpaa ainoastaan B.

4.2 Smalltalk

4.2.1 Luokat, oliot ja viitteet

Smalltalk on dynaamisesti tyypitetty kieli, jossa kaikki muuttujat ovat standardin [1] mukaan viitteitä. Muuttujaan voidaan sijoittaa viite minkä tahansa luokan ilmentymään. Muuttujat eivät kuitenkaan ole itse olioita. Kaikki metodit palauttavat viitteen johonkin olioon; oletuksena jokainen metodi palauttaa **self**-viitteen, joka

viittaa siihen olioon, jonka kautta metodia kutsuttiin.

Standardissa ei ole järjestelmäluokille tiettyä hierarkiaa, vaan luokat on määritelty sen mukaan, mihin protokolliin ne sopivat. Protokollalla tarkoitetaan tässä kuvausta, jossa luetellaan tiettyyn luokkaan liittyvät käsitteet (*glossary of terms*) sekä metodit. Object-protokolla on yhteinen kaikille luokille; tämä malli on peräisin Smalltalk-80:stä, jossa Object-luokka oli kaikkien luokkien yliluokka.

Standardissa määritellään, että luokka voi periä ominaisuuksia yhdeltä välittömältä yliluokalta. Perittyyn luokkaan voidaan lisätä attribuutteja ja metodeja, mutta siitä ei voi poistaa sellaisia piirteitä, jotka on määritelty ylempänä luokkahierarkiassa. Metodin poisto onnistuu kuitenkin käytännössä siten, että ”poistettavaksi” tarkoitetussa metodissa kutsutaan Object-luokasta perittyä metodia **shouldNotImplement**, joka aiheuttaa ajonaikaisen poikkeuksen.

4.2.2 Moniperintä

Smalltalk-80:n ensimmäisessä julkaisuversiossa ei ollut moniperintää, mutta se lisättiin seuraavaan julkaisuun. Toteutus oli kuitenkin kokeiluluontoinen, eikä sitä hyödynnetty lainkaan järjestelmäluokissa. LaLonde ja Pugh esittelevät kirjassaan [19] Smalltalk-80:n moniperintää seuraavan esimerkkiluokan avulla:

```
Class named: #ExperimentalReadWriteStream
  superclasses: 'ReadStream WriteStream'
  instanceVariableNames: ''
  classVariableNames: ''
  category: 'Experimental'
```

Nimikonfliktitilanteessa järjestelmä tutki, onko kyseessä yksi ja sama metodi. Jos näin ei ollut, tuloksena oli konfliktivirhe. Tällöin määriteltävänä olevaan luokkaan luotiin uusi samanniminen metodi, joka sijoitettiin erityiseen kategoriaan[4] nimeltä **conflicting inherited methods**. Oletuksena tämä metodi palautti virheviestin, mutta se voitiin määritellä uudelleen. Jos uudelleenmäärittelyssä haluttiin kutsua tiettyä perittyä metodia, se onnistui luokkatarkenteen avulla:

```
jokin testi
  ifTrue: [↑self ReadStream.size]
  ifFalse: [↑self WriteStream.size]
```

Moniperintämahdollisuus poistettiin myöhemmin Smalltalk-80:stä. Tällä hetkellä kielen tunnetuimmat toteutukset kuten Squeak, VisualWorks, Dolphin ja GNU

[4]Smalltalkissa voidaan ryhmitellä luokkia ja metodeita käyttötarkoituksen mukaan eri kategorioihin. Kategoriat eivät vaikuta luokan tai metodin toimintaan tai näkyvyyteen, mutta niiden avulla ohjelmoijan on helpompi hallita luokkaselaimessa suuria kokonaisuuksia.

Smalltalk eivät myöskään tue sitä.

Smalltalkin dynaaminen tyypitysjärjestelmä antaa mahdollisuuden jäljitellä moniperintää sellaisissa tapauksissa, joissa yhden luokan halutaan toteuttavan monta protokollaa. Kun luokkaan toteutetaan protokollien vaatimat metodit, sitä voidaan käyttää kaikkien kyseessä olevien protokollien edustajana. Luokan metodeja kutsuvien olioiden ei tarvitse tietää luokan sisäisestä toteutuksesta mitään, kunhan sen rajapinta täyttää halutut vaatimukset. Menetelmän heikkoutena on, että samaa koodia joudutaan kopioimaan kaikkiin luokkiin, joista halutaan tehdä tietyn protokollan edustajia. Tähän liittyvä ongelma on, että jos rajapinta muuttuu, muutos on tehtävä erikseen jokaiseen luokkaan, joka toteuttaa rajapinnan.

4.2.3 Käytöskomponentit

Squeakin versioon 3.9 (julkaistu 2007) on lisätty tuki käytöskomponenteille (*traits*), jotka täydentävät Smalltalkin yksittäisperinnän puutteita. Käytöskomponentti on kokoelma metodeja, jotka eivät ole sidoksissa mihinkään luokkahierarkiaan. Tekijöiden mukaan periaatteena on, että käytöskomponentit ovat ainoastaan uudelleenkäytettävyyttä edistäviä irrallisia osasia; olion luomisessa muottina toimii edelleen luokka [10].

Käytöskomponentti sisältää konkreettisia metodeja, joista muodostuu jokin yleinen käyttäytymismalli. Vastaavasti käytöskomponentti vaatii, että sen käyttäjä toteuttaa tietyt metodit, joita käytöskomponentin metodeissa kutsutaan. Käytöskomponenteissa ei ole lainkaan tilamuuttujia, eivätkä käytöskomponentin metodit koskaan käytä suoraan minkään luokan attribuutteja. Luokkia ja käytöskomponentteja voidaan koostaa muista käytöskomponenteista, mutta koostamisjärjestyksellä ei ole merkitystä. Jos koostamisessa ilmenee nimikonflikti, ohjelmoijan on selvitettävä se eksplisiittisesti. Käytöskomponentit eivät vaikuta luokan semantiikkaan: luokka olisi käsitteellisesti sama, jos kaikki käytöskomponenteista saadut metodit olisi määritelty luokassa itsessään. Sama pätee myös muista käytöskomponenteista koostettuun käytöskomponenttiin.

Ducasse ym. esittävät artikkelissaan [10] esimerkin, jossa luodaan käytöskomponenttien avulla tietovirtojen käsittelyyn tarkoitettu luokkakirjasto. Tietovirta voi olla luettava, kirjoitettava tai molempia sekä mahdollisesti synkronoitu. Tietovirroille yhteiset toiminnot on koottu käytöskomponenttiin nimeltä TPositionableStream, jonka avulla voidaan hallita sijaintia säiliöoliossa:

```
Trait named: #TPositionableStream uses: {}

  atStart
```

```
    ↑self position = self minPosition.

  atEnd
    ↑self position >= self maxPosition.

  setToEnd
    self position: self maxPosition.

  setToStart
    self position: self minPosition.

  maxPosition
    ↑self collection size.

  minPosition
    ↑0.

  nextPosition
    self position: self position + 1.
    ↑self position.

  collection: aCollection
    self requirement.

  collection
    self requirement.

  position: aNumber
    self requirement.

  position
    self requirement.
```

Metodeissa voidaan käyttää aktiiviseen olioon viittaavaa **self**-viitettä, koska sen arvo sidotaan vasta siinä vaiheessa, kun komponentti liitetään johonkin luokkaan. Sama koskee myös yliluokkaan viittaavaa **super**-viitettä. Käytöskomponentin käyttäjältä vaaditut metodit ilmaistaan asettamalla niiden runkoon lause **self requirement**.

TPositionableStream-komponenttia käyttää kaksi muuta komponenttia, TReadStream ja TWriteStream:

```
Trait named: #TReadStream uses: TPositionableStream

  on: aCollection
```

```
    self collection: aCollection.
    self setToStart.

  next
    ↑self atEnd
      ifTrue: [nil]
      ifFalse: [self collection at: self nextPosition].


Trait named: #TWriteStream uses: TPositionableStream

  on: aCollection
    self collection: aCollection.
    self setToEnd.

  nextPut: anElement
    ↑self atEnd
      ifTrue: [self error: 'no space']
      ifFalse: [self collection at: self nextPosition put: anElement].
```

Luokka ReadStream (ja vastaavasti WriteStream) voidaan nyt toteuttaa seuraavasti:

```
Object subclass: #ReadStream
  instanceVariableNames: 'position collection'
  uses: TReadStream

  initialize
    self collection: String new

  position
    ↑position.

  position: aNumber
    position ← aNumber.

  collection
    ↑collection.

  collection: aCollection
    collection ← aCollection.
```

Koostamisessa syntyy konfliktitilanne, jos kahdella identtisesti nimetyllä metodilla on eri rungot. Konfliktia ei synny, jos sama metodi tulee koostekokonaisuuteen useaa reittiä pitkin — tilanne on vastaava kuin entisessä Smalltalk-80:n moniperin-

nässä. Konfliktitilanteet on ratkaistava eksplisiittisesti koostettavassa luokassa tai käytöskomponentissa. Tähän on kaksi tapaa:

- Määritellään koostettavaan luokkaan tai käytöskomponenttiin uusi samanniminen metodi, joka korvaa ristiriitaiset metodit.
- Estetään metodin sisällyttäminen koosteeseen kaikista paitsi yhdestä käytöskomponentista.

Ducassen ja kumppanien esimerkissä on synkronointia varten käytöskomponentti TSynchronize:

```
Trait named: #TSynchronize uses: {}

  acquireLock
    self semaphore wait.

  initialize
    self semaphore: Semaphore new.
    self releaseLock.

  releaseLock
    self semaphore signal.

  semaphore
    self requirement.

  semaphore: aSemaphore
    self requirement.
```

Seuraavaksi määritellään käytöskomponentti TSyncReadStream, jossa hyödynnetään edellä määriteltyjä komponentteja TReadStream ja TSynchronize. Tässä komponentissa on metodi **next**, joka korvaa TReadStreamista saadun metodin. Korvattuun metodiin on kuitenkin mahdollista päästä käsiksi käyttämällä rinnakkaisnimeämistä (*aliasing*) määrittelyn **uses**-osassa. TReadStreamin **next**-metodille annetaan TSyncReadStreamissa toinen nimi, **readNext**.

```
Trait named: #TSyncReadStream
  uses: TSynchronize + (TReadStream @ {#readNext -> #next})

  next
    |read|
    self acquireLock.
    read ← self readNext.
    self releaseLock.
```

```
↑read.
```

Sekä luku- että kirjoitusoperaatiot sisältävä luokka ReadWriteStream voidaan koostaa käytöskomponenteista TReadStream ja TWriteStream. Määrittelyssä syntyy kuitenkin ristiriitatilanne, koska molemmat käytöskomponentit sisältävät omanlaisensa toteutuksen metodille **on**:. Ratkaisuna estetään TReadStreamin **on:**-metodin sisällyttäminen luokkaan. Määrittely on seuraavanlainen:

```
Stream subclass: #ReadWriteStream
  uses: (TReadStream - {#on:}) + TWriteStream
```

TPositionableStream-komponentista saadut ominaisuudet ovat identtisiä TReadStream- ja TWriteStream-komponenteissa, joten ne eivät aiheuta ristiriitoja.

4.2.4 Pohdintaa

Smalltalkin standardin määrittelemä perintä on ilmaisuvoimaltaan melko rajoittunut, mutta dynaaminen tyypitys antaa jonkin verran liikkumavaraa yksittäisperintämallin sisällä. Smalltalkissa ei tarvita lainkaan Javan liittymätyypin kaltaista rakennetta, koska luokka, jossa on tietyt metodit, on dynaamisen tyypityksen ansiosta suoraan tietyn rajapinnan edustaja, eikä sitä tarvitse erikseen kertoa muuttujan esittelyssä. Ongelmaksi jää kuitenkin koodin heikko uudelleenkäytettävyys.

Squeakin käytöskomponentit näyttäisivät parantavan uudelleenkäytettävyyttä merkittävällä tavalla. Työryhmä muotoili käytöskomponenttien avulla uudelleen Squeakin kokoelmaluokkakirjaston ja ilmoittaa, että lopputuloksena kirjastossa on 10 % vähemmän metodeita ja 12 % vähemmän lähdekoodia kuin alkuperäisessä toteutuksessa [10]. Lisäksi käytöskomponentit mahdollistivat luokkien järjestämisen käsitteellisesti loogisempaan hierarkiaan — aiemmin tässä suhteessa oli jouduttu tinkimään, koska koodin uudelleenkäytettävyys haluttiin maksimoida.

Dynaaminen tyypitys helpottaa myös käytöskomponenttien käyttöä. Jos metodien paluuarvoilla olisi staattinen tyyppi, nimikonfliktitilanteet olisivat huomattavasti hankalampia. Parametrien tyyppiristiriidat eivät aiheuta ongelmia, koska niistä selvitään kuormituksella, mutta jos paluuarvojen tyypit ovat poikkeavia eikä kumpikaan ole toisen alityyppi, metodeita ei pysty yhdistämään millään järkevällä tavalla, kuten Javan liittymien moniperinnän yhteydessä todettiin. Ongelmatilanne voitaisiin ratkaista valinnalla tai uudelleennimeämisellä, mutta kumpikin tapa lisää koodin monimutkaisuutta. Käytöskomponentit vaikuttavat kuitenkin käyttökelpoisilta myös staattisesti tyypitetyissä kielissä, ja Ducassen ja kumppanien työryhmä suunnitteleekin käytöskomponenttien toteuttamista mm. Javassa.

4.3 Eiffel

4.3.1 Tyypit ja luokat

Eiffel on Javan tapaan staattisesti tyypitetty kieli. Erona Javaan on, että Eiffelissä kaikki tyypit perustuvat luokkiin. Tyypit voivat olla viitetyyppejä tai laajennettuja (*expanded*) tyyppejä. Näiden kahden kategorian ero on samantapainen kuin Javan viitetyyppien ja primitiivityyppien ero: kun viitetyyppinen olio liitetään muuttujaan tai metodikutsun parametriin, niin kopioidaan ainoastaan viite, mutta laajennetun tyypin tapauksessa kopioidaan koko olio. Luokat ovat oletuksena viitetyyppisiä; ohjelmoija voi määritellä luokan laajennetuksi lisäämällä **class**-avainsanan edelle määritteen **expanded**.

Eiffelin luokkarakenteessa on yksi yhteinen yliluokka: GENERAL. Tästä on peritty luokka PLATFORM ja siitä edelleen ANY. Kaikki muut luokat — järjestelmäluokat ja käyttäjän määrittelemät luokat — ovat automaattisesti ANY-luokan suoria jälkeläisiä. GENERAL sisältää piirteitä, jotka ovat yhteisiä kaikille luokille, kuten esimerkiksi kopiointiin ja syöttö- ja tulostusoperaatioihin liittyviä metodeja. PLATFORM puolestaan määrittelee vakioattribuutteja, jotka ovat riippuvaisia kulloinkin käytössä olevasta laitteistoalustasta. ANY ei määrittele mitään uusia piirteitä vaan ainoastaan perii PLATFORM-luokan. Bertrand Meyerin [22] mukaan ratkaisun etuna on, että ANY voidaan tarvittaessa korvata omalla toteutuksella, jossa yhteisiä piirteitä voidaan määritellä uudestaan. Uuden toteutuksen ei välttämättä tarvitse olla missään yhteydessä GENERAL- ja PLATFORM-luokkiin, mutta tällainen äärimmäisyys on Meyerin mielestä vain harvoin suositeltavaa. On huomattava, että GENERAL ja PLATFORM eivät ole mukana Eiffelin standardissa [12].

ANY-luokan eräänlaisena vastakappaleena on luokka NONE, jonka katsotaan olevan kaikkien muiden luokkien jälkeläinen. Sitä ei ole olemassa tekstinä luokkakirjastossa, vaan kuten Meyer toteaa, se on ”mielikuvituksen tuote”, joka tekee luokkarakenteesta ja tyyppijärjestelmästä yhtenäisen. NONE-luokasta ei voi luoda ilmentymää, eikä siitä voi periä muita luokkia. Siinä ei myöskään ole yhtään ulospäin näkyvää piirrettä. ANY-luokassa on (perittynä GENERAL-luokasta) NONE-tyyppinen attribuutti **Void**, joka toimii kaikkien olioviitteiden alustusarvona. Viite, jonka arvo on sama kuin **Void**, tarkoittaa tyhjää viitettä.

4.3.2 Perintämekanismit

Eiffelissä piirteiden periytymistä ei voi rajoittaa, vaan luokka perii aina kaikki yliluokkiensa piirteet. Perittyjä piirteitä voidaan kuitenkin muokata perijäluokassa mo-

nella eri tavalla:

- Piirre, joka on määritelty tietyssä luokassa tietyllä nimellä, voidaan nimetä uudelleen jälkeläisluokissa.
- Perityn metodin toteutus voidaan korvata jälkeläisluokassa uudella toteutuksella eli metodi voidaan määritellä uudelleen.
- Piirre, jolla on tietty kutsumuoto, voi saada uuden kutsumuodon. Tämä saavutetaan uudelleenmäärittelyllä sekä ns. ankkuroidulla esittelyllä (*anchored declaration*), jossa muuttujan tyyppi sidotaan jonkin toisen muuttujan tyyppiin.
- Yliluokassa määritelty abstrakti metodi voidaan toteuttaa jälkeläisluokassa. Toteutettu metodi voidaan myös asettaa jälkeläisluokassa abstraktiksi.
- Perijäluokassa voidaan muuttaa perittyjen piirteiden näkyvyyttä ulkopuolelle.

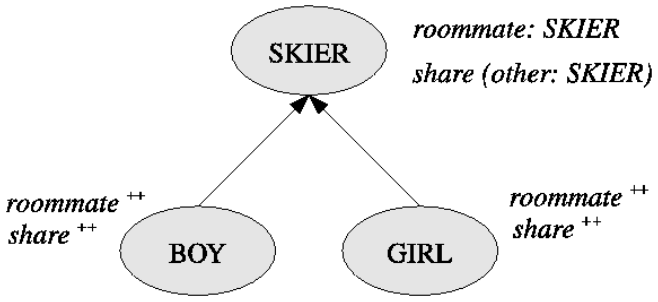
Lisäksi moniperintää ja toistuvaa perintää varten on määritteitä, joita käsitellään myöhemmin tässä luvussa.

Metodin uudelleenmäärittelyssä sallittu kutsumuodon muuttaminen on rajoitettu kovarianssiin: metodin parametrien ja paluuarvon on oltava yhteensopivia alkuperäisten vastineidensa kanssa, eli korvaavat tyypit voivat olla vain alkuperäisten luokkien aliluokkia. Parametrien lukumäärää ei voi muuttaa.

Kovariantissa uudelleenmäärittelyssä on polymorfismiin liittyvä heikkous, jonka Meyer esittelee kirjassaan [23]. Jos uudelleenmääriteltyä metodia kutsutaan polymorfisen muuttujan kautta, sille voidaan antaa parametri, joka rikkoo metodin tyyppirajoituksen. Meyer havainnollistaa ongelmaa esimerkkihierarkialla, joka kuvaa koulun hiihtojoukkuetta (kuva 4.1).

Luokka SKIER on määritelty seuraavasti:

```
class SKIER feature
  roommate: SKIER
            -- Hiihtäjän huonetoveri
  share (other: SKIER)
            -- Valitsee other-olion huonetoveriksi.
    require
      other_exists: other /= Void
    do
      roommate := other
    end
  ...
end -- class SKIER
```



Kuva 4.1: Kovariantti uudelleenmäärittely

Tavoitteena on, että tytöt ja pojat eivät voi olla toistensa huonetovereita, joten GIRL (ja vastaavasti BOY) määritellään kovarianssin avulla seuraavasti:

```
class GIRL inherit
  SKIER
    redefine roommate, share end
  feature
    roommate: GIRL
    share (other: GIRL)
      require
        other_exists: other /= Void
      do
        roommate := other
      end
end -- class GIRL
```

Nyt seuraavalla koodilla saadaan aikaan laiton tilanne:

```
s: SKIER; b: BOY; g: GIRL
...
create b; create g;     -- Luodaan BOY- ja GIRL-tyyppiset oliot.
s := b;          -- Polymorfinen sijoitus.
s.share(g);
```

Viimeinen sijoitus näyttää päällepäin tyyppiturvalliselta, koska **share** on SKIER-luokan julkinen piirre ja GIRL on yhteensopiva **share**-metodin **other**-parametrin tyypin kanssa.

4.3.3 Moniperintä

Eiffelissä on täysi tuki moniperinnälle. Nimikonflikteihin ja toistuvaan perintään on varauduttu useilla eri ominaisuuksilla, mutta ohjelmoijan on ratkaistava ristiriitatilanteet pääasiassa itse — järjestelmä ei tee sitä automaattisesti.

Jos kaksi erillisiin hierarkioihin kuuluvaa yliluokkaa aiheuttaa perityssä luokassa nimikonfliktin, se voidaan korjata nimeämällä toinen tai molemmat konfliktin aiheuttavista piirteistä uudelleen. Jos luokalla C on vanhemmat A ja B, joissa molemmissa on piirre (attribuutti tai metodi) nimeltä **fname**, C voidaan määritellä seuraavasti:

```
class C inherit
  A
    rename fname as name1 end;
  B
    rename fname as name2 end;
  ...
```

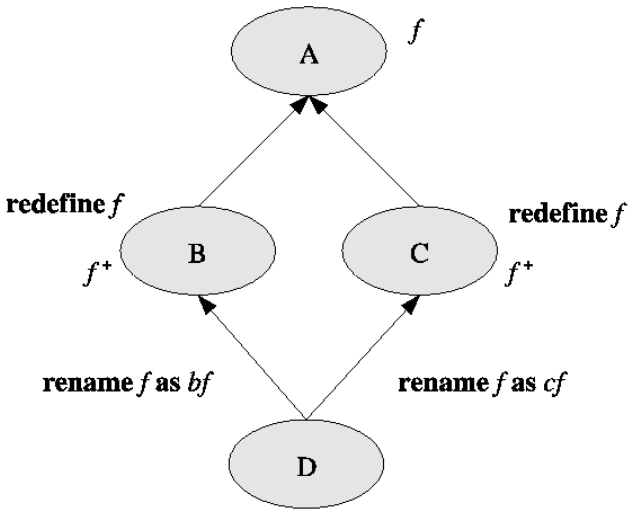
Kun C-olion **fname**-metodia kutsutaan A- tai B-tyyppisen muuttujan kautta, järjestelmä osaa valita kahdesta uudelleennimetystä metodista oikean. Uudelleennimeäminen ratkaisee siis yksinkertaisen nimikonfliktin, mutta toistuvassa perinnässä tilanne muuttuu hankalammaksi. Ensimmäinen ongelma on, halutaanko moneen kertaan peritty piirre moneksi erilliseksi piirteeksi vai vain yhdeksi piirteeksi. Eiffelissä on mahdollista valita kumpi tahansa lähestymistapa. Kielen kehittäjä Bertrand Meyer kutsuu ensin mainittua monistamiseksi ja jälkimmäistä jakamiseksi.

Meyer esittää kirjassaan [22] kolme tapausta, jotka aiheuttavat ongelmia toistuvassa perinnässä:

- Erillisillä poluilla tapahtuvat metodien uudelleenmäärittelyt.
- Attribuuttien kahdennukset.
- Ristiriitaiset parametrisoitujen luokkien perimiset.

Ensimmäisessä tapauksessa ongelmia esiintyy silloin, kun moneen kertaan perittyä metodia kutsutaan sellaisen muuttujan kautta, joka edustaa perintäverkon yhteistä yliluokkaa. Kuvassa 4.2 on esimerkki tällaisesta tilanteesta.

Kun luokan D olio sijoitetaan A-tyyppiseen muuttujaan, **f**-metodin kutsu on epäselvä, koska se voi tarkoittaa joko luokassa B tai luokassa C uudelleen määriteltyä metodia. Tästä syystä ohjelmoijan on Eiffelissä ilmoitettava **select**-määritteellä, mikä samannimisistä metodeista valitaan polymorfisissa kutsuissa. Esimerkin luokka D voitaisiin määritellä näin:



Kuva 4.2: Ristiriitaiset uudelleenmäärittelyt

```
class D inherit
  B
    rename
      f as bf
    select
      bf
    end;
  C
    rename
      f as cf
    end
end -- class D
```

Edellisessä esimerkissä luokasta A peritty metodi **f** monistettiin. Jos sen halutaan pysyvän luokassa D vain yhtenä metodina, Eiffelissä ainoa vaihtoehto on muuttaa B:n ja C:n kautta perityistä **f**-metodeista toinen abstraktiksi eli poistaa D:stä sen toteutus. Tässä tapauksessa D:n määrittely voisi olla seuraavanlainen:

```
class D inherit
  B;
  C
    undefine f end
end -- class D
```

Attribuuttien toistuvassa perinnässä esiintyy samankaltainen ongelma kuin metodien tapauksessa. Perityissä metodeissa voidaan käyttää muuttujaa, joka viittaa

kahteen tai useampaan eri attribuuttiin, eikä oikeaa attribuuttia pystytä päättelemään. Ratkaisuna Eiffelissä on sama keino kuin metodien valinnassa: monistetuista attribuuteista yksi on ristiriitojen välttämiseksi valittava hallitsevaksi **select**-määritteellä. Seuraavassa esimerkissä D-luokka perii kahteen kertaan A-luokan, jolla on **attr**-attribuutti [5].

```
class D inherit
  A
    rename
      attr as attr1
    select
      attr1
    end;
  A
    rename attr as attr2 end
end -- class D
```

Kolmas Meyerin mainitsema ongelmatapaus koskee perintäverkkoa, jossa yhteinen yliluokka on parametrisoitu. Meyer esittää seuraavanlaisen esimerkkitilanteen:

```
class A [G] feature
  f(x: G) ... end
end -- class A

class B inherit
  A [INTEGER]
end -- class B

class C inherit
  A [REAL]
end -- class C
```

Luokassa A on geneerinen parametri G, jonka tilalla on luokassa B INTEGER ja luokassa C REAL. Jos nyt luokista B ja C halutaan periä uusi luokka D, A:ssa määritelty metodi **f** aiheuttaa ristiriitatilanteen, sillä vaikka uudelleenmäärittelyjä ei ole, B:n ja C:n A:lle antamat geneeriset parametrit tekevät niiden **f**-metodeista keskenään yhteensopimattomia.

Meyer esittää ongelmaan samanlaista ratkaisua kuin aiemmissakin tapauksissa. Metodin jakaminen on mahdotonta, joten se on monistettava tai toinen B:n ja C:n **f**-metodeista on tehtävä D:ssä abstraktiksi. Jälkimmäinen ratkaisu on ongelmallinen, koska perityn metodin täytyy olla tyypiltään yhteensopiva kaikkien yliluokkien vastaavien metodien kanssa. Eiffelissä ainoa mahdollisuus on tehdä sekä B:n

[5] Meyer nimittää tällaista tapausta suoraksi toistuvaksi perinnäksi.

että C:n **f**-metodeista abstrakteja ja määritellä D:n **f** seuraavasti:

```
f(x: NONE) do ... end
```

NONE on Eiffelin järjestelmäluokka, joka on kaikkien luokkien jälkeläinen mutta jolla ei ole yhtään ulospäin näkyvää piirrettä. Lopputulos on, että esimerkin luokalla D ei ole mitään hyödyllistä käyttöä. Meyer on myöhemmin esittänyt, että moniperintä rajoitettaisiin tällaisissa tapauksissa monistamiseen [23], ja uusimmassa Eiffelin standardissa [12] tällainen moniperinnän muoto on kielletty kokonaan.

4.3.4 Pohdintaa

Eiffel tarjoaa hyvin monipuoliset mahdollisuudet moniperintään. Staattinen tyypitys ei aiheuta merkittäviä hankaluuksia moniperinnän kannalta. Dynaamiseen tyypitykseen verrattuna ohjelmoija joutuu kirjoittamaan enemmän koodia varsinkin konfliktitilanteissa, mutta toisaalta voidaan pitää etuna, että perintäsuhteista tulee melko täsmällisesti määriteltyjä.

Monipuolisuus tarkoittaa usein myös monimutkaisuutta. Pasi Manninen [20] moittii Eiffelissä muun muassa luokkien perittyjen piirteiden vapaata uudelleennimeämismahdollisuutta ja ehdottaa, että uudelleen nimeäminen voisi olla sallittua esimerkiksi vain sellaisille piirteille, jotka aiheuttavat perinnän yhteydessä nimikonfliktin. Manninen pitää ongelmallisena myös konflikteissa tarvittavan valintalausekkeen käyttöä ja sitä, miten se toimii yhdessä muiden mekanismien kanssa. Hänen mielestään tilannetta voisi parantaa rajoittamalla tietyissä tilanteissa olevien toimintavaihtoehtojen määrää ja lisäämällä automaattista konfliktien käsittelyä.

Markku Sakkinen [25] puolestaan kritisoi Eiffelin toimintaa alioliokeskeisen perintämallin näkökulmasta. Eiffelissä on mahdollista yhdistellä konfliktitilanteissa jakavaa ja monistavaa piirteiden perintää, mikä Sakkisen mukaan johtaa aliolioiden viipaloitumiseen ja alioliorakenteen yhtenäisyyden rikkoutumiseen. Sakkinen mainitseekin toisaalla [26], että Eiffel noudattaa perinnässä enemmän attribuuttikeskeistä periaatetta. Käsitteellisen selkeyden kannalta olisi kenties parempi, jos Eiffel säilyttäisi aliolioiden eheyden, mutta tämän vaatimuksen myötä saatettaisiin joutua tinkimään moniperintämekanismien joustavuudesta.

Sakkinen pitää myös puutteena sitä, että perinnän näkyvyyttä ei voi Eiffelissä rajoittaa samaan tapaan kuin C++:ssa. Toteutustason moniperinnän ja käsitteellisen moniperinnän sekoittuminen antaa mahdollisuuden väärinkäytöksiin ja aiheuttaa myös muita hankaluuksia, kuten alaluvussa 3.2.5 todettiin.

4.4 CLOS

4.4.1 Tyypit, luokat ja oliot

CLOS on dynaamisesti tyypitetty järjestelmä. Standardi [8] antaa tyypille tarkan määritelmän: Tyyppi on olioista koostuva (mahdollisesti ääretön) joukko. Olio voi liittyä useampaan kuin yhteen tyyppiin. Tyypit eivät ole olioita. Tyyppeihin viitataan epäsuorasti tyyppimääritteillä (*type specifiers*); nämä ovat olioita, jotka kuvaavat tyyppejä. Olio on yhteinen nimitys Common Lispin kaikille datakokonaisuuksille [6].

Olioilla on tyyppi, mutta muuttujilla ei. Yleensä millä tahansa muuttujalla voi olla arvonaan mikä tahansa olio. Jos muuttujaan halutaan vain tietyntyyppisiä arvoja, voidaan käyttää erityistä tyyppirajoitetta (*type declaration*). Tyypit on järjestetty suunnattuun asykliseen verkkoon.

Luokka on CLOS:ssa Smalltalkin tapaan olio, joka määrittää rakenteen ja käyttäytymisen toisille olioille, jotka ovat tämän olion ilmentymiä eli instansseja. Luokat on järjestetty suunnattuun asykliseen verkkoon, jossa huipulla on kaksi erikoisluokkaa, t ja standard-object. Luokka t on järjestelmän kaikkien muiden luokkien yliluokka, eikä sillä ole yliluokkia. Luokka standard-object puolestaan on luokan standard-class ilmentymä, ja se on yliluokka kaikille muille standard-classin ilmentymille. Kun **defclass**-makrolla määritellään uusi luokka, standard-object tulee automaattisesti luokan yliluokaksi.

CLOS yhdistää luokka-avaruuden tyyppiavaruuteen siten, että jokaisella luokalla, jolla on kunnollinen nimi, on automaattisesti vastaava tyyppi, jolla on sama nimi. Tällöin luokan nimi toimii samalla myös tyypin määritteenä. Luokan kunnollinen nimi (*proper name*) on standardin mukaan symboli, joka nimeää sen luokan, jonka nimenä on tuo symboli.

Myös jokainen luokkaolio on validi tyypinmäärite. Kaikille Common Lispin tyypeille ei kuitenkaan ole vastaavaa oliota. CLOS:ssa voi myös olla tyypinmääritteitä, jotka ovat listoja, kuten esimerkiksi **(vector double-float 100)**; näillekään ei ole vastaavia luokkia. Sonya E. Keenen [17] mukaan syynä tähän on se, että tällaisissa tapauksissa olisi hankala päätellä luokkien perintäjärjestystä, josta kerrotaan edempänä.

Luokka, joka vastaa jotakin aiemmin määriteltyä Common Lispin tyypinmäärittettä, voidaan toteuttaa kolmella eri tavalla: standardiluokkana, rakenteisena luok-

[6] Englanninkielinen sana *object* on Common Lispissä merkitykseltään laajempi kuin muissa tämän tutkielman kielissä. ”Olio” ei siis ehkä ole aivan täsmällinen suomennos, mutta se on valittu tähän yhtenäisyyden vuoksi, koska kyse on myös luokkien ilmentymistä.

kana (*structure class*) tai sisäänrakennettuna luokkana (*built-in class*). Rakenteisella luokalla tarkoitetaan tietorakennetta, joka on määritelty **defstruct**-makrolla. Rakenteiset luokat ovat hieman rajoittuneempia kuin **defclass**-makrolla määritellyt standardiluokat; niissä ei esimerkiksi ole moniperinnän mahdollisuutta. Sisäänrakennetut luokat puolestaan poikkeavat standardiluokista ominaisuuksiensa ja sisäisen toteutuksensa osalta, jotta ne pystyisivät hyödyntämään laitteiston arkkitehtuuria mahdollisimman tehokkaasti. Niistä ei voi luoda ilmentymiä **make-instance**-makrolla, eikä niitä voi käyttää muiden luokkien yliluokkina (poikkeuksena t, joka on kaikkien luokkien yliluokka). Niiden tilaa ei voi lukea **slot-value**-funktiolla, eikä niitä voi määritellä uudelleen. Keenen [17, s. 220] mukaan Common Lispin tyyppejä vastaavat luokat on toteutettu useimmiten sisäänrakennettuina luokkina.

4.4.2 Perintä

CLOS tukee moniperintää kuten Eiffelkin. Moniperinnän toteutuksessa on kuitenkin merkittäviä eroja. Eiffelissä yhdistellään verkkototeutusta ja puutoteutusta, kun taas CLOS:n ratkaisu on lineaarinen toteutus (ks. alaluku 3.2.5). Perintäverkko muunnetaan siis lineaariseksi poluksi, jolla ei ole kaksoisesiintymiä. Luokkien perintäjärjestys eli tärkeyslista (*class precedence list*) päätellään kahden säännön avulla [17]:

Luokkien tärkeyssääntö 1: Luokka on aina etusijalla yliluokkiinsa nähden.

Luokkien tärkeyssääntö 2: Jokainen luokka määrittelee välittömien yliluokkiensa tärkeysjärjestyksen.

Säännössä 2 välittömien yliluokkien tärkeysjärjestys määräytyy sen mukaan, missä järjestyksessä yliluokat listataan **defclass**-makrossa. Luokka on tärkeämpi (eli erikoistuneempi) kuin ne luokat, jotka tulevat sen jälkeen listassa.

Kun CLOS päättelee luokan perintäjärjestystä, se aloittaa luokan määrittelystä. Se soveltaa molempia tärkeyssääntöjä määrittelyyn ja saa siten joukon paikallisia järjestysrajoitteita. Sen jälkeen sääntöjä sovelletaan jokaisen välittömän yliluokan määrittelyyn ja vastaavasti niiden yliluokkien määrittelyihin, kunnes kaikki polut päätyvät järjestelmän juuriluokkaan t. Tuloksena saadaan järjestysrajoitteiden joukko kaikille käsittelyssä oleville luokille.

Seuraava vaihe perintäjärjestyksen muodostamisessa on löytää täydellinen järjestys, joka noudattaa kaikkia edellä saatuja järjestysrajoitteita. CLOS hoitaa asian tekemällä järjestysrajoitteiden joukolle topologisen lajittelun. Operaatio voi päättyä kolmella tavalla:

- Tasan yksi täydellinen järjestys täyttää ehdot.
- Monta täydellistä järjestystä täyttää ehdot.
- Mikään täydellinen järjestys ei täytä ehtoja.

Kahdessa ensimmäisessä tapauksessa CLOS muodostaa luokkien tärkeyslistan. Kolmannessa tapauksessa se antaa tulokseksi virheen.

Keene havainnollistaa näitä tapauksia esimerkeillä. Ensimmäistä tapausta kuvaa seuraava määrittely:

```
(defclass stream () ())
(defclass input-stream (stream) ())
(defclass char-stream (stream) ())

(defclass char-input-stream
  (char-stream input-stream)
  ())
```

Vain yksi järjestys on mahdollinen tälle määrittelylle. Se on seuraava:

```
(char-input-stream char-stream input-stream stream standard-object t)
```

Toinen tapaus saadaan aikaan seuraavilla luokkamäärityksillä:

```
(defclass stream () ())

(defclass buffered-stream (stream) ())
(defclass disk-stream (buffered-stream) ())

(defclass char-stream (stream) ())
(defclass ascii-stream (char-stream) ())

(defclass ascii-disk-stream
  (ascii-stream
   disk-stream)
  ())
```

Esimerkiksi seuraavat tärkeyslistat toteuttavat edellisten määritysten aiheuttamat rajoitteet:

```
(ascii-disk-stream ascii-stream char-stream
 disk-stream buffered-stream stream standard-object t)

(ascii-disk-stream ascii-stream disk-stream
 buffered-stream char-stream stream standard-object t)

(ascii-disk-stream ascii-stream disk-stream
 char-stream buffered-stream stream standard-object t)
```

CLOS valitsee näistä listoista ensimmäisen, koska se pyrkii pitämään luokkaperheet tärkeyslistoissa yhtenäisinä. Luokan ascii-stream sukupuu tulee tärkeyslistassa kokonaisuudessaan ennen luokan disk-stream puuta, lukuun ottamatta luokkaa stream, joka on sekä ascii-streamin että disk-streamin yliluokka. Ohjelmoija voi halutessaan muuttaa perintäjärjestystä luettelemalla luokkamäärittelyssä välittömien yliluokkien lisäksi myös kauempana ketjussa olevia yliluokkia, esimerkiksi näin:

```
(defclass ascii-disk-stream
  (ascii-stream disk-stream
   char-stream buffered-stream)
  ())
```

Tällä määrittelyllä valituksi tulee edellä esitetyistä luokkajärjestyksistä kolmas.

Tapaus, jossa mikään täydellinen järjestys ei noudata luokan asettamia rajoitteita, seuraa esimerkiksi tällaisesta määrittelystä:

```
(defclass stream () ())
(defclass input-stream (stream) ())
(defclass buffered-stream (stream) ())

(defclass disk-stream (buffered-stream input-stream) ())

(defclass tape-stream (input-stream buffered-stream) ())

(defclass disk-emulating-tape-stream (disk-stream tape-stream) ())
```

Viimeinen luokkamääritys aiheuttaa virheen, koska luokan toisessa välittömässä yliluokassa listataan buffered-stream ennen input-streamia ja toisessa taas input-stream ennen buffered-streamia. Tätä ristiriitaa CLOS ei pysty ratkaisemaan.

CLOS:n perintämekanismi toimii siten, että jos luokka perii monelta yliluokalta samannimisen attribuutin, luokkaan tulee vain yksi tätä nimeä vastaava attribuutti. Tuon attribuutin määritteet ovat yhdistelmä yliluokissa olevien samannimisten attribuuttien määritteistä. Ohjelmoija voi myös lisätä attribuutille uusia määritteitä.

Tyypityksen kannalta merkittävä asia on se, miten **:type**-määrite peritään monesta samannimisestä attribuutista. Kyseisellä määritteellä voidaan asettaa attribuutti tietyn tyyppiseksi. CLOS:n määrittelydokumentti [3] kertoo, että luokan tyyppirajoite muodostetaan yhdistämällä kaikkien perintälistassa olevien luokkien **:type**-määritteet. Jos yksikään luokka ei määrittele tyyppirajoitetta, attribuutin tyypiksi tulee automaattisesti t. Keene toteaa, että tämän johdosta luokassa ei voi keventää perittyjä tyyppirajoitteita, mutta niitä voidaan tiukentaa. On mahdollista, että perinnän kautta tietylle attribuutille muodostuu tyyppirajoite, jossa on keskenään yhteensopimattomia tyyppejä. Tällöin kaikki sijoitukset tuohon attribuuttiin tulevat

laittomiksi.

CLOS:n määrittelyssä kerrotaan, että jos attribuuttiin yritetään sijoittaa arvo, joka ei noudata tyyppirajoitteita, operaation tulos on määrittelemätön. Tyyppirajoitteen määrittely ei siis takaa sitä, että attribuutin arvo todella noudattaa rajoitetta. Joissakin toteutuksissa, kuten esimerkiksi CMUCL:ssä, voidaan kuitenkin määrätä järjestelmä tarkistamaan kaikki sijoitukset ja antamaan virheilmoitus, jos sijoitettavan arvon tyyppi ei ole sallittu.

Oikean metodin valinta monesta samannimisestä tehdään CLOS:ssa ensisijaisesti metodien parametrilistan pohjalta. Jos luokka on peritty monesta yliluokasta, joille kullekin on määritelty samanniminen metodi samanlaisella parametrilistalla, järjestelmä valitsee geneeristä funktiota kutsuttaessa sen metodin, joka on määritelty luokan tärkeyslistalla korkeimmalla olevalle luokalle. Jos luokalle on geneerisessä funktiossa oma metodi, se tulee valituksi, koska luokka on aina oman tärkeyslistansa ensimmäisenä. Parametrilistan lisäksi muutkin tekijät voivat vaikuttaa metodin valintaan, ja parametrilistasta voidaan tutkia myös muita piirteitä kuin parametrien tyyppejä.

Jos metodin suorituksen yhteydessä halutaan kutsua muita samannimisiä metodeja, voidaan käyttää CLOS:n järjestelmäfunktiota **call-next-method**, joka suorittaa samannimisistä metodeista sen, joka on luokan tärkeyslistalla seuraavana. Tähän ratkaisuun liittyy kuitenkin ongelmia, kuten Keene esittää. Jos halutaan, että kaikkia perintäverkon samannimisiä metodeja kutsutaan, on huolehdittava siitä, että kaikissa paitsi viimeisessä kutsutaan **call-next-method**-funktiota. Lisäksi on varauduttava siihen, että seuraavaa metodia ei löydykään, eli sen olemassaolo on testattava erikseen **next-method-p**-funktiolla.

Toinen mahdollisuus ohjata metodien kutsumista on käyttää CLOS:n lisämetodeja (*auxiliary methods*). Jos metodien toiminnallisuus sijoitetaan niin sanottuihin esimetodeihin (*before-methods*)[7], jotka suoritetaan aina ennen varsinaisen metodin suoritusta tärkeyslistan mukaisessa järjestyksessä, **call-next-method** ei enää ole tarpeellinen. Keenen mukaan haittapuolena tässä ratkaisussa on, että metodien roolit eivät täysin vastaa niiden käyttötarkoitusta. Tavallisesti päämetodi on se, joka tekee varsinaisen työn, kun taas lisämetodit huolehtivat nimensä mukaisesti vain toiminnallisuuteen liittyvistä lisätehtävistä.

[7]Muut lisämetodityypit ovat jälkimetodi (*after-method*) ja kääremetodi (*around-method*).

4.4.3 Pohdintaa

CLOS:n tapa mallintaa olioita on hyvin erilainen kuin useimmissa muissa kielissä, ja tämä heijastuu myös perintään. Kokonaisuus on kuitenkin harkittu ja toimiva, eikä voida sanoa, että esimerkiksi Eiffel olisi tässä suhteessa selkeästi parempi. Dynaaminen tyypitys sopii CLOS:n malliin hyvin, mutta järjestelmä voisi periaatteessa olla myös staattisesti tyypitetty. Geneeristen funktioiden toiminta pysyy samanlaisena, olipa tyypitystapa kumpi tahansa. Dynaamisen tyypityksen etuna on kuitenkin se, että samannimiset perityt attribuutit eivät aiheuta konflikteja, koska ne voidaan helposti yhdistää jälkeläisluokassa yhdeksi attribuutiksi[8].

CLOS:n valinnainen staattinen tyypitys on mielenkiintoinen ominaisuus. Se antaa mahdollisuuden parantaa ohjelmien tehokkuutta, turvallisuutta ja luettavuutta ilman, että olisi luovuttava kokonaan dynaamisen tyypityksen joustavuudesta. Ongelmana on kuitenkin, että tyyppitarkistukset ohitetaan kokonaan, mikäli järjestelmän turvallisuustaso on asetettu riittävän alhaiseksi. Toinen ongelma on, että perittyjen attribuuttien **:type**-määritteiden mahdollisia ristiriitoja ei pystytä käsittelemään kunnolla. Näitä ongelmia voidaan jossakin määrin lieventää toteutuksessa, mutta niiden täydellinen ratkaiseminen vaatisi luultavasti muutoksia standardiin.

4.5 JavaScript

4.5.1 Tyypitys

JavaScript on luokaton, prototyyppipohjainen ja dynaamisesti tyypitetty oliokieli. Muuttujan arvo voi olla yksi yhdeksästä perustyypistä, joita ovat Undefined, Null, Boolean, String, Number, Object, Reference, List ja Completion. Näistä kolmen viimeisen tyypin arvoja käytetään ainoastaan välituloksina lausekkeiden arvonmäärityksessä, eikä niitä voi asettaa olioiden piirteiksi. Undefined, Null, Boolean, String ja Number ovat primitiivityyppejä. Undefined on alustamattomien muuttujien tyyppi, ja Null kuvaa tyhjää viitettä.

Olio on JavaScriptissä Object-tyypin edustaja. Määritelmän [11] mukaan se on järjestämätön kokoelma piirteitä, joista jokainen sisältää primitiiviarvon, olion tai funktion. Primitiivityypeistä Boolean, Number ja String voidaan luoda vastaava olio kutsumalla **new**-lausekkeessa kyseisen tyypin konstruktorifunktiota ja antamalla parametriksi tyypin arvo. Tuloksena syntyvällä oliolla on implisiittinen, nimetön piirre, joka sisältää annetun primitiiviarvon. Primitiivityyppiä edustava olio

[8]CLOS:n perintämalli on attribuuttikeskeinen, ei niinkään alioliokeskeinen (ks. alaluku 3.2.2).

voidaan pakottaa takaisin primitiivityypin edustajaksi.

4.5.2 Prototyypit ja perintä

JavaScript tukee prototyyppipohjaista perintää (*prototype-based inheritance*). Tällaisessa perinnässä on kaksi yleisesti käytettyä mallia: kopiointi ja delegointi. Kopiointimallissa prototyypistä kopioidaan halutut piirteet uuteen olioon, kun taas delegointimallissa olio ohjaa prototyypilleen sellaiset kutsut, joita se ei itse osaa käsitellä. Taivalsaari [30] vertailee näitä kahta mallia ja toteaa, että delegoinnissa on kopiointiin nähden monia etuja, joista merkittävimpiä ovat tehokkaampi muistinkäyttö ja muutettujen ominaisuuksien dynaaminen päivittyminen kaikkiin delegoiviin olioihin.

JavaScriptissä käytetään delegointia. Jokaiseen konstruktoriin on liitetty yksi prototyypiksi kutsuttu olio, johon jokainen konstruktorilla luotu olio sisältää näkymättömän viitteen. Prototyyppi voi myös sisältää viitteen omaan prototyyppiinsä ja tämä edelleen omaansa. Olio perii prototyypiltään ja tämän kaikilta prototyypeiltä kaikki ominaisuudet eli datamuuttujat (attribuutit) ja metodit. Mekanismi on siis samankaltainen kuin Eiffelissä, Javassa ja Smalltalkissa, mutta merkittävä ero on luokkien puuttuminen. Prototyyppiin voidaan viitata lausekkeella *olio*.**constructor.prototype**. Kun tälle oliolle lisätään uusia piirteitä, nuo piirteet tulevat suoraan myös kaikille olioille, jotka liittyvät kyseiseen prototyyppiin.

JavaScriptissä oliolla voi määritelmän mukaan olla vain yksi prototyyppi, joten periaatteessa moniperintää ei tueta. Sitä voidaan kuitenkin jäljitellä, sillä konstruktorifunktiossa voi kutsua montaa eri konstruktoria. Richard Kitchen esittelee JavaScriptin rajoitettua moniperintää artikkelissaan [18]. Hän tekee esimerkissään seuraavat määrittelyt:

```
function Employee (name, dept) {
  this.name = name || "";
  this.dept = dept || "general";
  // Jos parametrilla ei ole arvoa, se tuottaa totuusarvolausekkeessa
  // epätoden tuloksen. Niinpä edellä olevissa sijoituksissa voidaan
  // hyödyntää loogista tai-operaattoria "||" sen tarkistamiseen,
  // onko parametri määritelty. Jos näin ei ole, muuttujiin sijoitetaan
  // oletusarvot.
}

function WorkerBee(name, dept, projs) {
  this.base = Employee;
  this.base(name, dept);
```

```
  this.projects = projs || [];
}
WorkerBee.prototype = new Employee(); // Tässä prototyypiksi voitaisiin
                                      // sijoittaa myös jokin aiemmin
                                      // luotu olio.

function Hobbyist(hobby) {
  this.hobby = hobby || "scuba";
}

function Engineer(name, projs, mach, hobby) {
  this.base1 = WorkerBee;
  this.base1(name, "engineering", projs);
  this.base2 = Hobbyist;
  this.base2(hobby);
  this.machine = mach || ""; // Alkuperäisessä esimerkissä parametri mach
                             // sijoitettiin muuttujaan this.projects,
                             // mutta se on selkeästi virhe.
}
Engineer.prototype = new WorkerBee();

dennis = new Engineer("Doe, Dennis", ["collabra"], "hugo")
```

Lopussa alustetulla dennis-muuttujalla on nyt seuraavat attribuutit:

```
dennis.name == "Doe, Dennis"
dennis.dept == "engineering"
dennis.projects == ["collabra"]
dennis.machine == "hugo"
dennis.hobby == "scuba"
```

Engineer-konstruktori perii siis ominaisuudet kahdelta eri konstruktorilta, Hobbyistilta ja WorkerBeelta, sekä lisäksi WorkerBeen prototyypiltä Employeelta. Rajoituksena on, että jos Hobbyist-konstruktoriin lisätään dynaamisesti ominaisuuksia jälkeen päin, muutokset eivät välity dennis-olioon. Kitchen jatkaa esimerkkiä seuraavalla lauseella:

```
Hobbyist.prototype.equipment = ["mask", "fins", "regulator", "bcd"]
```

Nyt kaikki tämän jälkeen Hobbyist-konstruktorilla luodut oliot saavat attribuutikseen equipment-taulukon, mutta aiemmin luotu dennis-olio ei. Jos Engineerin prototyypiksi määriteltyyn WorkerBeehen tai sen prototyyppiin Employeehen tehtäisiin vastaava lisäys, myös dennis saisi sen.

JavaScriptin rajoitetussa moniperinnässä nimikonfliktit eivät aiheuta virheitä, koska konstruktorin tai olion piirteitä määriteltäessä viimeiseksi määritelty piirre

korvaa kaikki muut samannimiset piirteet. Tehdään seuraavat yksinkertaiset määrittelyt:

```
function A() {
  this.name = "A";
  this.GetName = function() {
    return this.name;
  };
}

function B() {
  this.name = "B";
  this.GetName() = function() {
    return this.name;
  };
}

function C() {
  this.base1 = A;
  this.base1();
  this.base2 = B;
  this.base2();
}
C.prototype = new A();

cObject = new C();
```

Oliolla cObject on nyt attribuutti **name** ja metodi **GetName**. Molemmat palauttavat kutsuttaessa merkkijonon ”B”, koska C-konstruktorissa on kutsuttu viimeisenä B-konstruktoria, joka sijoittaa mainittuihin piirteisiin omat arvonsa. Se, että C:n prototyypiksi on asetettu A, ei vaikuta asiaan.

4.5.3 Tuleva ECMAScript-standardi

ECMAScript-standardin seuraavaan versioon on ehdotettu uusia ominaisuuksia, jotka ovat olio-ohjelmoinnin ja tyypityksen kannalta melko merkittäviä. Standardia valmistelevan työryhmän laatimasta dokumentista [13] käy ilmi, että kielen tämänhetkinen prototyyppimalli säilyy entisellään, mutta sen lisäksi uudessa versiossa voidaan määritellä myös luokkia ja liittymiä kuten Javassa ja muuttujille voidaan asettaa tyyppirajoituksia. Kieleen on tulossa myös tuki CLOS:n mallin mukaisille geneerisille funktioille. Ehdotuksen perintämalli noudattaa Javan periaatteita: luokalla voi olla vain yksi yliluokka, mutta liittymien yhteydessä sallitaan myös moniperintä. Ehdotuksessa ei kuitenkaan kerrota, voiko liittymissä olla vakioita ja sisä-

luokkia kuten Javassa.

4.5.4 Pohdintaa

JavaScriptin prototyyppimallissa on melko hyvät edellytykset toimivalle moniperinnälle, mutta käytännössä sen mahdollisuudet jäävät hieman rajallisiksi. Dynaaminen tyypitys toimisi hyvin nimikonfliktitilanteissa, koska piirteiden tyypit eivät aiheuttaisi ristiriitoja, mutta tätä mahdollisuutta ei päästä hyödyntämään, koska viimeisenä määritelty piirre korvaa kaikki muut samannimiset piirteet. Tuleva ECMAScript-standardi ei tuo tähän tilanteeseen parannusta, vaan se pikemminkin lisää Javasta tuttujen perintäongelmien todennäköisyyttä.

JavaScriptiin saataisiin puhdas moniperintä, jos konstruktoreihin lisättäisiin mahdollisuus usealle prototyypille (ja uuden standardin mukaisessa kielessä luokille mahdollisuus useaan välittömään yliluokkaan). Moniperintä toisi kuitenkin mukanaan omat ongelmansa: nimikonfliktit, ristiriitaiset uudelleenmäärittelyt ja näiden ehkäisemiseksi vaadittavat toimenpiteet, jotka tekevät koodista vaikeampaa ymmärtää ja hallita. Tällä hetkellä kieli toimii hyvin sellaisissa tapauksissa, joissa moniperintä on pienimuotoista eikä nimikonflikteja esiinny. Oliorakenteen on kuitenkin tällöin hyvä olla melko pysyvä, koska JavaScriptin dynaamiset muokkausominaisuudet toimivat vain prototyypeille — eivät muunlaisille koosterakenteille.

5 Yhteenveto

Tässä tutkielmassa tarkasteltiin moniperinnän ja tyypityksen keskinäistä suhdetta olio-ohjelmoinnissa. Tavoitteena oli tutkia, miten ohjelmointikielen tyypitysjärjestelmä vaikuttaa moniperinnän toteutusmahdollisuuksiin ja toimivuuteen. Pohjustavana tutkimuskysymyksenä oli: ”Onko dynaaminen tyypitys moniperinnän kannalta parempi kuin staattinen tyypitys?” Vastausta etsittiin tutkimalla moniperinnän ja tyypityksen yhdistelmää viidessä melko erilaisessa olio-ohjelmointikielessä ja etsimällä niistä erityisesti tähän alueeseen liittyviä ongelmakohtia. Aihetta käsiteltiin lähinnä käytännön ohjelmoinnin ja asioiden käsitteellisen selkeyden näkökulmista. Toteutusteknisiä seikkoja ei huomioitu.

Vertailun jälkeen näyttäisi siltä, että kumpikaan tyypitystapa ei ole merkittävästi toista parempi. Dynaaminen tyypitys toimii hieman joustavammin nimikonfliktitilanteissa, mutta myös staattisesti tyypitetyssä kielessä ristiriidat on mahdollista ratkaista melko selkeällä tavalla, kuten Eiffel osoittaa. Kummallakin tyypityksellä voidaan päätyä sekaviin rakenteisiin: Javan moniperintä liittymien sisäluokkien kautta on yhtä lailla hankala hahmottaa kuin CLOS:n lisämetodien avulla toteuttu perittyjen metodien ketjutus. On kuitenkin huomattava, että näissä esimerkkitapauksissa ongelmat johtuvat itse asiassa moniperinnän toteutusratkaisuista eivätkä tyypityksestä. Voidaan todeta, että niin staattisella kuin dynaamisellakin tyypityksellä on moniperinnässä jokseenkin samat edut ja haitat kuin muussakin ohjelmoinnissa.

Perinnän ja tyypityksen yhdistelmissä äärimmäisyydet eivät välttämättä ole parhaita ratkaisuja. Voidaan vaatia, että kielen tyypitys on joko täysin staattinen tai täysin dynaaminen ja että kielessä on joko täysi moniperintä tai ei moniperintää lainkaan, mutta on myös muita varteenotettavia vaihtoehtoja. Eräs mahdollisuus on rajoittaa moniperintää niin, että konfliktitilanteet vähenevät. Javassa moniperintä on sallittua ainoastaan liittymille; tämä ratkaisu ei kuitenkaan toimi kovin hyvin. Squeakin käytöskomponenteissa taas on ollut ajatuksena, että vain operaatioita voi periä monesta lähteestä. Tämä lähestymistapa yhdistettynä Smalltalkin dynaamisuuteen vaikuttaisi melko toimivalta. CLOS:ssa puolestaan on mahdollista yhdistää staattisen ja dynaamisen tyypityksen hyvät puolet käyttämällä tyyppirajoitteita dynaamisen tyypityksen lomassa. Ajatus on ainakin teoriassa hyvä — CLOS:n tapauksessa sen toimivuus tosin riippuu vahvasti toteutuksesta, koska standardi ei määrittele täsmällistä toiminnallisuutta.

Viitteet

[1] *ANSI INCITS 319-1998 (R2002): Programming Language Smalltalk* (ANSI-standardi), ANSI, 1998.

[2] David Bank, *The Java Saga*, Wired Magazine, Issue 3.12 (December 1995). Saatavilla WWW-muodossa <URL: `http://www.wired.com/wired/archive/3.12/java.saga.html`>, viitattu 4.4.2008.

[3] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, David A. Moon, *Common Lisp Object System specification*, ACM SIGPLAN Notices, Volume 23, Issue SI (September 1988), s. 1–142, ACM Press, 1988.

[4] Timothy Budd, *An Introduction to Object-Oriented Programming*, 2nd ed., ISBN 0-201-82419-1, Addison-Wesley, 1997.

[5] Jon Byous, *Java technology: the early years*, toukokuu 1998. Saatavilla WWW-muodossa <URL: `http://java.sun.com/features/1998/05/birthday.html`>, viitattu 4.4.2008.

[6] Luca Cardelli, Peter Wegner, *On understanding types, data abstraction, and polymorphism*, ACM Computing Surveys (CSUR), Volume 17, Issue 4 (December 1985), s. 471–523, ACM Press, 1985.

[7] Robert Cartwright, Mike Fagan, *Soft typing*, ACM SIGPLAN Notices, Volume 39, Issue 4 (April 2004), s. 412–428, ACM Press, 2004. Artikkeli julkaistu alun perin vuonna 1991: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, s. 278–292, ACM Press.

[8] *Common Lisp HyperSpec*, 1996. Perustuu ANSI:n Common Lisp -standardiin X3.226. Saatavilla WWW-muodossa <URL: `http://www.lisp.org/HyperSpec/FrontMatter/index.html`>, viitattu 4.4.2008.

[9] Ole-Johan Dahl, Kristen Nygaard, *SIMULA: an ALGOL-based Simulation language*, Communications of the ACM, Volume 9, Issue 9 (September 1966), s. 671–678, ACM Press, 1966.

[10] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, Andrew P. Black, *Traits: A mechanism for fine-grained reuse*, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 28, Issue 2 (March 2006), s. 331–388, ACM, 2006.

[11] *ECMAScript Language Specification*, Standard ECMA-262, 3rd edition, Ecma International, 1999.

[12] *Eiffel: Analysis, Design and Programming Language*, Standard ECMA-367, 2nd edition, Ecma International, 2006.

[13] *Proposed ECMAScript 4th Edition - Language Overview*, ECMA-TC39-TG1, 2007. Saatavilla WWW-muodossa <URL: `http://www.ecmascript.org/es4/spec/overview.pdf`>, viitattu 4.4.2008.

[14] Adele Goldberg, David Robson, *Smalltalk-80: the language and its implementation*, ISBN 0-201-11371-6, Addison-Wesley, 1983.

[15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, *The Java Language Specification*, 3rd ed., ISBN 0321246780, Addison-Wesley, 2005.

[16] Alan C. Kay, *The early history of Smalltalk*, The second ACM SIGPLAN conference on History of programming languages, s. 69–95, ACM Press, 1993.

[17] Sonya E. Keene, *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, ISBN 0-201-17589-4, Addison-Wesley, 1989.

[18] Richard Kitchen, *Object Hierarchy and Inheritance in JavaScript*, 1997. Saatavilla WWW-muodossa <URL: `http://www.cs.rit.edu/~atk/JavaScript/manuals/jsobj/index.htm`>, viitattu 4.4.2008.

[19] Wilf R. LaLonde, John R. Pugh, *Inside Smalltalk, Volume 1*, ISBN 0-13-468430-3, Prentice Hall, 1990.

[20] Pasi Manninen, *Moniperintä C++-, Eiffel- ja Java-ohjelmointikielissä*, pro gradu -työ, Jyväskylän yliopisto, tietojenkäsittelytieteiden laitos, Jyväskylä, 2002.

[21] Bertrand Meyer, *Eiffel: programming for reusability and extendibility*, ACM SIGPLAN Notices, Volume 22, Issue 2 (February 1987), s. 85–94, ACM Press, 1987.

[22] Bertrand Meyer, *Eiffel: the language*, ISBN 0-13-247925-7, Prentice Hall, 1992.

[23] Bertrand Meyer, *Object-Oriented Software Construction*, 2nd ed., ISBN 0-13-629155-4, Prentice Hall, 1997.

[24] Kristen Nygaard, Ole-Johan Dahl, *The development of the SIMULA languages*, The first ACM SIGPLAN conference on History of programming languages, s. 245–272, ACM Press, 1978.

[25] Markku Sakkinen, *Inheritance and Other Main Principles of C++ and Other Object-oriented Languages*, Jyväskylä Studies in Computer Science, Economics and Statistics 20, ISBN 951-680-817-4, Jyväskylän yliopisto, 1992.

[26] Markku Sakkinen, *Olio-ohjelmointi*, luentomoniste, Jyväskylän yliopisto, 2005.

[27] Ghan Bir Singh, *Single versus multiple inheritance in object oriented programming*, ACM SIGPLAN OOPS Messenger, Volume 6, Issue 1 (January 1995), s. 30–39, ACM Press, 1995.

[28] Bjarne Stroustrup, *A history of C++: 1979–1991*, The second ACM SIGPLAN conference on History of programming languages, s. 271–297, ACM Press, 1993.

[29] Christopher Strachey, *Fundamental Concepts in Programming Languages*, Higher-Order and Symbolic Computation, Volume 13, Numbers 1-2/April, 2000, s. 11–49, Springer Netherlands, 2000.

[30] Antero Taivalsaari, *A Critical View of Inheritance and Reusability in Object-oriented Programming*, Jyväskylä Studies in Computer Science, Economics and Statistics 23, ISBN 951-34-0161-8, Jyväskylän yliopisto, 1993.

[31] David Ungar, Randall B. Smith, *Self: The Power of Simplicity*, Conference proceedings on Object-oriented programming systems, languages and applications, s. 227–242, ACM Press, 1987.

[32] Peter Wegner, *Classification in object-oriented systems*, Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming, s. 173–182, ISBN 0-89791-205-5, ACM Press, 1986.