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. Olio-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 "tislatuksi" 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. Periytymisen 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" oliokielissä jokainen käyttäjän määrittelemä luokka on aina periytetty jostain toisesta luokasta, ainakin kieleen erikseen määritellystä "kaikkien luokkien äidistä" — luokasta `Object` tai vastaavasta. Toisissa kielissä, 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

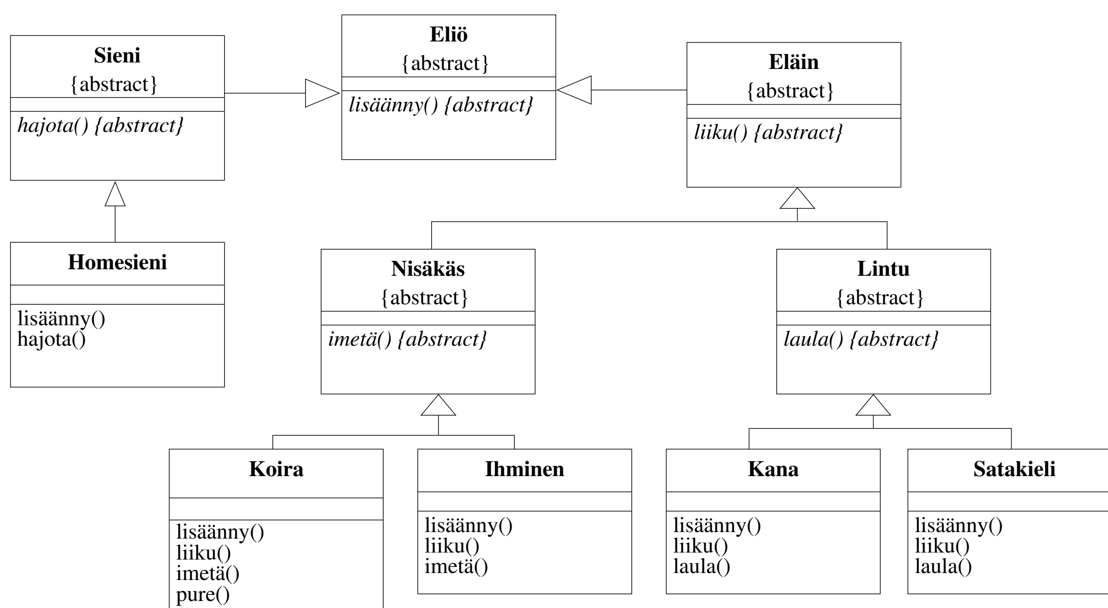
Termi	Selitys
Periytyminen (*inheritance*), (johtaminen)	Uuden luokan muodostuminen olemassa olevan luokan pohjalta niin, että uusi luokka sisältää kaikki toisen luokan ominaisuudet.
Kantaluokka (*base class*), (yliluokka (*superclass, parent class*))	Alkuperäinen luokka, jonka ominaisuudet periytyvät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



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.

Periytymisen 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 nisäkkäässä 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 homesieniä. Kielissä, joissa ei ole vahvaa tyypitystä, 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.

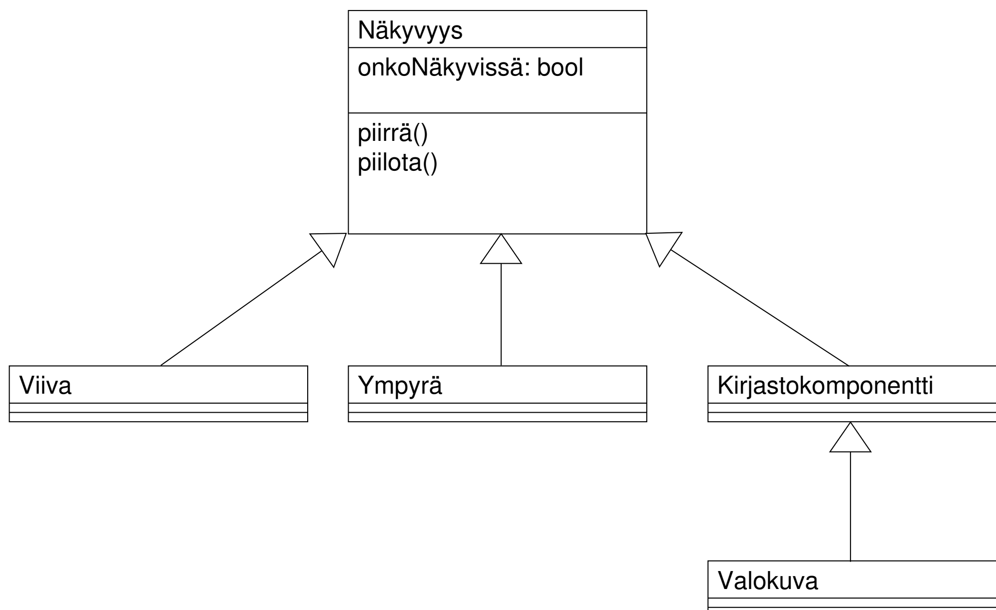
Ohjelmiston ylläpidettävyys paranee, jos kaikille luokille yhteiset attribuutit ja rajapintafunktiot toteutetaan vain kerran ja sama toteutus on kaikkien luokkien käytössä. Oliosuunnittelussa tämä **yleistäminen** (*generalization*) saadaan aikaiseksi määrittelemällä yhteiset osat yhteen luokkaan, josta 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-

rä-luokasta tehdään olio, se sisältää sekä Näkyvyys-luokan että ympyrän ominaisuudet. Tarkasteltaessa periytymistä periytetyn luokan kannalta se on erikoistettu versio kantaluokasta.

Luokkien periyttämistä voidaan jatkaa periytymishierarkiassa eteenpäin. Jos Kirjastokomponentilla on hyvin samanlaiset ominaisuudet kuin näytöllä esitettävällä valokuvalla, voimme periyttää luokan Valokuva Kirjastokomponentista. Valokuva-luokalla on nyt kaikki samat ominaisuudet kuin sen molemmilla kantaluokilla — myös näkyvyyteen liittyvät.

Periytyminen liittyy läheisesti ohjelmakoodin uudelleenkäyttöön. Aikaisemmin olleesta luokasta (mahdollisesti ostetussa komponentissa tai aikaisemmassa projektissa) saadaan käytetyksi kaikki halutut ominaisuudet, ja johdetussa uudessa versiossa tarvitsee ainoastaan lisätä ja muuttaa tarvittavat uudet osuudet. Tämä vahva ominaisuus sisältää myös riskin: pitkät (ja huonosti dokumentoidut) periytymishierarkiat ovat usein perinteistä vaikealukuisempaa ohjelmakoodia, koska hierarkian eri osat (luokat) voivat sijaita eri puolilla ohjelmakoodia (esimerkiksi eri tiedostoissa).

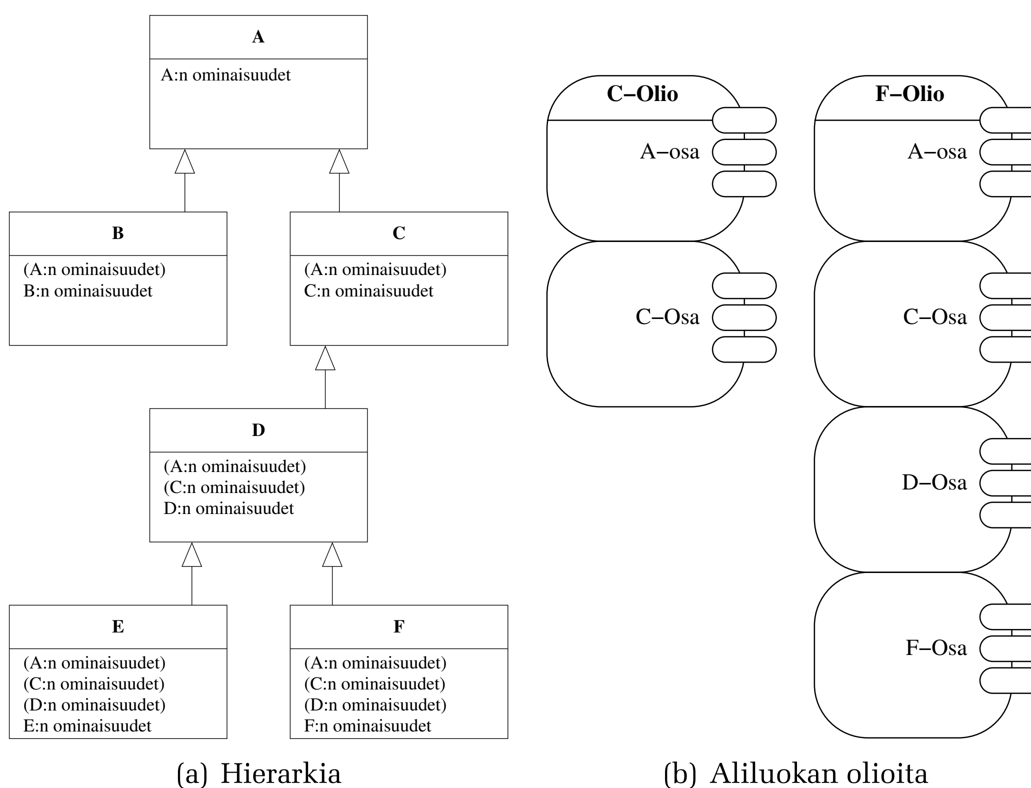


KUVA 6.3: Näkyvyyttä kuvaava kantaluokka ja periytettyjä luokkia

6.3 C++: Periytymisen perusteet

C++:ssa periytyminen tarkoittaa, että kielessä on mahdollista luoda uusi luokka jo olemassa olevaan luokkaan perustuen niin, että uuden luokan oliot perivät automaattisesti kaikki toisen luokan ominaisuudet, niin rajapinnan kuin sisäisen toteutuksenkin. Kuva 6.4 näyttää, mistä periytymisessä yksinkertaistettuna on kyse. Jokainen aliluokka perii kaikki esi-isiensä ominaisuudet ja voi lisäksi laajentaa ja rajoitetusti muokata niitä. Kuvaan on piirretty myös aliluokan olioita, joista näkyy olioiden "kerrosrakenne" — oliot ovat ikään kuin kantaluokan olioita, joihin on liimattu kiinni aliluokkien vaatimat lisäosat.

C++:ssä on myös mahdollista periyttää luokka useammasta kuin yhdestä luokasta. Tämä **moniperiytyminen** (*multiple inheritance*) on varsin kiistelty ominaisuus, ja sen käyttöä ei yleensä suositella kuin tiettyihin tarkoituksiin. Moniperiytymistä käsitellään jonkin verran



(a) Hierarkia (b) Aliluokan olioita

KUVA 6.4: Periytymishierarkia ja oliot [Koskimies, 2000]

aliluvussa 6.7. Aliluvussa 6.9.2 käsitellään myös yhtä moniperiytymisen käyttötapaa — rajapintaluokkia.

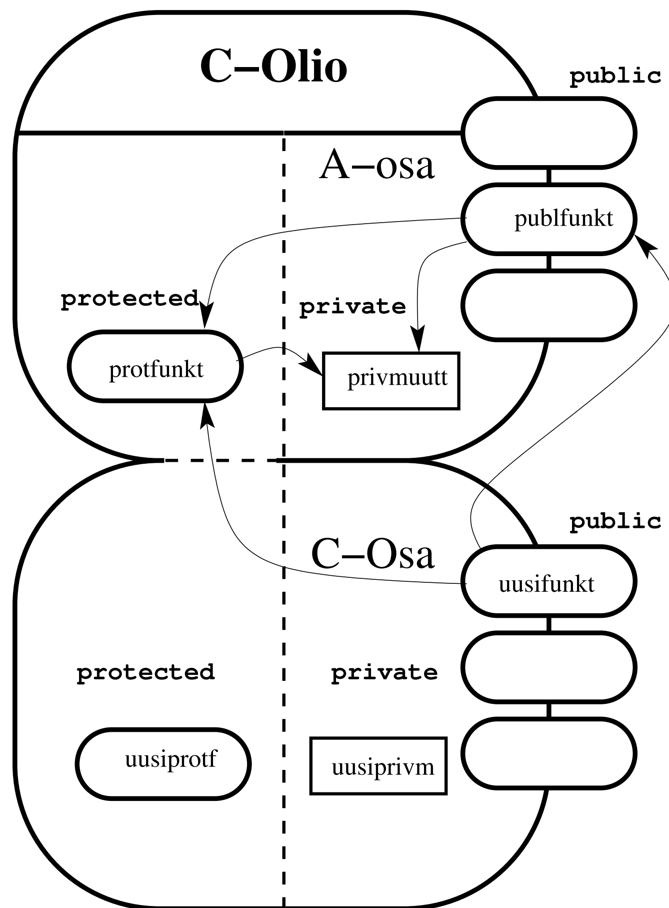
Periytymisen syntaksi on yksinkertainen: aliluokkaa esiteltäessä merkitään luokan nimen jälkeen kaksoispiste ja sen jälkeen periytymistyyppi (lähes aina **public**) ja kantaluokan nimi (moniperiytymisen yhteydessä näitä periytymistyyppi-kantaluokka-pareja on useita pilkulla toisistaan erotettuina). Listauksessa 6.1 on esimerkkinä kuvan 6.4 luokkahierarkiaa C++:lla toteutettuna. C++:ssa on mahdollista public-periytymisen lisäksi myös private- ja protected-periytyminen, mutta niiden käyttö on varsin harvoin olio-ohjelmoinnissa tarpeellista, eikä niitä käsitellä tässä teoksessa.

6.3.1 Periytyminen ja näkyvyys

Periytymisen yhteydessä aliluokka perii kantaluokasta niin ulospäin näkyvän rajapinnan kuin sisäisen toteutuksenkin. Perittyjen osien näkyvyys aliluokan oliossa on kuitenkin osin erilainen kuin kantaluokassa. Kuva 6.5 seuraavalla sivulla kuvaa aliluokan pääsyä kantaluokan eri osiin. Alla on luettelo, josta käy ilmi eri näkyvyysmääreiden vaikutus periytymiseen.

```
1  class A
2  {
3  // Luokan A ominaisuudet
4  };
5  class B : public A
6  {
7  // Luokan B A:han lisäämät ominaisuudet
8  };
9  class C : public A
10 {
11 // Luokan C A:han lisäämät ominaisuudet
12 };
13 class D : public C
14 {
15 // Luokan D C:hen lisäämät ominaisuudet
16 };
      ⋮
```

LISTAUS 6.1: Periytymisen syntaksi C++:lla



KUVA 6.5: Periytyminen ja näkyvyys

- **public:** Kantaluokan public-osat — sen ulkoinen rajapinta — ovat myös aliluokassa public-puolella. Näin aliluokan ulkoisessa rajapinnassa on kaikki se, mitä kantaluokassakin, sekä lisäksi aliluokan määrittelemät uudet rajapintafunktiot.

- **private:** Vaikka kantaluokan private-osa periytyykin aliluokkaan, ei niihin pääse käsiksi aliluokan jäsenfunktioista. Tämä johtuu siitä näkökannasta, että private-osat ovat kantaluokan sisäisenä toteutuksena kantaluokan oma asia, eikä aliluokan tarvitse päästä niihin käsiksi — eihän private-osa muutenkaan näy luokasta ulos. Aliluokan jäsenfunktiot voivat tietysti välillisesti käyttää kantaluokan private-osia kutsumalla kantaluokan rajapintafunktioita.

- **protected:** Tähän saakka protected-määreestä ei ole mainittu juuri mitään muuta, kuin että se on hyvin samantapainen kuin private. Protected-määreen käyttö liittyykin nimenomaan periytymiseen. Kantaluokan protected-osa periytyy aliluokan protected-osaksi. Näin protected-osa on kuin private-osa siinä mielessä, ettei se kuulu luokan ulkoiseen rajapintaan, mutta se on kuitenkin aliluokkien koodin käytettävissä. Protected-osan tehtävä onkin laajentaa kantaluokan rajapintaa aliluokan suuntaan tarjoamalla tälle erityisrajapinnan. Tällä tavoin kantaluokka voi tarjota aliluokille luokan laajentamiseen tarvittavia apufunktioita, joita ei kuitenkaan haluta yleiseen käyttöön. Kuten public-osaankin, protected-osaan tulee kirjoittaa vain jäsenfunktioita, vaikkei C++:n syntaksi tätä rajoitakaan.

Edellä olevista näkyvyyssäännöistä aloittelijaa ihmetyttää usein kantaluokan private-osan "eristäminen" aliluokalta. Kapseloinnin kannalta tämä C++:n (ja Javan) ratkaisu on kuitenkin oikea, koska aliluokkaa ei pitäisi toteuttaa niin, että se riippuu kantaluokan sisäisestä toteutuksesta. Mikäli aliluokan täytyy saada suorittaa sellaisia kantaluokan operaatioita, joita ei löydy kantaluokan julkisesta rajapinnasta, protected-määre tarjoaa käytännöllisen ratkaisun ongelmaan.

Jotkut oppikirjat [Lippman ja Lajoie, 1997] ovat ottaneet sen kannan, että koska kantaluokan suunnittelija harvoin tietää tarkoin, millaisia luokkia kantaluokasta tullaan periyttämään, pannaan varmuuden vuoksi koko kantaluokan sisäinen toteutus protected-puolelle — jäsenmuuttujineen kaikkineen. Tällöin kantaluokan ja aliluokan välinen kapselointi kuitenkin murtuu ja aliluokka pääsee riippumaan kantaluokan sisäisestä toteutuksesta. Paljon parempi ratkaisu on kirjoittaa protected-puolelle sopivat apujäsenfunktiot, joiden avulla aliluokka pääsee sopivasti käsiksi kantaluokan sisimpään.

6.3.2 Periytyminen ja rakentajat

Kuten kaikki oliot, myös aliluokan oliot täytyy alustaa kun ne luodaan. Aliluvussa 3.4 käsiteltiin olion alustamista rakentajajäsenfunktion avulla. Periytyminen tuo kuitenkin omat lisänsä olioiden alustamiseen. Jokainen aliluokan olio koostuu kantaluokkaosasta tai -osista sekä aliluokan lisäämistä laajennuksista. Selvästi aliluokan täytyy määritellä oma rakentajansa, jotta aliluokan uudet osat saadaan alus-

tetuksi, mutta miten pitäisi toimia kantaluokan jäsenmuuttujien alustamisen kanssa? Aliluokan jäsenfunktiothan — rakentaja mukaanlukien — eivät pääse kantaluokan private-osiin käsiksi.

Ratkaisu alustusongelmaan löytyy jälleen luokkien vastuualueista. *Aliluokan vastuulla* on aliluokan mukanaan tuomien uusien jäsenmuuttujien ja muiden tietorakenteiden alustaminen. Tätä tarkoitusta varten aliluokkaan kirjoitetaan oma rakentaja tai rakentajat. *Kantaluokan vastuulla* on pitää huoli siitä, että aliluokan olion kantaluokkaosa tulee alustetuksi oikein, aivan kuin se olisi irrallinen kantaluokan olio. Tämän alustuksen hoitavat aivan normaalit kantaluokan rakentajat.

Ainoaksi ongelmaksi jäävät rakentajien parametrit. Aliluokan rakentaja saa kyllä parametrinsa aivan normaaliin tapaan aliluokan oliota luotaessa, mutta ongelmana on, miten kantaluokan rakentajalle saadaan välitetyksi sen tarvitsemat parametrit. C++:n tarjoama ratkaisu on, että aliluokan rakentajan *alustuslistassa* “kutsutaan” kantaluokan rakentajaa ja välitetään sille tarvittavat parametrit. Tällä tavoin aliluokka voi itse päättää, millaisia parametreja sen kantaluokalle välitetään olion luomisen yhteydessä. Listaus 6.2 seuraavalla sivulla näyttää esimerkin periytymisestä ja rakentajien käytöstä.

Jos aliluokan rakentajan alustuslistassa *ei* kutsuta mitään kantaluokan rakentajaa, yrittää kääntäjä olla ystävällinen. Tällaisessa tilanteessa se kutsuu nimittäin kantaluokan *oletusrakentajaa,* joka ei tarvitse parametreja. Tämä aiheuttaa kuitenkin sen, että rakentajakutsun unohtuessa alustuslistasta olion kantaluokkaosa alustetaan oletusarvoonsa, joka tuskin on haluttu lopputulos! Onkin erittäin tärkeää, että aliluokan rakentajan alustuslistassa kutsutaan ***aina*** jotain kantaluokan rakentajaa.

Aliluokan olion alustusjärjestys C++:ssa on sellainen, että ensimmäisenä suoritetaan kantaluokan rakentaja kokonaisuudessaan, jonka jälkeen suoritetaan aliluokan oma rakentaja. Mikäli periytymishierarkia on korkeampi kuin kaksi luokkaa, lähdetään rakentajia suorittamaan aivan hierarkian huipusta lähtien alaspäin, niin että olio ikään kuin rakentuu vähitellen laajemmaksi ja laajemmaksi. Tämä alustusjärjestys takaa sen, että aliluokan rakentajassa voidaan jo turvallisesti kutsua kantaluokan jäsenfunktioita, koska aliluokan rakentajan koodiin päästäessä kantaluokkaosa on jo alustettu kuntoon (poikkeuksena ovat aliluokan uudelleenmäärittelemät *virtuaalifunktiot*, joiden käyttäytymistä rakentajien kanssa käsitellään tarkemmin aliluvus-

```
................................ Kantaluokka ................................
 1 class Lokiviesti
 2 {
 3 public:
 4   Lokiviesti(string const& viesti);
       ⋮
 5 private:
 6   string viesti_;
 7 };
 8
 9 Lokiviesti::Lokiviesti(string const& viesti) : viesti_(viesti)
10 {
11 }
................................ Aliluokka ................................
 1 class PaivattyLokiviesti : public Lokiviesti
 2 {
 3 public:
 4   PaivattyLokiviesti(Paivays const& pvm, string const& viesti);
       ⋮
 5 private:
 6   Paivays pvm_;
 7 };
 8
 9 PaivattyLokiviesti::PaivattyLokiviesti(Paivays const& pvm,
10   string const& viesti) : Lokiviesti(viesti), pvm_(pvm)
11 {
12 }
................................ Olion luominen ................................
 1   Lokiviesti viesti("Kävin leffassa");
 2   PaivattyLokiviesti pvmviesti(tanaan(), "Huono oli");
```

LISTAUS 6.2: Periytyminen ja rakentajat

sa 6.5.7).

6.3.3 Periytyminen ja purkajat

Alustamisen tapaan myös aliluokan olion siivoustoimenpiteet vaativat erikoiskohtelua luokan "kerrosrakenteen" vuoksi. Samoin kuin aliluokan olion alustaminen, myös sen siivoaminen on jaettu vastuualueiden kesken. Kantaluokan purkajan tehtävänä on siivota kantaluokkaolio sellaiseen kuntoon, että se voi rauhassa tuhoutua. Aliluokan purkajan vastuulla on vastaavasti pitää huoli siitä, että aliluokan

olion *periytymisessä lisätty laajennusosa* siivotaan tuhoutumiskuntoon.

Kuten aiemminkin, ohjelmoijan ei itse tarvitse huolehtia purkajien kutsumisesta vaan olion tuhoutuessa kääntäjä kutsuu automaattisesti tarvittavia purkajia. Periytymisen yhteydessä olion purkajien kutsujärjestys on päinvastainen rakentajien kutsujärjestykseen nähden eli ensin kutsutaan aliluokan purkajaa, sitten kantaluokan purkajaa ja tarvittaessa tämän kantaluokan purkajaa ja niin edelleen. Tämä kutsujärjestys varmistaa sen, että aliluokan purkajaa suoritettaessa olion kantaluokkaosa on vielä käyttökelpoinen, ja näin aliluokan purkaja voi vielä turvallisesti kutsua kantaluokan jäsenfunktioita (kuten rakentajissakin, aliluokan uudelleenmäärittelemät virtuaalifunktiot ovat poikkeustapaus. Niitä ja purkajia käsitellään tarkemmin aliluvussa 6.5.7).

Kantaluokan purkaja kannattaa määritellä lähes aina *virtuaaliseksi*, mutta tätä käsitellään vasta aliluvussa 6.5.5.

6.3.4 Aliluokan olion ja kantaluokan suhde

Koska aliluokka perii kantaluokan kaikki ominaisuudet, tarjoaa aliluokan olio ulospäin kaikki ne palvelut, jotka kantaluokan oliokin tarjoaa. Näin ollen aliluokan oliota voisi sen ulkoista rajapintaa ajatellen käyttää kaikkialla, missä kantaluokan oliotakin — periytymisessähän vain lisätään ominaisuuksia.

Tämä ajatus aliluokan olion kelpaamisesta kantaluokan olion paikalle on viety useissa oliokielissä vielä pitemmälle määrittelemällä, että kielen tyypityksen kannalta ***aliluokan olio on tyypiltään myös kantaluokan olio***. Näin aliluokan oliot kuuluvat ikään kuin useaan luokkaan: aliluokkaan itseensä, kantaluokkaan, mahdollisesti kantaluokan kantaluokkaan jne. Tämä *is-a* -suhde tulisi pitää mielessä aina, kun periytymistä käytetään. Jos aliluokka on muuttunut vastuualueeltaan niin paljon, että se ei enää ole kantaluokan mukainen, periytymistä on käytetty mitä ilmeisimmin väärin.

C++:n kannalta tämä ominaisuus tarkoittaa, että kielen tyypityksessä aliluokan olio kelpaa kaikkialle minne kantaluokan oliokin. Erityisesti kantaluokkaosoittimen tai -viitteen voi laittaa osoittamaan myös

aliluokan olioon:

```
class Kantaluokka { ... };
class Aliluokka : public Kantaluokka { ... };
void funktio(Kantaluokka& kantaolio);

Kantaluokka* k_p = 0;
Aliluokka aliolio;
k_p = &aliolio;
funktio(aliolio);
```

Tämä tilanne, jossa osoittimen päässä olevan olion tyyppi ei ole sama kuin osoittimen tyyppi, ei aiheuta yleensä ongelmia, koska jokainen aliluokan olio tarjoaa periytymisestä johtuen kaikki kantaluokan tarjoamat palvelut. Tämä mahdollisuus käyttää aliluokan olioita ohjelmassa kantaluokan sijaan on yksi tärkeimpiä olio-ohjelmoinnin ja periytymisen työkaluja.[ϒ] Sen käyttöä käsitellään aliluvussa 6.5.2.

6.4 C++: Periytymisen käyttö laajentamiseen

Periytymisen kenties yksinkertaisin käyttötarkoitus on olemassa olevan luokan laajentaminen. Siinä aliluokka laajentaa kantaluokan tarjoamia palveluita tarjoamalla kaikki kantaluokan tarjoamat palvelut identtisinä ja sen lisäksi vielä uusia palveluita. Tällainen periytymisen käyttö mahdollistaa koodin uudelleenkäytön, koska aliluokan ei tarvitse kirjoittaa uudelleen kantaluokan jo kertaalleen toteuttamia palveluita.

Listaus 6.3 seuraavalla sivulla näyttää yksinkertaisen luokan `Kirja`, joka muistaa yksittäisen kirjan tiedot. Periaatteessa listauksessa esitetty luokka sopisi muuten hyvin kirjaston kortistojärjestelmässä käytetyksi kirjaksi, mutta kirjaston kirjoilla on yksi olennainen lisäominaisuus: niillä on viimeinen palautuspäivämäärä. (Listauksen rivillä 5 on purkajan edessä avainsana **`virtual`**. Sillä ei ole tämän esimerkin kannalta merkitystä, mutta näin tehdään yleensä kaikissa kantaluokissa. Virtuaalipurkajan merkitys selitetään aliluvussa 6.5.5.)

[ϒ] C++:ssa vaaraksi muodostuu **viipaloituminen** (*slicing*). Kun esimerkiksi kantaluokan olioon sijoitetaan aliluokan olio (joka on tyypiltään myös kantaluokan olio), suoritetaan sijoituksessa vain kantaluokkaosan sijoitus ja aliluokkaosa jää sijoituksessa käyttämättä. Tätä käsitellään aliluvussa 7.1.3.

```
................................ Luokan esittely ................................
 1 class Kirja
 2 {
 3 public:
 4   Kirja(std::string const& nimi, std::string const& tekija);
 5   virtual ~Kirja();
 6   std::string annaNimi() const;
 7   std::string annaTekija() const;
          ⋮
11 private:
          ⋮
13   std::string nimi_;
14   std::string tekija_;
15 };
............................... Luokan toteutus ...............................
 1 Kirja::Kirja(string const& n, string const& t) : nimi_(n), tekija_(t)
 2 {
 3   cout << "Kirja " << nimi_ << " luotu" << endl;
 4 }
 5
 6 Kirja::~Kirja()
 7 {
 8   cout << "Kirja " << nimi_ << " tuhottu" << endl;
 9 }
10
11 string Kirja::annaNimi() const
12 {
13   return nimi_;
14 }
          ⋮
```

LISTAUS 6.3: Kirjan tiedot muistava luokka

Jos olemassa olevaa kirjaluokkaa halutaan käyttää uuden luokan pohjana periytymisessä, on tärkeää ensin varmistua siitä, että uusi luokka tarjoaa sellaisenaan kaikki vanhan luokan palvelut ja lisää siihen lisäksi uusia palveluita. Kirjan ja kirjaston kirjan tapauksessa nämä edellytykset toteutuvat, koska kaikki kirjan tarjoamat palvelut sopivat sellaisenaan myös kirjaston kirjalle. Listauksessa 6.4 on luokasta `Kirja` periytetty uusi laajempi luokka `KirjastonKirja`, joka tarjoaa kaikki kirjan palvelut ja lisäksi palautuspäivämäärän käsittelyn.

Periyttämistä ei koskaan kannata käyttää turhaan, koska se monimutkaistaa ohjelmaa. Esimerkiksi yllä oleva kirjastonkirjan periyttäminen kirjasta on perustelua vain, jos jossain todella tarvitaan myös tavallista kirjaluokkaa tai jos kirjaluokka on saatu muualta. Mikäli

Luokan esittely

```
1  class KirjastonKirja : public Kirja
2  {
3  public:
4    KirjastonKirja(std::string const& nimi, std::string const& tekija,
5                   Paivays const& palpvm);
6    virtual ~KirjastonKirja();
7    bool onkoMyohassa(Paivays const& tanaan) const;
       ⋮
9  private:
10   Paivays palpvm_;
11 };
```

Luokan toteutus

```
1  KirjastonKirja::KirjastonKirja(string const& nimi, string const& tekija,
2    Paivays const& palpvm) : Kirja(nimi, tekija), palpvm_(palpvm)
3  {
4    cout << "Kirjastonkirja " << nimi << " luotu" << endl;
5  }
6
7  KirjastonKirja::~KirjastonKirja()
8  {
9    cout << "Kirjastonkirja " << annaNimi() << " tuhottu" << endl;
10 }
11
12 bool KirjastonKirja::onkoMyohassa(Paivays const& tanaan) const
13 {
14   return palpvm_.paljonkoEdella(tanaan) < 0;
15 }
```

LISTAUS 6.4: Kirjaston kirjan palvelut tarjoava aliluokka

aalisen moniperiytymisen käyttöä ja tekee luokkien välisen vastuunjaon 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.

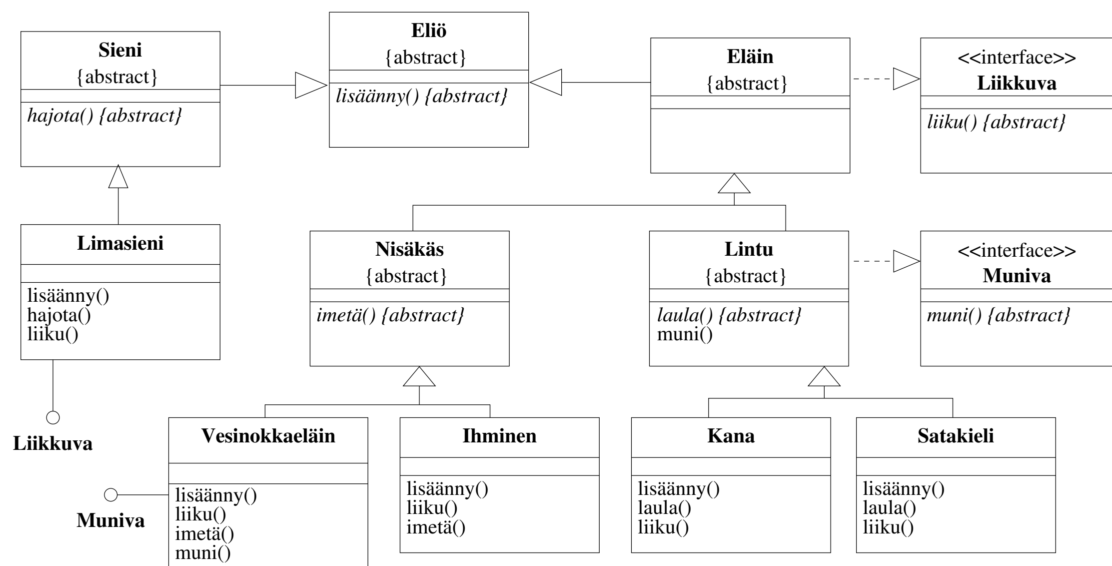
[*] Tarkasti ottaen C++-standardi joutuu ottamaan kantaa virtuaaliseen periytymiseen purkajienkin yhteydessä, kun se määrittelee purkajien keskinäisen kutsujärjestyksen. Tällä ei kuitenkaan normaalisti ole mitään merkitystä, ja purkajia edelleen kutsutaan käänteisessä järjestyksessä rakentajiin verrattuna.

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 käännösaikaista tyypitystä, 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. Listaus 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 }
.............................. 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 }
```

LISTAUS 6.15: Erilliset rajapinnat Javassa

6.9.2 C++: Rajapintaluokat ja moniperiytyminen

C++:ssa rajapintaluokkia mallinnetaan abstrakteilla kantaluokilla ja moniperiytymisellä. 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 moniperiytymisellä C++:ssa

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

Rajapintaluokkien toteuttaminen C++:ssa moniperiytymisen avulla aiheuttaa kuitenkin jonkin verran kömpelyyttä ohjelmaan. Ensinnäkin, koska sekä normaali periytyminen että rajapinnan toteuttaminen tehdään kielessä periytymissyntaksilla, ei luokan esittelystä suoraan näe, mitkä sen kantaluokista ovat puhtaita rajapintoja ja mitkä sisältävät myös toteutusta. Tähän ei C++:ssa ole muuta ratkaisukeinoa kuin nimetä rajapintaluokat niin, että käy selvästi ilmi niiden olevan pelkkiä rajapintoja.

Rajapintaluokat, rakentajat ja virtuaalipurkaja

Rajapintaluokat ovat C++:ssa normaaleja luokkia, joten niilläkin on rakentaja, jota kutsutaan aliluokan rakentajasta. Käytännössä tämä rakentaja on kuitenkin aina tyhjä, koska rajapintaluokat nimensä mukaan määrittävät vain rajapinnan eivätkä sisällä jäsenmuuttujia tai toiminnallisuutta. Tämän vuoksi rakentajien kirjoittaminen rajapintaluokille olisi turhauttavaa. Tässä onneksi C++:n normaalisti vaarallinen automatiikka auttaa.

Aliluvussa 3.4.1 todettiin, että jos luokalle ei kirjoiteta yhtään rakentajaa, tekee kääntäjä sinne automaattisesti “tyhjän” oletusrakentajan. Vastaavasti aliluvussa 6.3.2 kävi ilmi, että jos kantaluokan rakentajan kutsu jätetään pois aliluokan rakentajan alustuslistasta, kutsuu kääntäjä automaattisesti kantaluokan oletusrakentajaa. Näiden ominaisuuksien ansiosta rajapintaluokkaan ei tarvitse kirjoittaa rakentajaa ollenkaan, koska tällöin kääntäjä automaattisesti luo tyhjän oletusrakentajan 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 (aliluku 6.5.5), jotta niistä periytettyjen luokkien oliot tuhottaisiin aina oikein. Rajapintaluokat ovat kantaluokkia, joten tämä sääntö koskee 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 luokkaesittelyn *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 esittelyn 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