

Luku 6

Periytyminen

For animals, the entire universe has been neatly divided into things to (a) mate with, (b) eat, (c) run away from, and (d) rocks.

– Equal Rites [Pratchett, 1987]

Yksi ihmismielelle ominainen piirre on pyrkimys kategorisoida — ryhmitellä asioita ja käsitteitä eri luokkiin niistä löytyvien yhtäläisyyksien perusteella. Tätä kategorisointia tekevät jo aivan pienet lapsetkin, ja ihmisten käyttämä kielikin perustuu suurelta osalta sanoihin, jotka eivät tarkoita yksittäistä asiaa vaan koko joukkoa keskenään jollain lailla samankaltaisia asioita.

Kategorisointia käytetään yleisesti myös tieteissä. Biologiassa *Carl von Linné* käytti tätä periaatetta 1700-luvulla jaotellessaan kasveja ja sieniä kategorioihin teoksessaan “*Systema Naturae*” [Linnaeus, 1748]. On kuitenkin huomattava, että tämä pyrkimys jaotteluun on nimenomaan ihmisen ajattelusta lähtöisin ja että todellinen maailma ei välttämättä aina taivu kovin hyvin tällaisiin malleihin.

Koska jaottelu aliryhmiin erilaisten yhteisten ominaisuuksien perusteella on niin luontevaa ihmisille, se on pyritty ottamaan käyttöön myös ohjelmoinnissa. Olijo-ohjelmoinnin painopiste on kahdessa asiassa: olioiden ulkoisessa käyttäytymisessä ja niiden sisäisessä toteutuksessa. Niinpä onkin luonnollista keskittyä kategorisoinnissa näihin aihealueisiin.

Ulkoisen käyttäytymisen jaottelulla saadaan ryhmiteltyä luokkia joukoiksi, jotka joiltain osin käyttäytyvät yhteneväisellä tavalla. Tästä on hyötyä, sillä tällöin sama ohjelmakoodi voi käsitellä kaikkien näiden luokkien olioita, koska olioiden käyttäytyminen on samantapaista.

Sisäisen toteutuksen jaottelussa taas on pyrkimys saada “tislatusi” eri luokkien yhteisiä osia yhteen paikkaan, jotta samaa ohjelmakoodia ei tarvitsisi kirjoittaa moneen kertaan. Tästä on tietysti selvää hyötyä ylläpidossa ja virheiden korjaamisessa, ja ohjelman kokokin voi pienentyä. Lisäksi koodiin lisätyt uudet luokat voivat ottaa suoraan käyttöönsä olemassa olevien luokkien ominaisuuksia, eli koodia voidaan käyttää uudelleen.

Tälle luokkien jaottelulle ja yhteisistä ominaisuuksista muodostuville “sukulaisuussuhteille” on annettu olio-ohjelmoinnissa nimi **periytyminen** (*inheritance*), ja monet pitävät sitä kapseloinnin ohella olio-ohjelmoinnin tärkeimpänä uutena ominaisuutena perinteisiin ohjelmointikieliin verrattuna. Periytyminen tarkasta merkityksestä ja käyttötavoista keskustellaan ja väitellään olioteoreetikkojen kesken kuitenkin edelleen (matematiikkaa pelkäämättömät voivat tutustua vaikka *Martín Abadin* ja *Luca Cardellin* kirjaan “Theory of Objects” [Abadi ja Cardelli, 1996]).

Periytymisessä on kyse suhteesta, jossa yksi luokka pohjautuu toiseen luokkaan ja perii sen ominaisuudet. Monissa “puhtaissa” oliokiellisissä jokainen käyttäjän määrittelemä luokka on aina periytetty jostain toisesta luokasta, ainakin kieleen erikseen määritellystä “kaikkien luokkien äidistä” — luokasta `Object` tai vastaavasta. Toisissa kielessä, kuten C++:ssa, tällaista periytymispakkoa ei kuitenkaan ole vaan oletusarvoisesti uusi luokka ei liity mitenkään jo olemassa oleviin luokkiin. Kuvassa 6.1 seuraavalla sivulla on lueteltu tässä teoksessa käytettyjä periytymiseen liittyviä termejä ja selitetty ne lyhyesti.

6.1 Periytyminen, luokkahierarkiat, polymorfismi

Varsinkin eurooppalaiset olio-ohjelmoinnin asiantuntijat korostavat periytymisessä olioiden ulkoisen käyttäytymisen — rajapintojen — kategorista jaottelua [Koskimies, 2000]. Kuvassa 6.2 sivulla 145 on

Termi	Selitys
Periytyminen (<i>inheritance</i>), (johtaminen)	Uuden luokan muodostuminen olemassa olevan luokan pohjalta niin, että uusi luokka sisältää kaikki toisen luokan ominaisuudet.
Kantaluokka (<i>base class</i>), (yliluokka (<i>superclass, parent class</i>))	Alkuperäinen luokka, jonka ominaisuudet periytyvät uuteen luokkaan.
Aliluokka (<i>subclass</i>), (periytetty/johdettu luokka (<i>derived class</i>))	Uusi luokka, joka perii kantaluokan ominaisuudet ja voi tämän lisäksi sisältää uusia ominaisuuksia.
Periytymishierarkia (<i>inheritance hierarchy</i>), (luokkahierarkia, (<i>class hierarchy</i>))	Kantaluokista ja niiden aliluokista muodostuva puumainen rakenne, jossa puussa alempana olevat luokat perivät ylempänä olevien ominaisuudet. Katso kuva 6.4 sivulla 149.
Esi-isä (<i>ancestor</i>)	Periytymishierarkiassa luokan kantaluokka, kantaluokan kantaluokka, tämän kantaluokka jne. ovat luokan esi-isiä.
Jälkeläinen (<i>descendant</i>)	Kaikki kantaluokasta periytetyt luokat, niistä periytetyt luokat jne. ovat luokan jälkeläisiä.

— KUVA 6.1: Periytymiseen liittyvää terminologiaa —

esitetty eliöiden toimintaa mallintavan ohjelman **periytymishierarkiaa** (*inheritance hierarchy*), tosin varsin rajoitetusti.

Tässä esimerkissä vain luokat Homesieni, Koira, Ihminen, Kana ja Satakieli ovat varsinaisia “todellisia eliöitä mallintavia” luokkia, joista ohjelmassa tehdään olioita. Luokat Eliö, Sieni, Eläin, Nisäkäs ja Lintu puolestaan vain kuvaavat todellisten luokkien “yläkäsittettä”. Niitä käytetään ryhmittelemään todellisia luokkia kategorioihin, joihin kuuluvien olioiden rajapinta ja käyttäytyminen ovat jollain tavalla yhteneväisiä.

Tällaisia luokkia, joista ei ole mielekästä tehdä olioita, mutta joita silti tarvitaan kuvaamaan ohjelmassa esiintyvien todellisten luokkien rajapintoja ja niiden suhteita, kutsutaan **abstrakteiksi kantaluokiksi** (*abstract base class*). UML:ssä abstraktit kantaluokat usein merkitään merkinnällä “{abstract}” kuten kuvassa. Samoin merkinnällä

aalisen moniperiytyamisen käyttöä ja tekee luokkien välisen vastuun- jaon vaikeammaksi. Se on kuitenkin välttämätön rajoitus olioajattelun kannalta. Jos kaksi toisistaan tietämätöntä kantaluokkaa joutuu jakamaan yhteisen kantaluokkaosan, on luonnollista että nämä luokat moniperiyttävä aliluokka määrää, miten tämä yhteinen kantaluokka alustetaan. Helpommaksi tilanne muuttuu, jos jaetulla kantaluokalla on vain oletusrakentaja, jolloin rakentajan parametreista ei tarvitse välittää aliluokissa.

Olioiden tuhoamisen yhteydessä vastaavia ongelmia ei tule, vaan jaetun kantaluokan purkajaa kutsutaan normaalisti kertaalleen ilman, että aliluokkien täytyy ottaa siihen kantaa.[Ⓔ]

Kaiken kaikkiaan toistuva kantaluokka aiheuttaa moniperiytymiseen niin paljon monimutkaisuutta, rajoituksia ja ongelmia, että jotkut ovat antaneet tällaiselle periytymishierarkialle osuvan nimen “*Dreaded Diamond of Death*”. Varsinkin virtuaalista moniperiytymistä pitäisikin yleensä välttää, ellei tiedä tarkkaan mitä on tekemässä. Joskus se on kuitenkin kelvollinen apukeino ohjelmoijan työkalupakissa.

6.9 Periytyminen ja rajapintaluokat

On varsin yleistä, että abstrakteissa kantaluokissa määritellään luvun alun eliöesimerkin tapaan pelkästään puhtaita virtuaalifunktioita, jolloin abstraktit kantaluokat eivät sisällä mitään muuta kuin rajapinnan määrittelyjä. Tällöin puhutaan usein **rajapintaluokista** (*interface class*). Rajapintaluokissa ei siis ole jäsenmuuttujia eikä jäsenfunktioiden toteutuksia, vaan ainoastaan (yleensä julkisen) rajapinnan määrittely. Joissain oliokielissä, kuten Javassa, tällaisille puhtaille rajapinnoille on oma syntaksinsa eikä niitä edes varsinaisesti lasketa luokiksi.

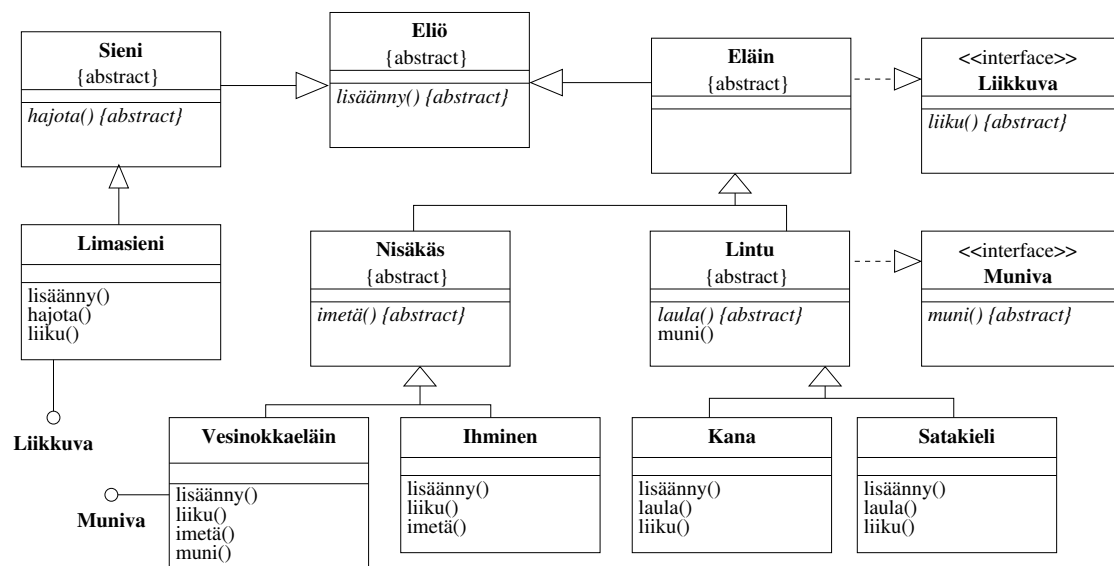
.....
[Ⓔ]Tarkasti ottaen C++-standardi joutuu ottamaan kantaa virtuaaliseen periytymiseen purkajien yhteydessä, kun se määrittelee purkajien keskinäisen kutsujärjestyksen. Tällä ei kuitenkaan normaalisti ole mitään merkitystä, ja purkajia edelleen kutsutaan käänteisessä järjestyksessä rakentajiin verrattuna.

6.9.1 Rajapintaluokkien käyttö

Rajapintaluokat ovat varsin käteviä, koska niiden avulla voidaan käyttäjälle paljastaa luokkahierarkiasta pelkkä hierarkkinen rajapinta ja kätkeä itse rajapintafunktioiden toteutus konkreettisiin luokkiin. Rajapintaluokkia ja dynaamista sitomista käyttämällä ohjelmoija voi lisäksi itse päättää, millä hierarkiatasolla olioita käsittelee. Joitain funktioita kiinnostaa vain, että niiden parametrit ovat mitä tahansa eläimiä, johonkin tietorakenteeseen talletetaan mitä tahansa sieniä ja niin edelleen.

Usein tulee eteen tilanne, jossa kaikkia haluttuja rajapintoja ei voi millään panna samaan luokkahierarkiaan, koska itse rajapinnat ovat toisistaan riippumattomia ja konkreettisten luokkien rajapinnat ovat erilaisia yhdistelmiä näistä rajapinnoista. Tällaisessa tapauksessa konkreettiset luokat pitäisi pystyä koostamaan erilaisista rajapintakomponenteista luokkahierarkiasta riippumatta. Kuva 6.10 näyttää esimerkin useiden toisistaan riippumattomien rajapintojen käytöstä.

Eri oliokielissä ongelma on ratkaistu eri tavoilla. Ongelmaa ei ole niissä kielissä, joissa ei ole käännoisaikaista tyyppitystä, koska itse rajapintaluokan käsitettä ei tarvita. Esimerkiksi Smalltalkissa miltä ta-



— KUVA 6.10: Luokat, jotka toteuttavat erilaisia rajapintoja —

hansa oliolta voidaan pyytää mitä tahansa palvelua ja vasta ohjelman ajoaikana tarkastetaan, pystyykö olio tällaista palvelua tarjoamaan.

Javan erilliset rajapinnat tarjoavat elegantin tavan yhdistellä rajapintoja todellisissa luokissa. Java ei rajoita tällaisten rajapintojen määrää yhteen, vaan luokka voi luetella mielivaltaisen määrän rajapintoja, jotka se toteuttaa. Tällä tavoin luokat voivat luokkahierarkiasta riippumatta toteuttaa erilaisia rajapintoja. Näiden rajapintojen avulla voidaan sitten käsitellä kaikkia rajapinnan toteuttavia luokkia samassa koodissa, koska luokista ei tarvita muuta tietoa kuin se, että ne toteuttavat halutun rajapinnan. Lista 6.15 näyttää osan kuvan 6.10 luokkien toteutuksesta Javalla.

```

..... Liikkuva.java .....
1 public interface Liikkuva
2 {
3     public void liiku(Sijainti paamaara);
4 }
..... Muniva.java .....
1 public interface Muniva
2 {
3     public void muni();
4 }
..... Elain.java .....
1 public abstract class Elain extends Elio implements Liikkuva
2 {
3     :
4 }
..... Vesinokkaelain.java .....
1 public class Vesinokkaelain extends Nisakas implements Muniva
2 {
3     public void lisaanny() { }
4     public void liiku(Sijainti paamaara) { }
5     public void imeta() { }
6     public void muni() { }
7     :
8 }

```

LISTAUS 6.15: Erilliset rajapinnat Javassa

6.9.2 C++: Rajapintaluokat ja moniperiytyminen

C++:ssa rajapintaluokkia mallinnetaan abstrakteilla kantaluokilla ja moniperiytyemisellä. Mikäli todellinen luokka toteuttaa useita toisistaan riippumattomia rajapintoja, se periytetään kaikista rajapinnat määräävistä abstrakteista kantaluokista. Tällä tavoin saadaan aikaan useita juuriluokkia sisältävä luokkahierarkia, jossa rajapintaluokat ovat kaikkien rajapinnan toteuttavien todellisten luokkien kantaluokkia. Listaus 6.16 näyttää osan kuvan 6.10 toteutuksesta C++:lla.

Mikäli abstraktit kantaluokat sisältävät ainoastaan puhtaita virtu-

```

17 class Liikkuva
18 {
19 public:
20     virtual ~Liikkuva();
21     virtual void liiku(Sijainti paamaara) = 0;
22 };
    :
23 class Muniva
24 {
25 public:
26     virtual ~Muniva();
27     virtual void muni() = 0;
28 };
    :
29 class Elain : public Elio, public Liikkuva
30 {
31 public:
32 private:
33 };
    :
47 class Vesinokkaelain : public Nisakas, public Muniva
48 {
49 public:
50     virtual ~Vesinokkaelain();
51     virtual void lisaanny();
52     virtual void liiku(Sijainti paamaara);
53     virtual void imeta();
54     virtual void muni();
55 };

```

LISTAUS 6.16: Rajapintaluokkien toteutus moniperiytyemisellä C++:ssa

aalifunktioita ja ovat näin pelkkiä rajapintaluokkia, ei moniperiyty-
misen käytöstä aiheudu yleensä ongelmia. Mikäli moniperiyty-
misen kantaluokat sen sijaan sisältävät myös rajapintojen toteutuksia ja
jäsenmuuttujia, moniperiytyminen aiheuttaa yleensä enemmän on-
gelmia kuin ratkaisee, kuten aiemmin on todettu. C++:ssa moniperiy-
tyymisen järkevä käyttö on jätetty ohjelmoijan vastuulle, eikä kieli itse
yritä varjella ohjelmoijaa siinä esiintyviltä vaaroilta.

Rajapintaluokkien toteuttaminen C++:ssa moniperiytyymisen avulla
aiheuttaa kuitenkin jonkin verran kömpelyyttä ohjelmaan. Ensinnä-
kin, koska sekä normaali periytyminen että rajapinnan toteuttaminen
tehdään kielessä periytymissyntaksilla, ei luokan esittelystä suoraan
näe, mitkä sen kantaluokista ovat puhtaita rajapintoja ja mitkä sisäl-
tävät myös toteutusta. Tähän ei C++:ssa ole muuta ratkaisukeinoja kuin
nimetä rajapintaluokat niin, että käy selvästi ilmi niiden olevan pelk-
kiä rajapintoja.

Rajapintaluokat, rakentajat ja virtuaalipurkaja

Rajapintaluokat ovat C++:ssa normaaleja luokkia, joten niilläkin on ra-
kentaja, jota kutsutaan aliluokan rakentajasta. Käytännössä tämä ra-
kentaja on kuitenkin aina tyhjä, koska rajapintaluokat nimensä mu-
kaan määrittävät vain rajapinnan eivätkä sisällä jäsenmuuttujia tai
toiminnallisuutta. Tämän vuoksi rakentajien kirjoittaminen rajapin-
taluokille olisi turhauttavaa. Tässä onneksi C++:n normaalisti vaaralli-
nen automatiikka auttaa.

Aliluvussa 3.4.1 todettiin, että jos luokalle ei kirjoiteta yhtään ra-
kentajaa, tekee kääntäjä sinne automaattisesti ”tyhjän” oletusrakenta-
jan. Vastaavasti aliluvussa 6.3.2 kävi ilmi, että jos kantaluokan raken-
tajan kutsu jätetään pois aliluokan rakentajan alustuslistasta, kutsuu
kääntäjä automaattisesti kantaluokan oletusrakentajaa. Näiden omi-
naisuuksien ansiosta rajapintaluokkaan ei tarvitse kirjoittaa rakenta-
jaa ollenkaan, koska tällöin kääntäjä automaattisesti luo tyhjän ole-
tusrakentajan ja kutsuu sitä aliluokissa. Rajapintaluokat ovat C++:ssa
lähes ainoa tilanne, jolloin luokalle ei tyyliohjeista poiketen kannata
kirjoittaa rakentajaa.

C++:ssa kantaluokkien purkajien tulisi olla virtuaalisia (alilu-
ku 6.5.5), jotta niistä periytettyjen luokkien oliot tuhottaisiin aina
oikein. Rajapintaluokat ovat kantaluokkia, joten tämä sääntö kos-
kee myös niitä. Ikävä kyllä, jos rajapintaluokan esittelyssä esitellään

purkaja virtuaaliseksi, pitää tälle purkajalle kirjoittaa myös toteutus, vaikka kyseessä onkin pelkkä rajapintaluokka ilman toiminnallisuutta tai dataa, ja näin purkajan toteutus on aina tyhjä.

Ärsyttäväksi tämän vaatimuksen virtuaalipurkajan toteutuksesta tekee se, että koska rajapintaluokassa ei muuten ole toiminnallisuutta, olisi luontevaa kirjoittaa sille pelkkä otsikkotiedosto (.hh) ja jättää varsinainen kooditiedosto (.cc) kokonaan kirjoittamatta. Purkajan toteutus taas normaalisti kirjoitettaisiin juuri kooditiedostoon. Tähän ongelmaan löytyy onneksi ratkaisu. C++ antaa mahdollisuuden kirjoittaa jäsenfunktion toteutuksen suoraan luokkaesittelyyn *sisälle* (samaan tapaan kuin Javassa tehdään). Tämä ei ole normaalisti hyvää tyyliä, koska luokan toteutusta ja esittelyä ei ole syytä sotkea keskenään. Lisäksi tällaiset luokan esittelyyn upotetut jäsenfunktiot optimoidaan aina kuten **inline**-funktiot (liitteen aliluku A.2), mikä ei yleisessä tapauksessa ole välttämättä hyvä asia.

Koska kuitenkin rajapintaluokan purkaja on aina tyhjä, voi tästä tyylisäännöstä ainakin tämän kirjan kirjoittajien mielestä poiketa tässä tilanteessa. Näin rajapintaluokan purkaja saadaan kirjoitettua kokonaan otsikkotiedostoon eikä erillistä toteutustiedostoa tarvita. Listaus 6.17 näyttää esimerkkinä rajapintaluokan `Liikkuva` esittelyyn näin kirjoitettuna.

6.9.3 Ongelmia rajapintaluokkien käytössä

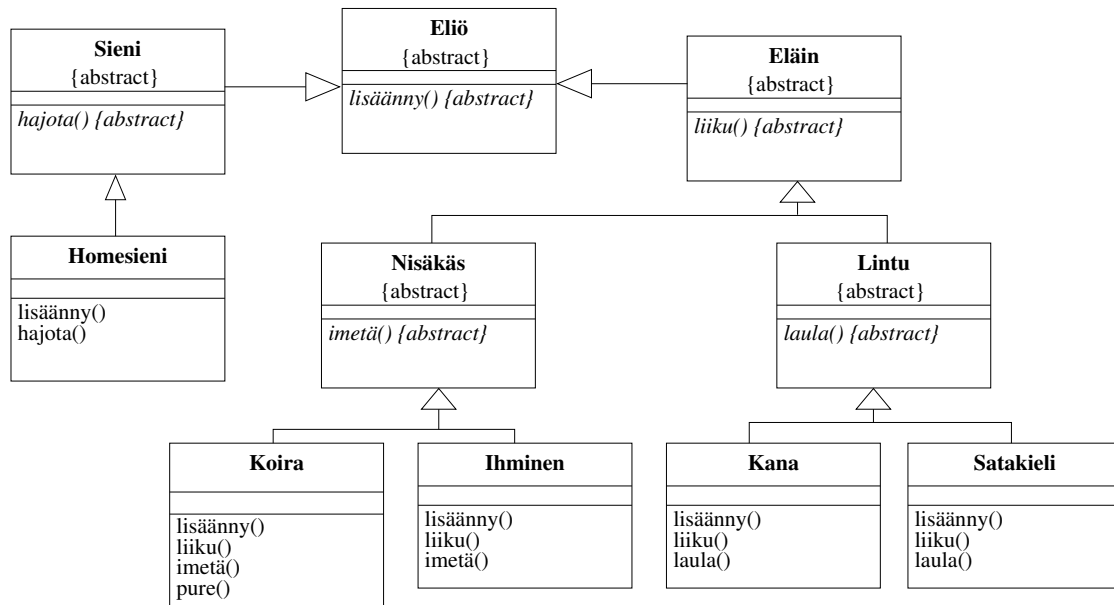
Erilliset rajapinnat tai rajapintaluokat selkeyttävät luokkahierarkiaa ja mahdollistavat kätevästi sen, että saman rajapinnan toteuttavia olioita voidaan käsitellä rajapintaosoittimen tai -viitteen avulla riippumatta luokkien sijainnista luokkahierarkiassa. Rajapintaluokat eivät kuitenkaan ratkaise kaikkia ongelmia.

```

1 class Liikkuva
2 {
3 public:
4     // Kääntäjä tuottaa automaattisesti tyhjän oletusrakentajan
5     virtual ~Liikkuva() {} // Upotettu tyhjä virtuaalipurkaja (inline)
6     virtual void liiku(Sijainti paamaara) = 0;
7 };

```

— **LISTAUS 6.17:** Rajapintaluokan purkaja esittelyyn upotettuna —



— KUVA 6.2: Eliöitä mallintavan ohjelman periytymishierarkiaa —

“{abstract}” voidaan vielä merkitä ne rajapinnan jäsenfunktiot, joille abstrakti kantaluokka ei tarjoa toteutusta.

Periytymistä käytetään kuvaamaan luokkien välisiä suhteita. Luokka Eliö kuvaa kaikkia ohjelmassa esiintyviä eliöitä. Sen rajapinta sisältää kaikille eliöille yhteisen rajapinnan, tässä esimerkissä lisääntymisen. Kaikkia esimerkin todellisia olioita voi pyytää lisääntymään, ja jokaisen eliön lisääntymispalvelu näyttää ulospäin täsmälleen samanlaiselta. Sen sijaan lisääntymisen varsinainen toteutus saattaa hyvinkin vaihdella luokasta toiseen — on varsin todennäköistä, että homesieni ja koira lisääntyvät eri tavalla!

Esimerkissä eliöt jaetaan kahteen alikategoriaan, sieniin ja eläimiin. Nämä eroavat toisistaan ulkoisesti siinä, että sieniä voi pyytää hajottamaan eloperäistä materiaalia, eläimiä puolestaan liikkumaan paikasta toiseen. Luokat Sieni ja Eläin on **periytetty** (*derived*) luokasta Eliö. Tällöin niiden rajapintaan kuuluu niiden omien palveluiden lisäksi automaattisesti myös luokan Eliö rajapinta — sienten ja eläinten rajapinta on siis laajennettu eliöiden rajapinnasta. Jälleen kysymys on vain rajapinnasta, ja jokainen todellinen sieniluokka voi toteuttaa hajottamispalvelun haluamallaan tavalla.

Samalla tavoin jaetaan eläimet vielä nisäkkäisiin, joita voi käskeä imettämään, sekä lintuihin, joita voi pyytää laulamaan. Viimein näistä luokista on periytetty todelliset ohjelmassa esiintyvät eläinluokat Koira, Ihminen, Kana ja Satakieli.

Periytyksen hyötynä on, että hierarkia antaa mahdollisuuden puhua esimerkiksi kaikista nisäkkäistä luokkaa Nisäkäs käyttämällä. Jos myöhemmin ohjelmassa tulee esimerkiksi tarve lisätä kaikkien nisäkkäiden rajapintaan uusi palvelu — vaikkapa synnyttäminen —, tämä käy yksinkertaisesti lisäämällä kyseinen operaatio luokan Nisäkäs rajapintaan. Tämän jälkeen täytyy tietysti vielä toteuttaa synnyttäminen kussakin nisäkkäessä lajille ominaisella tavalla.

Vastaavasti jos ohjelmassa on funktio, jonka tehtävänä on siirtää sille parametrina annettuja eläimiä, tämä funktio voi ottaa parametrinaan yksinkertaisesti viitteen Eläin-luokan olioon. Koska sekä Koira, Ihminen, Kana että Satakieli ovat luokkahierarkiassa eläimiä, siirtymisfunktiolle voi tällöin antaa siirrettäväksi *minkä tahansa* eläimen.

Siirtymisfunktion itsensä ei tarvitse tietää yksityiskohtia siitä, minkä eläinlajin edustajaa se on siirtämässä. Sille riittää tieto siitä, että koska kyseessä on eläin, sen julkisessa rajapinnassa on tarvittava palvelu eläimen siirtämiseen. Tällaista tilannetta, jossa funktio hyväksyy eri tyyppisiä parametreja, kutsutaan **polymorfismiksi** (*polymorphism*) eli “monimuotoisuudeksi”. Kääntäjä voi luokkahierarkian avulla lisäksi tarkastaa, että siirtymisfunktiolle annetaan siirrettäväksi vain sellaisia olioita, joita todella voi siirtää — ei esimerkiksi home-sieniä. Kielissä, joissa ei ole vahvaa tyyppitystä, tarkastetaan sopivan jäsenfunktion löytyminen yleensä vasta ajoaikana. Tällaisissa kielissä luokkahierarkian käyttö rajapintojen luokitteluun on tarpeetonta, koska esimerkiksi siirtymisfunktiolle voisi antaa parametrina minkä tahansa olion, ja jos kyseinen olio ei osaa siirtyä, annetaan ajoaikainen virheilmoitus. Esimerkiksi Smalltalk kuuluu tällaisiin oliokieliin.

Luokkahierarkioiden etuna on vielä se, että jokaisessa todellisessa luokassa rajapintafunktio voidaan tarvittaessa toteuttaa eri tavalla. Edellä mainittu siirtymisfunktio pystyy siirtämään mitä tahansa eläimiä kutsumalla näiden “liiku”-palvelua. Vaikka siirtymisfunktio ei välitäkään siitä, minkä lajin eläimestä on kysymys, voi jokainen eläin silti toteuttaa liikkumisen omalla tavallaan — linnut lentämällä, koira jolkottamalla jne. Näin sama siirtymisfunktiossa oleva “liiku”-palvelun kutsu voi ajoaikana aiheuttaa palvelupyynnön vastaanottajaolion luokasta riippuen erilaisen toiminnon. Tätä kutsutaan

puolestaan **dynaamiseksi sitomiseksi** (aliluku 6.5.2).

Periytymiseen ja hierarkioihin perustuvassa kategorisoinnissa korostuu olion tarjoaman palvelun ja sen toteutuksen ero. Kaikki Eläinluokasta periytetyt luokat sisältävät palvelun “liiku” ja lupaavat, että sitä kutsumalla olio liikkuu paikasta toiseen. Rajapinta ei kuitenkaan lupaa mitään siitä, *millä tavalla* liikkuminen tapahtuu. Tämän voi ajatella myös niin, että hierarkiassa alempana olevat luokat **erikoistavat** (*specialize*) ylempänä määrättyjen palveluiden toimintaa. Erikoistaminen voi olla myös useampivaiheista, esimerkiksi luokassa Lintu saatettaisiin erikoistaa liikkumista määräämällä, että se tapahtuu lentäen. Siitä huolimatta kanat ja satakielet voisivat vielä erikoistaa tätä lisää toteuttamalla lentämisen eri tavalla.

6.2 Periytyminen ja uudelleenkäyttö

Luokkasuunnittelun edetessä tulee vastaan tilanteita, joissa huomataan osan luokkaa (attribuuttien tai toiminnallisuuden) olevan samanlainen useassa luokassa. Esimerkiksi jos mietimme muita grafiikkaluokkia kuin aikaisemmin esimerkkinä ollut Pistettä, niin keksimme ehkä ympyrän, viivan ja valokuvan. Näillä kaikilla on yhteisenä ominaisuutena tieto näkyvyydestä (onko mallinnettu grafiikkaolio piirrettynä näyttölaitteelle), joka ilmaistaan luokan attribuuttina ja sen käyttöön liittyvinä rajapintafunktioina. Attribuutin ilmaisema tilatieto ja siihen liittyvä rajapinta on kaikilla hierarkian luokilla sama.

Ohjelmiston ylläpidettävyys paranee, jos kaikille luokille yhteiset attribuutit ja rajapintafunktiot toteutetaan vain kerran ja sama toteutus on kaikkien luokkien käytössä. Oliosunnittelussa tämä **yleistämisen** (*generalization*) saadaan aikaiseksi määrittelemällä yhteiset osat yhteen luokkaan, josta muut ominaisuutta tarvitsevat luokat *perivät toteutuksen* osaksi itseään. Yhteinen toiminnallisuus tavallaan nostetaan periytymishierarkiassa ylemmälle tasolle yhteiseen kantaluokkaan.

Kuvassa 6.3 seuraavalla sivulla on määritelty kantaluokka Näkyvyys, jonka vastuulle on merkitty tietämys siitä, onko grafiikkaolio näkyvillä näyttölaitteella (attribuutti), ja siihen liittyvät rajapintafunktiot. Kantaluokasta periytetyt luokat perivät *kaikki* kantaluokan ominaisuudet osaksi itseään. Tämä tarkoittaa sitä, että kun Ympy-