# Periytyvyys ja oliosuunnittelu

Monet ihmiset ovat sitä mieltä, että periytyvyys on yhtä kuin oliopohjainen ohjelmointi. Siitä riittää keskusteltavaa, onko asia näin, mutta tämän kirjan muissa osissa olevien Kohtien määrän pitäisi vakuuttaa sinut siitä, että kun on kysymys tehokkaasta C++-ohjelmoinnista, käytössäsi on paljon enemmän työkaluja kuin jos yksinkertaisesti määrität, mitkä luokat periytyvät mistäkin luokista.

Luokkahierarkioiden suunnittelu ja toteuttaminen on silti pohjimmiltaan erilaista kuin mikään muu C-kielen maailmassa. Periytyvyyden ja oliosuuntautuneisuuden suunnittelu on varmasti se alue, joka saa sinut radikaalisti ajattelemaan uudelleen asennoitumistasi ohjelmistojärjestelmien rakentamiseen. C++-kieli tarjoaa lisäksi häkellyttävän valikoiman olio-ohjelmoinnin rakennuspalikoita, mukaan lukien julkiset, suojatut ja yksityiset kantaluokat; virtuaaliset ja epävirtuaaliset kantaluokat; ja virtuaaliset ja epävirtuaaliset jäsenfunktiot. Jokainen näistä ei vaikuta pelkästään toisiinsa, vaan myös kielen muihin komponentteihin. Tästä on tuloksena, että kun yrität ymmärtää, mitä kukin ominaisuus tarkoittaa, koska sitä pitäisi käyttää, ja kuinka se yhdistetään parhaiten C++-kielen oliosuuntaumattomiin aspekteihin, tuloksena voi olla pelottava pyrkimys.

Asiaa mutkistaa lisäksi se, että kielen eri piirteet tuntuvat tekevän enemmän tai vähemmän samaa asiaa. Esimerkkejä:

- Tarvitset luokkakokoelman, jolla on monia jaettuja tuntomerkkejä. Pitäisikö sinun käyttää periytyvyyttä ja pitäisikö kaikkien luokkien periytyä yleisestä kantaluokasta, pitäisikö sinun käyttää malleja ja pitäisikö niiden kaikkien tulla luoduksi yhteisestä koodirungosta?
- Luokka A toteutetaan luokan B ehdoilla. Pitäisikö A:lla olla tietojäsen, joka on tyyppiä B, vai pitäisikö A:n periytyä yksityisesti B:stä?

- Sinun kannattaa suunnitella tyyppiturvallinen homogeeninen säiliöluokka, eli sellainen luokka, jota ei ole normaalissa kirjastossa. (Katso Kohdasta 49 luettelo säiliöistä, jotka kirjasto *tarjoaa*.) Pitäisikö sinun käyttää malleja, vai olisiko parempi rakentaa tyyppiturvallisia rajapintoja sen luokan ympärille, joka itse on toteutettu käyttämällä geneerisiä (`void*`) osoittimia?

Tarjoan tämän osan Kohdissa opastusta siihen, kuinka vastataan tämänkaltaisiin kysymyksiin. En voi kuitenkaan ottaa kantaa oliopohjaisen suunnittelun jokaiseen näkökulmaan. Keskityn sen sijaan selittämään sitä, mitä C++-kielen eri piirteet todella *tarkoittavat*, sitä mitä sinä todella *sanot*, kun käytät tiettyä piirrettä. Esimerkiksi julkinen periytyminen tarkoittaa samaa kuin *on eräänlainen* (katso Kohta 35), ja jos yrität saada sen tarkoittamaan mitään muuta, joudut hankaluuksiin. Vastaavasti virtuaalifunktio tarkoittaa "rajapinnan täytyy periytyä", siinä missä epävirtuaalifunktio tarkoittaa "sekä rajapinnan että toteutuksen täytyy periytyä". Monille C++-ohjelmoijille on aiheutunut sanomatonta surua, kun he eivät ole onnistuneet tekemään mitään eroa näiden välille.

Jos ymmärrät C++-kielen eri piirteiden merkitykset, huomaat, että ajattelutapasi oliosuunnittelusta vaihtelee. Sen sijaan, että se olisi harjoitus, jossa eriytetään kielen rakenteet, siitä tulee sopivasti juttu, jossa yritetään hahmottaa se, mitä yrität sanoa ohjelmistojärjestelmästäsi. Sitten kun tiedät mitä kertoa, kykenet kääntämään sen C++-kielen vastaaviksi piirteiksi ilman suurempia hankaluuksia.

Sen tärkeyttä, että sanot mitä tarkoitat ja ymmärrät mitä sanot, ei voi korostaa liikaa. Seuraavat kohdat sisältävät yksityiskohtaista tutkimusta siitä, kuinka tämä tehdään tehokkaasti. Kohdassa 44 luodaan yhteenveto kirjeenvaihdosta C++-kielen oliosuuntautuneiden rakenteiden välillä ja siitä, mitä ne tarkoittavat. Se toimii mukavana johdatuksena tähän osaan, samoin kuin suppeana viittauksena tulevaisuuden konsultaatioon.

## Kohta 35: Varmista, että julkinen periytyvyys on malliltaan *on eräänlainen*.

Kirjassaan *Some Must Watch While Some Must Sleep* (W. H. Freeman and Company, 1974), William Dement viittaa tarinaansa yrityksestä saada oppilaidensa päähän kurssin tärkeimmät harjoitukset. Väitetään hänen kertoneen luokalleen, että keskiverto brittiläinen koululainen ei muista paljon muuta historiasta, kuin että Hastingsin taistelu oli vuonna 1066. Dement korosti, että jos lapsi muistaa jotain muuta, hän muistaa vuoden 1066. Dement jankkasi *kurssinsa* oppilaille, että on olemassa vain muutama keskeinen viesti, mukaanlukien - kiinnostavaa kyllä - tosiasia, että unilääkkeet aiheuttavat unettomuutta. Hän pyysi hartaasti oppilaitaan muistamaan nämä muutamat kriittiset tosiasiat, vaikka he unohtaisivat kaiken muun kurssilla käsitellyn, ja hän palasi näihin perusohjeisiin toistuvasti lukukauden aikana.

Kurssin lopussa olevan viimeisen tentin viimeinen kysymys oli "Kirjoita yksi kurssin asia, jonka tulet varmasti muistamaan koko loppuelämäsi". Kun Dement tarkisti tentit, hän oli ällistynyt. Melkein kaikki olivat kirjoittaneet "1066".

Julistan teille nyt tämän takia vapisten tärkeimmän yksittäisen säännön oliopohjaisessa C++-ohjelmoinnissa: julkinen periytyvyys tarkoittaa samaa kuin *on eräänlainen*. Paina tämä sääntö muistiin.

Jos kirjoitat, että luokka D ("Derived") periytyy julkisesti luokasta B ("Base"), kerrot C++-ohjelmoijille (samoin kuin koodiasi lukeville ihmisille), että tyypin D jokainen olio on myös olio tyypiltään B, mutta ei *päinvastoin*. Kerrot, että B esittää yleisemmän käsitteen kuin D, mutta D esittää uudelleen erikoistuneemman käsitteen kuin B. Vakuutat, että B-tyyppistä oliota voidaan käyttää missä tahansa samoin kuin tyyppiä D, koska jokainen olio, joka on tyyppiä D, on tyyppiä B oleva olio. Toisaalta, jos tarvitset olion, joka on tyyppiä D, olio, joka on tyyppiä B, ei käy: jokainen D *on eräänlainen* B, mutta ei päinvastoin.

C++-kieli pakottaa tämän tulkinnan julkisesta periytymisestä. Tutki tätä esimerkkiä:

```
class Person { ... };
class Student: public Person { ... };
```

Tiedämme jokapäiväisestä kokemuksesta, että jokainen opiskelija on henkilö, mutta jokainen henkilö ei ole opiskelija. Tämä on juuri sitä, mitä tämä hierarkia vakuuttaa. Odotamme, että kaikki mikä on totta henkilöstä - esimerkiksi se, että hänellä on syntymäaika - on myös totta opiskelijasta, mutta emme odota, että kaikki, mikä on totta opiskelijasta - että hän on esimerkiksi kirjautuneena johonkin kouluun - on totta ihmisistä yleensä. Henkilön käsite on yleisempi kuin opiskelijan käsite, opiskelija on erikoistuneen tyyppinen henkilö.

C++-kielessä mikä tahansa argumenttia odottava funktio, joka on tyyppiä `Person` (tai osoitin `Person`-luokkaan tai viittaus `Person`-luokkaan) ottaa sen sijaan `Student`-olion (tai osoittimen `Student`-olioon tai viittauksen `Student`-olioon):

```
void dance(const Person& p);   // jokainen osaa tanssia
void study(const Student& s);  // vain opisk. opiskelevat
Person p;                      // p on Person
Student s;                     // s Student
dance(p);                      // toimii, p on Person
```

```
dance(s);                       // toimii, s on Student,
                                // ja Student "on" Person
study(s);                       // toimii
study(p);                       // virhe! p ei ole Student
```

Tämä on totta vain *julkiselle* periytyvyydelle. C++-kieli käyttäytyy kuvaamallani tavalla vain, jos `Student` on julkisesti periytetty `Person`-luokasta. Yksityinen periytyvyys tarkoittaa jotain täysin erilaista (katso Kohta 42), ja kukaan ei tunnu tietävän, mitä suojatun periytymisen oletetaan tarkoittavan.

Julkisen periytyvyyden ja *on eräänlainen* -lauseen samanarvoisuus kuulostaa yksinkertaiselta, mutta asia ei käytännössä ole aina niin suoraviivainen. Näkemyksesi voi joskus johtaa sinut harhaan. Esimerkiksi on tosiasia, että pingviini on lintu ja on tosiasia, että linnut osaavat lentää. Jos yritämme naiivisti ilmaista tämän asian C++-kielellä, yrityksemme tuottaa:

```
class Bird {
public:
  virtual void fly();           // linnut osaavat lentää
  ...
};
class Penguin: public Bird {    // pingviinit ovat lintuja
  ...
};
```

Olemme yhtäkkiä vaikeuksissa, koska hierarkia sanoo, että pingviinit osaavat lentää. Tiedämme, että tämä ei ole totta. Mitä tapahtui??

Olemme tässä tapauksessa epävarsinaisen kielen (suomen kieli) uhreja. Kun sanomme, että linnut osaavat lentää, emme todella tarkoita, että kaikki linnut osaavat lentää, vaan yleisesti ottaen linnuilla on kyky lentää. Jos olemme vielä tarkempia, tunnistamme, että itse asiassa on olemassa usean tyyppisiä lintuja, jotka eivät lennä, ja meille voisi tulla mieleen seuraava hierarkia, joka mallintaa todellisuutta paljon paremmin:

```
class Bird {
  ...                           // mitään fly-funktiota
};                              // ei ole esitelty
class FlyingBird: public Bird {
public:
  virtual void fly();
  ...
};
```

```
class NonFlyingBird: public Bird {
  ...                          // mitään fly-funktiota
                               // ei ole esitelty
};
class Penguin: public NonFlyingBird {
  ...                          // mitään fly-funktiota
                               // ei ole esitelty
};
```

Tämä hierarkia on paljon uskollisempi sille, jonka todella tunnemme, kuin alkuperäinen suunnitelma.

Emme vieläkään ole käsitelleet ihan loppuun asti näitä lintutarha-asioita, koska eräille ohjelmistojärjestelmille voi olla täysin asianmukaista sanoa, että pingviini *on eräänlainen* lintu. Varsinkin, jos ohjelmallasi on paljon tekemistä nokkien ja siipien kanssa ja sillä ei ole mitään tekemistä lentämisen kanssa, alkuperäinen hierarkia voisi toimia aivan hienosti. Vaikka tämä voi tuntua ärsyttävältä, se on yksinkertainen heijastus siitä, että kaikille ohjelmille ei ole yhtä ihanteellista kaavaa. Paras työtapa riippuu siitä, mitä järjestelmän odotetaan tekevän, sekä nyt että tulevaisuudessa. Jos sovelluksellasi ei ole tietämystä lentämisestä ja sillä ei koskaan odoteta olevankaan, `Penguin`-luokasta tekeminen `Bird`-luokasta periytetty luokka voi olla täysin kelpo työtapapäätös. Se voi itse asiassa olla suositeltavampaa päätöksessä, jossa tehdään ero lentävien ja lentämättömien lintujen välillä, koska tällainen erityispiirre voi puuttua maailmasta, jota yrität mallintaa. Haitallisten luokkien lisääminen hierarkiaan voi olla aivan yhtä huono työtapapäätös kuin se, että luokkien välillä on väärä periytymissuhde.

On olemassa toinen koulukunta, jonka ajatuksia siitä, kuinka hoidetaan se, mitä minä kutsun "kaikki linnut osaavat lentää, pingviinit ovat lintuja, pingviinit eivät osaa lentää, huh huh"-ongelmaan. Se on määrittää pingviinien `fly`-funktio uudelleen niin, että se saa aikaan suoritusaikaisen virheen:

```
void error(const string& msg); // määritetty muualla

class Penguin: public Bird {
public:
  virtual void fly() { error("Pingv. eivät osaa lentää!");
}

  ...
};
```

SmallTalkin tapaisilla tulkituilla kielillä on taipumus omaksua tämä työtapa, mutta on tärkeä tunnistaa se, että tämä työtapa ajattelee aivan eri tavalla kuin sinä. Tämä ei sano, että "Pingviinit eivät osaa lentää." Tämä kertoo, että "Pingviinit osaavat lentää, on virhe, jos ne yrittävät sitä."

Kuinka erotat asiat? Sillä hetkellä, kun virhe saadaan selville. Kielto "Pingviinit eivät osaa lentää" voidaan pakottaa kääntäjillä, mutta lauseen väärinkäytökset "On virheellistä, jos pingviinit yrittävät lentää", huomataan vain suorituksen aikana.

Kun haluat ilmaista rajoituksen "Pingviinit eivät osaa lentää", varmista, että `Penguin`-olioille ei ole määritetty tällaista funktiota:

```
class Bird {
  ...                          // mitään fly-funktiota
                               // ei ole esitelty
};
class NonFlyingBird: public Bird {
  ...                          // mitään fly-funktiota ei
                               // ei ole esitelty
};
class Penguin: public NonFlyingBird {
  ...                          // mitään fly-funktiota ei
                               // ei ole esitelty
};
```

Jos yrität saada pingviinin lentämään, kääntäjät nuhtelevat sinua lain rikkomisesta:

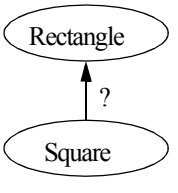
```
Penguin p;
p.fly();                       // virhe!
```

Tämä on hyvin erilaista käyttäytymistä kuin Smalltalk-kielen työtapa. Kääntäjät eivät sano tällä metodiikalla sanaakaan.

C++-kielen filosofia on pohjimmiltaan erilainen kuin Smalltalk-kielen filosofia, joten sinun kannattaa tehdä asiat C++-kielen tavalla niin kauan kuin ohjelmoit C++-kielellä. Lisäksi on tiettyjä teknisiä etuja siitä, että virheet huomataan suorituksen sijasta kääntämisen aikana - katso Kohta 46.

Hyväksyt ehkä sen, että ornitologinen näkemyksesi voi olla puutteellinen, mutta voit luottaa geometrian alkeiden hallintaasi, eikö niin? Tarkoitan sitä, että kuinka monimutkaisia suorakulmiot ja neliöt voivat olla?

Vastaa tähän yksinkertaiseen kysymykseen: pitäisikö `Square`-luokan periytyä julkisesti `Rectangle`-luokasta?



"Pöh!" sanot, "Tottakai sen pitäisi! Jokainen tietää, että suorakulmio on neliö, mutta ei yleisesti ottaen päinvastoin." Tämä on totta, mutta ei lukiossa. Luulenpa, että emme enää ole lukiossa.

Tutki tätä koodia:

```
class Rectangle {
public:
  virtual void setHeight(int newHeight);
  virtual void setWidth(int newWidth);

  virtual int height() const;       // palauta nykyiset
  virtual int width() const;        // arvot

  ...

};

void makeBigger(Rectangle& r)       // funktio, jolla li-
{                                   // sätään r:n aluetta
  int oldHeight = r.height();

  r.setWidth(r.width() + 10);       // lisää 10 r:n leveys

  assert(r.height() == oldHeight);  // vakuuta, että r:n
}                                   // kork. ei ole muutt.
```

Vakuuttelun ei selvästi pitäisi koskaan epäonnistua. `makeBigger`-funktio muuttaa vain `r`:n leveyden. Sen korkeutta ei koskaan muuteta.

Tutki seuraavaksi tätä koodia, joka käyttää julkista periytyvyyttä, jonka avulla neliöitä kohdellaan kuten suorakulmioita:

```
class Square: public Rectangle { ... };

Square s;

...

assert(s.width() == s.height());    // tämän täytyy olla
                                    // totta kaik. nel.
makeBigger(s);                      // periytyvyyden avul-
                                    // la s "on eräänlai-
                                    // nen" Rectangle,
                                    // joten voimme lisä-
                                    // tä sen aluetta

assert(s.width() == s.height());    // tämän täytyy edell.
                                    // olla totta kaikil-
                                    // le neliöille
```

Tässä kohtaa on yhtä selvää kuin edellä, että viimeisen vakuuttelun ei myöskään pitäisi koskaan epäonnistua. Määritys kuuluu, että neliön leveys on sama kuin sen korkeus.

Mutta meillä on nyt ongelma. Kuinka voimme sovittaa seuraavat vakuuttelut?

- Ennen kuin kutsumme `makeBigger`-funktiota, `s`:n korkeus on sama kuin sen leveys;
- s:n leveys on muuttunut `makeBigger`-funktion sisällä, mutta sen korkeus ei ole muuttunut:
- Kun `s` palautuu `makeBigger`-funktiosta, sen korkeus on jälleen sama kuin sen leveys. (Huomaa, että `s` välitetään `makeBigger`-funktiolle viittausparametrillä, joten `makeBigger` muuttaa itse `s`:n , ei `s`:n kopioita.)

No?

Tervetuloa julkisen periytyvyyden ihanaan maailmaan eli sinne missä vaistot, jotka olet kehittänyt tutkimuksen muilla kentillä - mukaan lukien matematiikka - eivät palvele sinua aivan niin hyvin kuin olisit odottanut. Tässä tapauksessa on se perustavaa laatua oleva vaikeus, että jokin suorakulmioon sovellettavissa oleva (sen leveys voidaan muuttaa riippumatta sen korkeudesta) ei ole sovellettavissa neliöön (sen leveys ja korkeus on pakotettu olemaan samat). Mutta julkinen periytyvyys vakuuttaa, että kaikki, joka on sovellettavissa kantaluokan olioihin - *kaikki!* - on myös sovellettavissa periytyneisiin luokkaolioihin. Suorakulmion ja neliön tapauksessa (ja vastaavassa esimerkissä, joka liittyy Kohdassa 40 sarjoihin ja listoihin), tästä vakuutuksesta ei voida pitää kiinni, joten julkisen periytyvyyden käyttäminen mallinnettaessa niiden suhdetta on aivan väärin. Kääntäjät antavat tietenkin tehdä näin, mutta kuten olemme nähneet, se ei ole mikään takuu siitä, että koodi tulee käyttäytymään kunnolla. Jokaisen ohjelmoijan täytyy oppia (eräiden hieman useammin kuin muiden), että vaikka koodi kääntyy, se ei välttämättä toimi.

Ei kannata harmitella sitä, että vuosien varrella kehittämäsi ohjelmistointuitio pettää, kun lähestyt oliopohjaista työtapaa. Tuo tieto on edelleen arvokasta, mutta nyt kun olet lisännyt periytyvyyden työtapavaihtoehtojen arsenaaliisi, sinun täytyy kasvattaa intuitiotasi uusilla oivalluksilla, jotka opastavat sinua käyttämään periytyvyyttä asianmukaisesti. Ajan myötä käsite, että `Penguin` perii `Bird`-luokalta tai `Square` perii `Rectangle`-luokalta, antaa sinulle saman hassun tunteen kuin minkä saat, kun joku näyttää sinulle funktion, joka on monta sivua pitkä. On *mahdollista*, että se on oikea tapa lähestyä asioita, mutta se ei ole kovin todennäköistä.

*On eräänlainen* -suhde ei tietenkään ole ainoa, joka voi olla olemassa luokkien välillä. Kaksi muuta yleistä luokkien sisäistä suhdetta ovat *omistaa* ja on *toteutettu jonkin ehdoilla*. Nämä suhteet käsitellään Kohdissa 40 ja 42. C++-kielen työtavoille on yleistä, että ne kasvavat kieroon, koska toinen näistä tärkeistä suhteista oli väärin mallinnettu *on eräänlainen* -tyyppisenä, joten sinun kannattaa varmistaa, että ymmärrät eron näiden suhteiden välillä olena ja tiedät, kuinka ne mallinnetaan parhaiten C++-kielessä.

## Kohta 36: Erota rajapinnan ja toteutuksen periytyvyys.

Periytyvyyden (julkisen) näennäisesti suoraviivainen käsite osoittautuu tarkastuksessa koostuvan kahdesta erotettavissa olevasta osasta: funktion rajapintojen periytyvyydestä ja funktion toteutuksien periytyvyydestä. Näiden kahdentyyppisen periytyvyyden ero vastaa tarkasti tämän kirjan esittelyssä käsiteltyjen funktioiden esittelyiden ja funktioiden määritysten eroa.

Haluat luokkien suunnittelijana periytettyjen luokkien joskus periytyvän vain jäsenfunktion rajapinnasta (esittely); joskus haluat periytettyjen luokkien periytyvän sekä funktion rajapinnasta että toteutuksesta, mutta haluat niiden pystyvän ohittamaan tarjoamasi toteutuksen; ja haluat niiden periytyvän joskus sekä rajapinnasta että toteutuksesta ilman, että sallit niiden ohittaa mitään.

Jotta saisit paremman tuntuman näiden optioiden välisestä erosta, tutki luokkahierarkiaa, joka esittää geometrisiä muotoja grafiikkaohjelmassa:

```
class Shape {
public:
  virtual void draw() const = 0;

  virtual void error(const string& msg);

  int objectID() const;

  ...
};

class Rectangle: public Shape { ... };

class Ellipse: public Shape { ... };
```

`Shape` on abstrakti luokka; sen puhdas virtuaalifunktio `draw` merkitsee sen sellaisena. Tästä on tuloksena, että asiakkaat eivät voi luoda ilmentymiä `Shape`-luokasta, vaan siitä periytetyistä luokista. `Shape` rynnistää joka tapauksessa vahvan vaikutuksen kaikkiin niihin luokkiin, jotka (julkisesti) periytyvät siitä, koska:

- Jäsenfunktioiden *rajapinnat ovat aina periytyneitä*. Kuten selvitettiin Kohdassa 35, julkinen periytyvyys tarkoittaa samaa kuin *on eräänlainen*, joten kaiken, mikä on totta kantaluokassa, täytyy myös olla totta siitä periytetyissä luokissa. Tästä johtuu, että jos funktio pätee luokkaan, sen täytyy myös päteä sen aliluokkiin.

`Shape`-luokassa esitellään kolme funktiota. Ensimmäinen, `draw`, piirtää senhetkisen olion implisiittisellä näytöllä. Jäsenfunktiot kutsuvat toista, `error`, jos niiden täytyy raportoida virheestä. Kolmas, `objectID`, palauttaa senhetkisille olioille yksilöllisen kokonaislukutunnisteen; Kohdassa 17 nähdään esimerkki siitä, kuinka tällaista funktiota voitaisiin käyttää. Jokainen funktio esitellään eri tavalla: `draw` on puhdas virtuaalifunktio, `error` on yksinkertainen (epäpuhdas?) virtuaalinen funktio

ja `objectID` on ei-virtuaalifunktio. Miten nämä erilaiset esittelyt kietoutuvat toisiinsa?

Tutki ensin puhdasta virtuaalifunktiota `draw`. Kaksi kaikkein silmäänpistävintä piirrettä puhtaasti virtuaalisissa funktioissa on, että minkä tahansa todellisen luokan, joka perii ne, *täytyy* esitellä ne uudelleen, ja niistä ei tyypillisesti ole mitään määritystä abstrakteissa luokissa. Pistä nämä kaksi petturia yhteen ja tajuat, että

- puhtaan virtuaalifunktion esittelyn tarkoitus on se, että periytetyt luokat perivät vain funktion *rajapinnan*.

Tämä on täysin järkevää `Shape::draw`-funktiolle, koska on järkevä vaatimus, että kaikkien `Shape`-olioiden täytyy olla piirrettävissä (`draw`), mutta `Shape`-luokka ei voi tarjota mitään järkevää oletustoteutusta tuolle funktiolle. Algoritmi, jolla piirretään ellipsi, on hyvin erilainen kuin esimerkiksi algoritmi, jolla piirretään suorakulmio. Hyvä tapa tulkita `Shape::draw`-funktion esittely on sama kuin sanoisi aliluokkien suunnittelijoille: "Sinun täytyy tarjota `draw`-funktio, mutta minulla ei ole mitään käsitystä, kuinka aiot toteuttaa sen."

Puhtaalle virtuaalifunktiolle o*n* sattumalta mahdollista tarjota määritys. Tämä tarkoittaa sitä, että voisit tarjota toteutuksen `Shape::draw`-funktiolle ja C++ ei valittaisi, mutta ainoa tapa kutsua sitä olisi määrittää kutsu täysin luokan nimellä:

```
Shape *ps = new Shape;          // virhe! Shape on abstr.

Shape *ps1 = new Rectangle;     // toimii
ps1->draw();                    // kutsuu Rectangle::draw

Shape *ps2 = new Ellipse;       // toimii
ps2->draw();                    // kutsuu Ellipse::draw

ps1->Shape::draw();             // kutsuu Shape::draw

ps2->Shape::draw();             // kutsuu Shape::draw
```

Siitä huolimatta, että voit tehdä ohjelmoijaystäviisi vaikutuksen cocktail-kutsuilla, tietoisuus tästä piirteestä on yleisesti ottaen rajoittunut työkalu. Kuten näemme yllä, se voidaan kuitenkin ottaa käyttöön mekanismina, jolla tarjotaan tavallista turvallisempi oletustoteutus yksinkertaisille (epäpuhtaille) virtuaalifunktioille.

Joskus on käytännöllistä esitellä luokka, joka ei sisällä *mitään muuta* kuin virtuaalifunktioita. Tällainen Protocol-luokka voi tarjota vain funktion rajapintoja periytetyille luokille, ei koskaan toteutuksia. Protocol-luokat kuvataan Kohdassa 34 ja ne mainitaan uudelleen Kohdassa 43.

Yksinkertaisten virtuaalifunktioiden takana oleva tarina on hieman erilainen kuin puhtaiden virtuaalifunktioiden. Kuten tavallista, periytetyt luokat perivät funktion rajapinnan, mutta yksinkertaiset virtuaalifunktiot tarjoavat perinteisesti toteutuksen, jonka periytetyt luokat valitsevat tai eivät valitse ohittamaan. Jos
mietit tätä minuutin, tajuat, että

- Yksinkertaisen virtuaalifunktion esittelyn tarkoitus on se, että *periytetyt luokat perivät funktion rajapinnan lisäksi oletustoteutuksen*.

`Shape::error`-funktion tapauksessa rajapinta määrää, että jokaisen luokan täytyy tukea kutsuttavaa funktiota silloin, kun tapahtuu virhe, mutta jokainen luokka on vapaa käsittelemään virheet sille sopivalla tavalla. Jos luokka ei halua tehdä mitään erityistä, se voi pelkästään tyytyä `Shape`-luokan tarjoamaan oletuksena olevaan virheidenkäsittelyyn. Tämä tarkoittaa sitä, että `Shape::error`-funktion esittely kertoo aliluokan suunnittelijoille: "Sinun täytyy tukea `error`-funktiota, mutta jos et halua kirjoittaa omaa funktiota, voit tyytyä `Shape`-luokan oletusversioon."

Käy ilmi, että voi olla vaarallista, että yksinkertaisten virtuaalifunktioiden sallitaan määrittää sekä funktion esittely että oletustoteutus. Jos haluat tietää miksi, tutki XYZ Airlines -yhtiön lentokoneiden hierarkiaa. XYZ-yhtiöllä on vain kahdenlaisia koneita, Malli A ja Malli B, ja molemmat lentävät täsmälleen samalla tavalla. XYZ suunnittelee täten seuraavan hierarkian:

```
class Airport { ... };          // esittää lentokenttiä

class Airplane {
public:
  virtual void fly(const Airport& destination);

  ...

};

void Airplane::fly(const Airport& destination)
{
  oletuskoodi, jolla lentokone lennätetään
  annettuun määränpäähän
}

class ModelA: public Airplane { ... };

class ModelB: public Airplane { ... };
```

Jotta ilmaistaisiin se, että kaikkien koneiden täytyy tukea `fly`-funktiota, ja tunnistettaessa se tosiasia, että lentokoneen eri mallit voisivat periaatteessa vaatia erilaisia toteutustapoja `fly`-funktiolle, `Airplane::fly`-funktio esitellään virtuaalisena. Kun kuitenkin identtisen koodin kirjoittamista `ModelA` ja `ModelB` -luokille nimenomaan vältetään, oletuksena oleva lentämiskäyttäytyminen tarjotaan funktion `Airplane::fly` rungossa, jonka sekä `ModelA` että `ModelB` perivät.

Tämä on klassista oliosuunnittelua. Kaksi luokkaa jakaa yhteisen piirteen (tavan, jolla ne toteuttavat `fly`-funktion), joten yhteinen piirre siirretään kantaluokkaan, ja kaksi luokkaa perivät piirteen. Tämä kaava tekee yleisistä piirteistä eksplisiittisiä, välttää koodin jäljentämistä, helpottaa tulevaisuuden arvonnousua ja helpottaa pitkällä tähtäyksellä ylläpitoa - kaikkia niitä asioita, joista oliopohjaista teknologiaa niin korkealle ylistetään. XYZ Airplanes -yhtiön kannattaa olla ylpeä.

Oletetaan nyt, että XYZ, jonka osakkeet ovat nousussa, päättää hankkia uuden tyyppisen lentokoneen, Mallin C. Malli C eroaa Mallista A ja Mallista B. Varsinkin siinä, että se lentää eri lailla.

XYZ:n ohjelmoijat lisäävät hierarkiaan luokan Mallille C, mutta hätiköidessään saadakseen uuden mallin palvelukseen, he unohtavat määrittää `fly`-funktion uudelleen:

```
class ModelC: public Airplane {
  ...                          // mitään fly-funktiota
                               // ei ole esitelty
};
```

Heillä on koodissaan täten jotain, joka on seuraavalle sukua:

```
Airport JFK(...);              // JFK on lentokenttä
                               // New York Cityssä

Airplane *pa = new ModelC;

...

pa->fly(JFK);                  // kutsuu Airplane::fly!
```

Tämä on onnettomuus: tapahtuu yritys, jolla `ModelC`-oliota yritetään lennättää samalla tavalla, kuin se olisi `ModelA` tai `ModelB`. Tämä ei ole senkaltaista käyttäytymistä, joka herättäisi luottamusta matkustavassa yleisössä.

Ongelma ei ole siinä, että funktiolla `Airplane::fly` on oletuskäyttäytyminen, vaan siinä, että `ModelC`-luokan sallittiin periä tuo käyttäytyminen sanomatta eksplisiittisesti, että se halusi sitä. Onneksi oletuksena oleva käyttäytyminen on helppo tarjota aliluokille, ja sitä ei määritetä niille, paitsi jos ne eivät pyydä määrittämistä. Temppu on siinä, että virtuaalifunktion *rajapinnan* ja sen *oletustoteutuksen* väliltä katkaistaan yhteys. Tässä on yksi tapa tehdä se:

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;

  ...

protected:
  void defaultFly(const Airport& destination);
};
```

```
void Airplane::defaultFly(const Airport& destination)
{
  oletuskoodi, jolla lentokone lennätetään
  annettuun määränpäähän
}
```

Huomaa kuinka `Airplane::fly`-funktiosta on käännetty *puhdas* virtuaalifunktio. Se tarjoaa rajapinnan lentämiselle. `Airplane`-luokassa on myös oletustoteutus, mutta nyt se on itsenäisen funktion `defaultFly` muodossa. Luokat, kuten `ModelA` ja `ModelB`, jotka haluavat käyttää oletuskäyttäytymistä, suorittavat yksinkertaisesti avoimena funktiona määritetyn kutsun niiden `fly`-funktion rungon sisällä olevaan `defaultFly`-funktioon (mutta katso Kohta 33, jos haluat tietoa avointen funktion ja virtuaalifunktioiden vuorovaikutuksesta):

```
class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { defaultFly(destination); }

  ...

};

class ModelB: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { defaultFly(destination); }

  ...

};
```

`ModelC`-luokalla ei ole mitään mahdollisuutta periä `fly`-funktion väärää toteutusta vahingossa, koska `Airplane`-luokan puhdas virtuaalifunktio pakottaa tarjoamaan oman versionsa `fly`-funktiosta.

```
class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);

  ...

};

void ModelC::fly(const Airport& destination)
{
  koodi, jolla ModelC-lentokone lennätetään annettuun
määränpäähän
}
```

Tämä skeema ei ole idioottivarma (ohjelmoijat voivat yhä kopioida ja liittää itsensä ongelmiin), mutta se on luotettavampi kuin alkuperäinen kaava. Mitä tulee `Airplane::defaultFly`-funktioon, se on suojattu, koska se on itse asiassa `Air-`

`plane`-luokan ja siitä periytettyjen luokkien toteutuksen yksityiskohta. Lentokoneita käyttävien asiakkaiden kannattaa huolehtia vain siitä, että heidät lennätetään, ei siitä, kuinka lentäminen on toteutettu.

On myös tärkeää se, että `Airplane::defaultFly` *ei* ole virtuaalifunktio. Tämä johtuu siitä, että minkään aliluokan ei pitäisi määrittää tätä funktiota uudelleen. Tälle totuudelle on omistettu Kohta 37. Jos `defaultFly` olisi virtuaalifunktio, sinulla olisi kehämäinen ongelma: mitä, jos jokin aliluokka unohtaa määrittää `defaultFly`-funktion uudelleen silloin, kun sen pitäisi?

Jotkut ihmiset ovat sitä ajatusta vastaan, että rajapinnan tarjoamiselle ja oletustoteutukselle on erilliset funktiot, kuten edellä mainitut `fly` ja `defaultFly`. He huomauttavat, että ensinnäkin ne saastuttavat luokan nimiavaruuden niihin läheisesti liittyvällä funktionimien uudiskasvulla. He ovat kuitenkin samaa mieltä siitä, että rajapinta ja oletustoteutus pitäisi erottaa toisistaan. Kuinka ratkaistaan tämän näennäinen ristiriita? Ottamalla hyöty siitä tosiasiasta, että puhtaat virtuaalifunktiot täytyy esitellä uudelleen aliluokissa, mutta niillä voi myös olla omat toteutuksensa. Tässä on tapa, jolla `Airplane`-hierarkia voisi hyötyä taidosta määrittää puhdas virtuaalifunktio:

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;

  ...

};

void Airplane::fly(const Airport& destination)
{
  oletuskoodi, jolla lentokone lennätetään annettuun
  määränpäähän
}

class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { Airplane::fly(destination); }

  ...

};

class ModelB: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { Airplane::fly(destination); }

  ...

};
```

```
class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);

  ...

};

void ModelC::fly(const Airport& destination)
{
  koodi, jolla ModelC-lentokone lennätetään annettuun
määränpäähän
}
```

Tämä on melkein sama työtapa kuin aikaisemmin, paitsi että puhtaan virtuaalifunktion `Airplane::fly` runko tulee itsenäisen `Airplane::defaultFly`-funktion tilalle. `fly`-funktio on sisäiseltä olemukseltaan hajotettu kahteen peruskomponenttiin. Sen esittely määrittää sen rajapinnan (jota periytettyjen luokkien *täytyy* käyttää) siinä, missä sen määritys määrittelee oletuskäyttäytymisen (jota periytetyt luokat voivat käyttää vain, jos ne pyytävät eksplisiittisesti sitä). Kun yhdistät `fly`- ja `defaultFly`-funktiot, olet kuitenkin menettänyt kyvyn antaa kahdelle funktiolle erilaiset suojaustasot: koodi, joka ennen oli tyypiltään `protected` (olemalla funktiossa `defaultFly`), on nyt tyypiltään `public` (koska se on `fly`-funktiossa).

Olemme lopulta saapuneet `Shape`-luokan epävirtuaalifunktioon, `objectID`. Silloin, kun jäsenfunktio on ei-virtuaalifunktio, sen ei odoteta käyttäytyvän eri lailla periytetyissä luokissa. Itse asiassa epävirtuaalinen jäsenfunktio erittelee *erikoistumisen ohittavan vakion*, koska se määrittää käyttäytymisen, jonka ei odoteta muuttuvan, riippumatta siitä, kuinka erikoistunut periytetystä luokasta tulee. Sellaisenaan:

- Epävirtuaalisen funktion esittelyn tarkoitus on se, että periytetyt luokat perivät funktion *rajapinnan lisäksi pakollisen toteutuksen*.

Voit ajatella `Shape::objectID`-funktion esittelystä kuin sanoisit: "Jokaisella `Shape`-oliolla on funktio, joka saa aikaan olion tunnistimen, ja tuo olion tunnistin lasketaan aina samalla tavalla. Tuo tapa määräytyy `Shape::objectID`-funktion määrityksellä, ja minkään periytetyn luokan ei tulisi yrittää muuttaa tapaa, jolla se on tehty." Koska epävirtuaalinen funktio yksilöi erikoistumisen ohittavan vakion, sitä ei koskaan pitäisi määrittää uudelleen alaluokassa. Tämä asia käsiteltiin yksityiskohtaisesti Kohdassa 37.

Puhtaasti virtuaalisten, yksinkertaisten virtuaalisten ja epävirtuaalifunktioiden esittelyiden erot sallivat sinun määrittää tarkasti sen, mitä haluat periytettyjen luokkien perivän: vain rajapinnan, rajapinnan ja oletustoteutuksen, tai rajapinnan ja pakollisen toteutuksen, kunkin erikseen. Koska nämä esittelyiden eri tyypit tarkoittavat pohjimmiltaan eri asioita, sinun täytyy valita huolellisesti niiden väliltä silloin, kun esittelet jäsenfunktiosi. Jos valitset, niin sinun kannattaa välttää kahta kaikkein yleisintä virhettä, jonka kokemattomat luokkien suunnittelijat tekevät.

Ensimmäinen virhe on, että kaikki funktiot esitellään epävirtuaalisina. Tämä ei jätä ollenkaan tilaa erikoistumiselle periytetyissä luokissa: varsinkin epävirtuaalimuodostinfunktiot ovat ongelmallisia (katso Kohta 14). On tietenkin täysin järkeenkäypää suunnitella luokka, jonka ei ole tarkoitus tulla käytetyksi kantaluokkana. Joukko yksinomaisesti epävirtuaalisia jäsenfunktioita on tässä tapauksessa sopiva. Tällaiset luokat esitellään kuitenkin liian usein joko tietämättömänä virtuaalisten ja epävirtuaalisten funktioiden eroista tai tuloksena siitä, että ei ole esitetty päteviä syitä huoleen virtuaalifunktioiden suorituksen kustannuksista. Tosiasia on, että melkein mikä tahansa luokka, jota tullaan käyttämään kantaluokkana, tulee sisältämään virtuaalifunktioita (katso jälleen Kohta 14).

Jos olet huolestunut virtuaalifunktioiden kustannuksista, salli minun ottaa esille 80-20 -sääntö (katso myös Kohta 33), joka esittää, että tyypillisessä ohjelmassa 80 prosenttia suoritusajasta kulutetaan suorittamalla vain 20 prosenttia koodista. Tämä sääntö on tärkeä, koska se tarkoittaa sitä, että keskimäärin 80 prosenttia funktiokutsuistasi voi olla virtuaalisia ilman, että sillä on pienintäkään tunnistettavissa olevaa vaikutusta ohjelmasi kokonaissuoritukseen. Ennen kuin harmaannut huolestuen, onko sinulla varaa virtuaalifunktion kustannuksiin, ota käyttöön yksinkertainen varokeino, jolla varmistat, että keskityt ohjelmasi siihen 20 prosenttiin, jossa päätöksellä voi todella olla merkitystä.

Toinen yleinen ongelma on, että *kaikki* jäsenfunktiot esitellään virtuaalisina. Joskus tämä on oikea tapa tehdä asiat - todista esimerkiksi Protocol-luokat (katso Kohta 34). Se voi kuitenkin olla merkki luokkasuunnittelijalle, jolta puuttuu selkärankaa ottaa vankka vastuu. Eräiden funktioiden *ei* pitäisi olla uudelleen määritettävissä periytetyissä luokissa, ja silloin, kun näin on asia, sinun täytyy kertoa se tekemällä näistä funktioista ei-virtuaalisia. Kukaan ei hyödy siitä, kun teeskentelet, että luokkasi voi olla kaikenkattava ihmisille, jos vain ottaa oman aikansa määrittää funktiosi uudelleen. Muista, että jos sinulla on kantaluokka B, joka on periytetty luokasta D, ja jäsenfunktio `mf`, tällöin jokaisen `mf`-funktioon tapahtuvista kutsuista *täytyy* toimia kunnolla:

```
D *pd = new D;
B *pb = pd;

pb->mf();                       // kutsuu mf-funktiota
                                // osoittimella kantaan

pd->mf();                       // kutsuu mf-funktiota
                                // osoitt. periytettyyn
```

`mf`-funktiosta täytyy joskus tehdä epävirtuaalifunktio varmistaaksesi, että asiat toimivat odotetulla tavalla (katso Kohta 37). Jos käytössä on erikoistumisen ohittava vakio, älä pelkää sanoa niin!

## Kohta 37: Älä koskaan määritä periytettyä epävirtuaalifunktiota uudelleen.

Tätä asiaa voidaan tutkia kahdella tavalla: teoreettisella ja pragmaattisella. Aloittakaamme pragmaattisella tavalla. Teoreetikothan ovat tunnetusti kärsivällisiä.

Oletetaan, että sanon sinulle, että luokka `D` on julkisesti periytetty luokasta `B,` ja että on olemassa julkinen jäsenfunktio `mf`, joka on määritetty luokassa `B`. Funktion `mf` parametrit ja paluutyyppi eivät ole tärkeitä, joten olettakaamme, että ne ovat molemmat `void`-tyyppisiä. Toisin sanoen, sanon tämän:

```
class B {
public:
  void mf();
  ...
};

class D: public B { ... };
```

Vaikka et tietäisi mitään `B`-luokasta, `D`-luokasta tai `mf`-funktiosta, kun otetaan olio `x`, joka on tyyppiä `D`,

```
D x;                            // x on d-tyyppinen olio
```

olisit varmaan aika yllättynyt, jos tämä:

```
B *pB = &x;                     // ota osoitin x:ään
pB->mf();                       // kuts. mf:ää osoittimella
```

käyttäytyisi eri lailla kuin tämä:

```
D *pD = &x;                     // ota osoitin x:ään
pD->mf();                       // kuts. mf:ää osoittimella
```

Tämä johtuu siitä, että molemmissa tapauksissa pyydät `x`-olioon avuksi jäsenfunktiota `mf`. Koska sama funktio ja sama olio ovat kyseessä molemmissa tapauksissa, niiden pitäisi käyttäytyä samalla tavalla, eikö niin?

Niinhän niiden pitäisi. Mutta voi olla, että ne eivät käyttäydy. Ne eivät käyttäydy niin varsinkaan, jos `mf` on epävirtuaalinen ja `D`-luokalla on määritettynä oma versio `mf`-funktiosta:

```
class D: public B {
public:
  void mf();                    // piilottaa B::mf-funkt.;
                                // katso Kohta 50
```

```
};
pB->mf();                    // kutsuu B::mf
pD->mf();                    // kutsuu D::mf
```

Syy tähän kaksinaamaiseen käyttäytymiseen on siinä, että *epävirtuaalifunktiot*, kuten `B::mf` ja `D::mf`, ovat staattisesti sidottuja (katso Kohta 38). Tämä tarkoittaa sitä, että koska `pB`:n on esitelty olevan tyyppiä osoitin `B`-luokkaan, `pB`:n kautta avuksi pyydetyt epävirtuaalifunktiot tulevat *aina* olemaan niitä, jotka on määritetty `B`-luokalle, vaikka jos `pB` osoittaa `B`-luokasta periytetyn luokan olioon, kuten se tekee tässä esimerkissä.

*Virtuaali*funktiot ovat toisaalta dynaamisesti sidottuja (katso jälleen Kohta 38), joten ne eivät kärsi tästä ongelmasta. Jos `mf` olisi virtuaalifunktio, kutsu `mf`-funktioon joko `pB`:n tai `pD`:n kautta johtaisi `D::mf`:-funktion avuksipyytämiseen, koska se mihin `pB` ja `pD` *todella* osoittavat, on olio, joka on tyypiltään `D`.

Pääasia tällöin on, että jos olet kirjoittamassa luokkaa `D` ja määrität uudelleen epävirtuaalifunktion `mf`, jonka perit luokasta `D`, `D`-oliot esittävät todennäköisesti skitsofreenista käyttäytymistä, ja päättävällä tekijällä ei ole mitään tekemistä itse olion kanssa, vaan esitellyn tyyppisellä osoittimella, joka osoittaa siihen. Viittaukset käyttäytyvät samalla tavalla hämmentävästi kuin osoittimet.

Se pragmaattisesta argumentista. Tiedän, että se mitä haluat nyt, on jonkinlainen teoreettinen oikeutus sille, että periytettyjä epävirtuaalifunktioita ei määritetä uudelleen. Olen mielissäni velvoitteesta.

Kohdassa 35 selitetään, että julkinen periytyvyys tarkoittaa samaa kuin *on eräänlainen*, ja Kohdassa 36 kuvataan sitä, miksi epävirtuaalifunktion esittely luokassa vakiinnuttaa erikoistumisen ohittavan vakion sille luokalle. Jos sovellat näitä havaintoja luokkiin `B` ja `D` ja epävirtuaaliseen jäsenfunktioon `B::mf`, niin silloin

- Kaikki, mikä on sovellettavissa `B`-olioihin, on myös sovellettavissa `D`-olioihin, koska jokainen `D`-olio *on eräänlainen* `B`-olio;
- `B`:n aliluokkien täytyy periä sekä `mf`-funktion rajapinta että toteutus, koska `mf` on epävirtuaalinen `B`:ssä.

Jos `D` määrittää `mf`-funktion nyt uudelleen, työtavassasi on ristiriita. Jos `D`:n täytyy *todella* toteuttaa `mf` eri lailla kuin `B`:n, ja jos jokaisen `B`-olion - ei väliä, kuinka erikoistuneen - täytyy *todella* käyttää `B`:n toteutusta `mf`-funktiosta, niin silloin ei yksinkertai-sesti ole totta, että jokainen `D` *on eräänlainen* `B`. Tässä tapauksessa `D`:n ei pitäisi julkisesti periytyä `B`:stä. Jos `D`:n toisaalta *todella* täytyy periytyä julkisesti `B`:stä, ja jos `D`:n *todella* täytyy toteuttaa `mf` erilaisella tavalla kuin `B`, silloin ei ole totta, että `mf` heijastaa erikoistumisen ohittavan vakion `B`:lle. Tässä tapauksessa `mf`-funktion pitäisi olla virtuaalifunktio. Lopulta, jos jokainen `D` todella *on eräänlainen* `B`, ja jos `mf` todella vastaa erikoistumisen ohittavan vakion `B`:lle, silloin `D`:n ei todella tarvitse määrittää `mf`-funktiota uudelleen, eikä sen pitäisikään yrittää tehdä sitä.

Riippumatta siitä mikä argumentti pätee, jonkun täytyy kelvata, ja missään tapauksessa se ei ole kielto määrittää uudelleen periytetty epävirtuaalifunktio.

## Kohta 38: Älä koskaan määritä uudelleen periytettyä parametrin oletusarvoa.

Yksinkertaistakaamme tämä keskustelu heti aluksi. Oletusparametri voi olla olemassa vain osana funktiota, ja voit periä vain kahdentyyppisiä funktioita: virtuaalisia ja epävirtuaalisia. Täten ainoa tapa määrittää uudelleen parametrin oletusarvo on määrittää uudelleen peritty funktio. Uudelleen perityn ei-virtuaalifunktion uudelleen määrittäminen on kuitenkin aina väärin (katso Kohta 37), joten voimme turvallisesti rajoittaa keskusteluamme tässä tilanteeseen, jossa perit *virtuaali*funktion parametrin oletusarvolla.

Kun näin on asia, tämän Kohdan oikeutus tulee aika suoraviivaiseksi: virtuaalifunktiot ovat dynaamisesti sidottuja, mutta parametrien oletusarvot ovat staattisesti sidottuja.

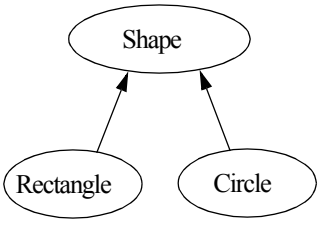
Mistä on kyse? Sinusta ehkä tuntuu, että et ole ajan tasalla uusimmasta oliopohjaisesta erikoiskielestä tai ehkä staattisen ja dynaamisen sidonnan ero on lipsahtanut sinun jo kuormittuneen mielesi ohi? Tarkastelkaamme asiaa sitten uudelleen.

Olion *staattinen tyyppi* on tyyppi, jonka määrittelet olevan ohjelman tekstissä. Tutki tätä luokkahierarkiaa:

```
enum ShapeColor { RED, GREEN, BLUE };

// geometristen muotojen luokka
class Shape {
public:
  // kaikkien muotojen täytyy tarjota funktio, jolla ne
  // piirtävät itsensä
  virtual void draw(ShapeColor color = RED) const = 0;

  ...

};


class Rectangle: public Shape {
public:
  // huomaa parametrin eri oletusarvo - väärin!
  virtual void draw(ShapeColor color = GREEN) const;

  ...

};
```

```
class Circle: public Shape {
public:
  virtual void draw(ShapeColor color) const;

  ...
};
```

Graafisena se näyttää tältä:




Tutki seuraavaksi näitä osoittimia:

```
Shape *ps;                    // static type = Shape*
Shape *pc = new Circle;       // static type = Shape*
Shape *pr = new Rectangle;    // static type = Shape*
```

`ps`, `pc` ja `pr` on tässä esimerkissä kaikki esitelty olevan tyyppiä osoitin `Shape`-luokkaan, joten se on niillä kaikilla staattisena tyyppinä. Huomaa, että sillä ei ehdottomasti kuitenkaan ole mitään eroa, mihin ne *todella* osoittavat - niiden staattinen tyyppi on joka tapauksessa `Shape*`.

Olion *dynaaminen tyyppi* on määritetty sen olion tyypillä, johon se sillä hetkellä viittaa. Tämä tarkoittaa sitä, että sen dynaaminen tyyppi osoittaa sen, kuinka se käyttäytyy. `pc`:n dynaaminen tyyppi on edellä mainitussa esimerkissä `Circle*`, ja `pr`:n dynaaminen tyyppi on `Rectangle*`. Mitä tulee `ps`:ään, sillä ei todella ole dynaamista tyyppiä, koska se ei (vielä) viittaa mihinkään olioon.

Dynaamiset tyypit, kuten niiden nimi viittaa, voivat muuttua ohjelman suorituksen aikana, tyypillisesti sijoitusten kautta:

```
ps = pc;                      // ps:n dynaaminen tyyppi
                              // on nyt Circle*

ps = pr;                      // ps:n dynaaminen tyyppi
                              // on nyt Rectangle*
```

Virtuaalifunktiot ovat *dynaamisesti sidottuja*, tarkoittaen sitä, että erityinen kutsuttu funktio määräytyy sen olion dynaamisella tyypillä, jonka kautta sitä on pyydetty avuksi:

```
pc->draw(RED);                // kutsuu Circle::draw(RED)
pr->draw(RED);                // kuts. Rectangle::draw(RED)
```

Tiedän, että tämä kaikki on vanhaa tietoa, ymmärrät jo varmaan virtuaalifunktiot. Mutkia tulee silloin, kun tutkit virtuaalifunktioita niiden parametrien oletusarvoilla, ja kuten sanoin edellä, virtuaalifunktiot ovat dynaamisesti sidottuja, mutta oletusparametrit ovat staattisesti sidottuja. Tämä tarkoittaa sitä, että päädyt ehkä pyytämään avuksi virtuaalifunktiota, joka on määritetty *periytetyssä luokassa*, mutta käytät *kantaluokan* parametrin oletusarvoa:

```
pr->draw();                    // kuts. Rectangle::draw(RED)!
```

`pr`:n dynaaminen tyyppi on tässä tapauksessa `Rectangle*`, joten `Rectangle`-luokan virtuaalifunktiota kutsutaan, juuri kuten odotitkin. Funktiossa `Rectangle::draw` parametrin oletusarvo on `GREEN`. Koska `pr`:n staattinen tyyppi on kuitenkin `Shape*`, tämän funktiokutsun parametrin oletusarvo otetaan `Shape`-luokasta, ei `Rectangle`-luokasta! Tuloksena on kutsu, joka koostuu oudosta ja melko varmasti ennalta aavistamattomasta yhdistelmästä sekä `Shape`- että `Rectangle`-luokan esittelyitä `draw`-funktiolle. Luota minuun, kun kerron, että et varmastikaan halua ohjelmasi käyttäytyvän tällä tavalla, tai usko ainakin minua, kun kerron sinulle, että *asiakkaasi* eivät halua ohjelmasi käyttäytyvän tällä tavalla.

Tarpeetonta on sanoa, että `ps`, `pc` ja `pr` ovat osoittimia, joilla ei tässä asiassa ole jälkiseurauksia. Jos ne olisivat viittauksia, ohjelma ei hellittäisi. Tärkeää on ainoastaan se, että `draw` on virtuaalifunktio ja yksi sen parametrien oletusarvoista on määritetty uudelleen aliluokassa.

Miksi C++-kieli vaatii käyttäytymään tällä perverssillä tavalla? Vastauksen täytyy liittyä suoritusaikaiseen tehokkuuteen. Jos parametrien oletusarvot olisivat dynaamisesti sidottuja, kääntäjien täytyisi keksiä tapa, jolla määritetään sopiva(t) oletusarvo(t) virtuaalifunktioiden parametreille suorituksen aikana, mikä olisi hitaampaa ja paljon monimutkaisempaa kuin nykyinen mekanismi, jolla ne määritetään kääntämisen aikana. Päätöksenä on hairahduksesta nopeuden ja toteutuksen yksinkertaisuuden puolelle, ja tuloksena on, että nautit nyt suorituskäyttäytymisestä, joka on tehokas, mutta hämmentävä, jos et ota vaarin tämän Kohdan ohjeista.

## Kohta 39: Vältä periytyvyyshierarkian alasmuunnoksia.

Näinä talouden myllerryksen aikoina on hyvä ajatus pitää silmällä taloudellisia instituutioitamme, joten tutki pankkitilien `Protocol`-luokkaa (katso Kohta 34):

```
class Person { ... };

class BankAccount {
public:
  BankAccount(const Person *primaryOwner,
              const Person *jointOwner);
  virtual ~BankAccount();

  virtual void makeDeposit(double amount) = 0;
  virtual void makeWithdrawal(double amount) = 0;

  virtual double balance() const = 0;

  ...
};
```

Monet pankit tarjoavat nyt häkellyttävän joukon tilityyppejä, mutta pitääksemme asiat yksinkertaisina olettakaamme, että on olemassa vain yhdentyyppinen pankkitili, nimittäin talletustili:

```
class SavingsAccount: public BankAccount {
public:
  SavingsAccount(const Person *primaryOwner,
                 const Person *jointOwner);
  ~SavingsAccount();

  void creditInterest();       // lisää tiliin korko

  ...
};
```

Tämä ei paljoakaan muistuta talletustiliä, mutta toisaalta, mikä *muistuttaa* näinä päivinä? Se on kuitenkin riittävä meidän tarkoituksiimme millä tahansa tasolla.

Pankki tulee melko varmasti pitämään kaikista tileistään listaa, joka on ehkä toteutettu peruskirjaston `list`-luokkamallin avulla (katso Kohta 49). Oletetaan, että tämä lista on mielikuvituksellisesti nimetty nimellä `allAccounts`:

```
list<BankAccount*> allAccounts;    // kaikki pankin
                                   // tilit
```

Kuten kaikki perussäilöt, `list`-luokat tallentavat niihin sijoitettujen asioiden *kopiot*, joten välttääkseen useiden kopioiden tallentamista jokaisesta `BankAccount`-tilistä, pankki on päättänyt niin, että `allAccounts` säilyttää osoittimet `BankAccount`-olioihin itse `BankAccount`-olioiden sijasta.

Kuvitellaan nyt, että sinun on tarkoitus kirjoittaa koodia, jolla iteroidaan kaikki tilit, aktivoiden erääntyvä korko jokaiselta. Voisit yrittää tätä:

```
// silmukka, joka ei käänny (katso alla, jos et ole kos-
// kaan ennen nähnyt koodin käyttävän "iteraattoreita")
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {

  (*p)->creditInterest();      // virhe!

}
```

mutta kääntäjäsi palauttaisivat pian maan pinnalle: `allAccounts` sisältää osoittimet `BankAccount`-olioihin, ei `SavingsAccount`-olioihin, joten `p` osoittaa silmukan jokaisella kierroksella `BankAccount`-olioon. Tämä tekee kutsusta `creditInterest`-funktioon kelvottoman, koska `creditInterest` on esitelty vain `SavingsAccount`-olioille, ei `BankAccount`-olioille.

Jos lause "`list<BankAccount*>::iterator p = allAccounts.begin()`" näyttää sinun mielestäsi enemmän voimansiirtolinjan metelitä kuin C++-kieleltä, sinulla ei ilmeisesti koskaan ole ollut iloa tavata peruskirjaston säiliöluokan malleja. Tämä osa kirjastoa tunnetaan yleensä nimellä Standard Template Library (se "STL"), ja voit lukea yleiskuvauksen siitä Kohdasta 49. Sinun pitää tällä hetkellä tietää ainoastaan se, että muuttuja `p` toimii osoittimena, joka käy silmukalla läpi `AllAccounts`-luokan kaikki elementit, alusta loppuun. Tämä tarkoittaa sitä, että `p` toimii, kuin sen tyyppi olisi `BankAccount**` ja listan elementit olisi tallennettu taulukkoon.

On turhauttavaa, että edellä mainittu silmukka ei käänny. `allAccounts` on tottakai määritetty tallentamaan `BankAccount*`-osoittimet, mutta *tiedät*, että se tosiaankin tallentaa kaikki `SavingsAccount*`-osoittimet edellä mainitussa silmukassa, koska `SavingsAccount` on ainoa luokka, joka voidaan instantioida. Typerät kääntäjät! Päätät kertoa niille sen, minkä tiedät olevan itsestään selvää ja mikä on niille liian typerää, että ne pystyisivät hahmottamaan itse: `allAccounts` sisältää todella `SavingsAccount*`-osoittimet:

```
// silmukka, joka kääntyy, mutta on joka tapauksessa huono
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {

  static_cast<SavingsAccount*>(*p)->creditInterest();

}
```

Kaikki ongelmasi on ratkaistu! Ratkaistu selvästi, ratkaistu tyylikkäästi, ratkaistu suppeasti, kaikki tämä käyttämällä yksinkertaisesti muunnosta. Vaikka tiedät, minkä tyyppisen osoittimen `allAccounts` todella tallentaa, höyrypäiset kääntäjäsi eivät tiedä, joten käytät muunnosta kertomaan sen niille. Mikä voisikaan olla loogisempaa?

On olemassa raamatullinen rinnakkaisilmiö, jonka haluaisin tuoda esiin tässä. Muunnokset ovat C++-ohjelmoijille sitä, mitä omena oli Eevalle.

Tämän tyyppistä muunnosta kantaluokan osoittimesta periytettyyn luokkaosoittimeen - kutsutaan nimellä "alasmuunnos", koska muunnat periytymisen hierarkian alas. Alasmuunnos sattuu toimimaan juuri tarkastelemassasi esimerkissä, mutta se johtaa, kuten näet pian, ylläpidolliseen painajaiseen.

Mutta menkäämme takaisin pankkiin. Olettakaamme talletustilien suosion viitoittamana, että pankki päättää tarjota myös shekkitilejä. Olettakaamme lisäksi, että shekkitilit kasvavat myös korkoa, kuten talletustilitkin:

```
class CheckingAccount: public BankAccount {
public:
  void creditInterest();       // lisää korko tiliin

  ...
};
```

On tarpeetonta sanoa, että `allAccounts` on nyt lista, joka sisältää osoittimia sekä talletus- että shekkitileille. Koron laskeva silmukka, jonka kirjoitit aikaisemmin, on yhtäkkiä vakavissa vaikeuksissa.

Ensimmäinen ongelmasi on, että se kääntyy edelleen, vaikka et ole muuttanut sitä niin, että se heijastaisi `CheckingAccount`-olioiden olemassaolon. Tämä johtuu siitä, että kääntäjät uskovat hupsusti sinua, kun kerrot niille (`static_cast`-operaattorin avulla), että `p*`-osoitin osoittaa todella `SavingsAccount*`-osoittimeen. Sinähän olet pomo. Tämä on Ylläpidon Ongelma Numero Yksi. Ylläpidon Ongelma Numero Kaksi on se, mitä olet pakotettu tekemään korjataksesi ongelman, tämä on tyypillisesti se, että kirjoitat tämänkaltaista koodia:

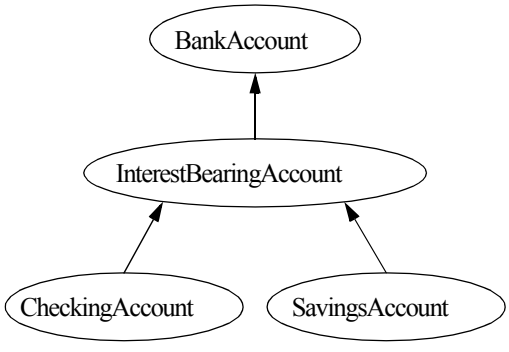
```
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {

  if (*p osoittaa SavingsAccount-luokkaan)
    static_cast<SavingsAccount*>(*p)->creditInterest();
  else
    static_cast<CheckingAccount*>(*p)->creditInterest();

}
```

Aina, kun löydät itsesi kirjoittamasta koodia, joka on muotoa "jos olio on tyyppiä `T1`, tee jotain, mutta jos se on tyyppiä `T2`, tee jotain muuta", läpsäytä itseäsi poskelle. Tämä ei ole C++:n Tapa. Se on kyllä järkevä strategia C- tai Pascal-kielessä, mutta ei C++-kielessä. C++-kielessä käytetään virtuaalifunktioita.

Muista, että *kääntäjät* ovat virtuaalifunktioiden kanssa vastuussa varmistamisesta, että oikeaa funktiota kutsutaan, riippuen käytössä olevan olion tyypistä. Älä sotke koodiasi ehdoilla tai switch-lauseilla; anna kääntäjiesi tehdä työ. Tässä tapa:

```
class BankAccount { ... };     // kuten edellä

// uusi luokka esittää tilejä, jotka keräävät korkoa
class InterestBearingAccount: public BankAccount {
public:
  virtual void creditInterest() = 0;

  ...

};

class SavingsAccount: public InterestBearingAccount {

  ...                          // kuten edellä

};

class CheckingAccount: public InterestBearingAccount {

  ...                          // kuten edellä

};
```

Graafisesti se näyttää tältä:




Koska sekä talletus- että shekkitilit kasvavat korkoa, haluaisit luonnollisesti siirtää tuon yleisen käyttäytymisen yleiseen kantaluokkaan. Olettaen kuitenkin, että pankin kaikki tilit eivät välttämättä kasva korkoa (varmasti oikea olettamus minun kokemuksieni pohjalta), et voi siirtää sitä `BankAccount`-luokkaan. Tästä on tuloksena, että olet esitellyt uuden `BankAccount`-luokan aliluokan, jonka nimi on `InterestBearingAccount`, ja olet muodostanut luokat `SavingsAccount` ja `CheckingAccount`, jotka periytyvät siitä.

`InterestBearingAccount`-luokan puhdas virtuaalifunktio `creditInterest`, osoittaa sen tosiasian, että sekä talletus- että shekkitilit kasvavat korkoa. Se on oletettavasti määritetty uudelleen sen aliluokissa `SavingsAccount` ja `CheckingAccount`.

Tämä uusi luokkahierarkia antaa sinun kirjoittaa silmukkasi uudelleen seuraavalla tavalla:

```
// parempi, mutta ei vieläkään täydellinen
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {
  static_cast<InterestBearingAccount*>(*p)->creditInterest();
}
```

Vaikka tässä silmukassa on yhä inhottavan pieni *muunnos*, se on paljon vankempi kuin se oli ennen, koska se jatkaa toimintaansa, vaikka ohjelmaasi lisättäisiin `InterestBearingAccount`-luokan uudet aliluokat.

Jos haluat päästä tästä muunnoksesta eroon kokonaan, sinun täytyy tehdä joitakin lisämuutoksia työtapaasi. Yksi työtapa on tiivistää tililuettelosi määritystä. Jos voisit saada listan `InterestBearingAccount`-olioista `BankAccount`-olioiden listan sijasta, kaikki olisi loistavaa:

```
// kaikki korkoa keräävät tilit pankissa
list<InterestBearingAccount*> allIBAccounts;

// silmukka, joka kääntyy ja toimii, nyt ja ikuisesti
for (list<InterestBearingAccount*>::iterator p =
       allIBAccounts.begin();
     p != allIBAccounts.end();
     ++p) {
  (*p)->creditInterest();
}
```

Jos erikoistuneemman listan saamista ei ole valittavissa, voisi olla järkevää sanoa, että `creditInterest`-toiminto on sovellettavissa *kaikille* pankkitileille, mutta toimintoa ei olisi valittavissa niille tileille, jotka eivät kasva korkoa. Tämä voitaisiin ilmaista tällä tavalla:

```
class BankAccount {
public:
  virtual void creditInterest() {}
  ...
};
class SavingsAccount: public BankAccount { ... };
class CheckingAccount: public BankAccount { ... };
list<BankAccount*> allAccounts;
```

```
// katso äiti, ei muunnosta!
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {

  (*p)->creditInterest();

}
```

Huomaa, että virtuaalinen funktio `BankAccount::creditInterest` tarjoaa tyhjän oletustoteutuksen. Tämä on mukava tapa tarkentaa, että sen käyttäytyminen ei oletuksena ole valittavissa, mutta se voi olla omiaan johtamaan aavistamattomiin vaikeuksiin. Sisäpiirin tietona, miksi samoin kuin kuinka eliminoida vaara, lue Kohta 36. Huomaa myös, että `creditInterest` on (implisiittisesti) muodostettu avoin funktio. Siinä ei ole mitään väärää, mutta koska se on myös virtuaalifunktio, `inline`-direktiivi tullaan luultavasti jättämään huomiotta. Kohdassa 33 selvitetään miksi.

Kuten olet nähnyt, alasmuunnokset voidaan eliminoida monella eri tavalla. Paras tapa on korvata tällaiset muunnokset kutsuilla virtuaalifunktioihin, tehden myös mahdollisesti jokaisesta funktiosta valitsemattoman mille tahansa luokalle, jolle se ei todella ole käytössä. Toinen metodi on tiivistää kirjoittamista niin, että osoittimen esitellyllä tyypillä ja osoittimen tyypillä, jonka tiedät olevan siellä, ei ole monimerkityksisyyttä. Olkoon vaadittu ponnistus mikä tahansa päästäksemme eroon alasmuunnoksista, se kannattaa, koska alasmuunnokset ovat rumia ja alttiita virheille, ja ne johtavat koodiin, jota on vaikea ymmärtää, kehittää ja ylläpitää.

Se, mitä olen juuri kirjoittanut, on totuus ja vain totuus. Se ei kuitenkaan ole koko totuus. On tilanteita, jolloin sinun *täytyy* todella tehdä alasmuunnos.

Oletetaan esimerkiksi, että kohtaisit tilanteen, jota käsittelimme tämän Kohdan alussa, eli `allAccounts` säilyttäisi `BankAccount`-osoittimet, `creditInterest` olisi määritetty vain `SavingsAccount`-olioille, ja sinun täytyisi kirjoittaa silmukka, jolla keräisit kaikkien tilien korot. Oletetaan edelleen, että kaikki nämä asiat olisivat hallintamme ulkopuolella; et voisi muuttaa `BankAccount-`, `SavingsAccount-` tai `allAccounts`-olioiden määrityksiä. (Näin voisi tapahtua, jos ne olisivat määritettyjä kirjastossa, johon sinulla olisi luku- ja kirjoitusoikeus.) Jos näin olisi asia, sinun *olisi* pitänyt käyttää alasmuunnosta, huolimatta siitä, kuinka mauttomalta asia sinusta tuntuisi.

On joka tapauksessa olemassa parempi tapa tehdä se kuin edellä näkemämme raaka muunnos. Parempaa tapaa kutsutaan nimellä "turvallinen alasmuunnos", ja se toteutetaan C++-kielen `dynamic_cast`-operaattorin avulla. Kun käytät `dynamic_cast`-operaattoria osoittimella, muunnosta yritetään, ja jos se onnistuu (eli jos osoittimen dynaaminen tyyppi (katso Kohta 38) vastaa sitä, mistä se on muunnettu), uuden tyypin oikea osoitin palautetaan. Jos `dynamic_cast`-operaattori epäonnistuu, null-arvoinen osoitin palautetaan.

Tässä on pankkiesimerkki, johon on lisätty turvallinen muunnos alaspäin:

```
class BankAccount { ... };       // samalla tavalla kuin
                                 // tämän Kohdan alussa

class SavingsAccount:            // samat sanat
  public BankAccount { ... };

class CheckingAccount:           // samat sanat taas
  public BankAccount { ... };

list<BankAccount*> allAccounts;  // tämän pitäisi näyttää
                                 // tutulta...

void error(const string& msg);   // virheenkäsittel. funktio;
                                 // katso alla

// no, äiti, ainakin muunnokset ovat turvallisia...
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {

  // yritä turvallista alasmuunnosta *p SavingsAccount*;
  // katso alapuolelle, jos haluat tietoa
  // psa:n määrityksestä
  if (SavingsAccount *psa =
        dynamic_cast<SavingsAccount*>(*p)) {
    psa->creditInterest();
  }

  // yritä turvallista alasmuunnosta CheckingAccount-luok.
  else if (CheckingAccount *pca =
             dynamic_cast<CheckingAccount*>(*p)) {
    pca->creditInterest();
  }

  // oh hoh — tuntematon tilityyppi
  else {
    error("Tuntematon tilityyppi!");
  }
}
```

Tämä skeema on kaukana ihanteellisesta, mutta voit ainakin tunnistaa sen, kun alasmuunnoksesi epäonnistuu. Tämä on mahdotonta ilman `dynamic_cast`-operaattorin käyttöä. Huomaa kuitenkin, että harkitsevaisuus sanelee sinulle sen, että tarkistat tapauksen, missä *kaikki* muunnokset alaspäin epäonnistuvat. Tämä on edellä mainitussa koodissa olevan viimeisen `else`-lauseen tarkoitus. Virtuaalifunktioille tällaiselle testille ei ole tarvetta, koska jokaisen virtuaalikutsun täytyy johtaa *johonkin* funktioon. Kun aloitat muunnoksen alaspäin, kaikki vedot on lyöty lukkoon. Jos joku lisäisi esimerkiksi uudentyyppisen tilin hierarkiaan, mutta epäonnistuisi päivittämään edellä mainitun koodin, kaikki muunnokset alaspäin epäonnistuisivat.

Tämän takia on tärkeää, että käsittelet tuon mahdollisuuuden. Todennäköisesti ei ole tarkoitus, että kaikki muunnokset voivat epäonnistua, mutta kun sallit muunnoksen alaspäin, hyvillekin ohjelmoijille alkaa tapahtua pahoja asioita.

Puhdistitko silmälasisi paniikissa, kun edellä huomasit `if`-lauseiden ehtolauseissa ne, jotka näyttävät muuttujan määrittelyiltä? Jos huomasit, älä huolehdi, näkösi on hyvä. Kyky määrittää tällaisia muuttujia lisättiin kieleen samaan aikaan kuin `dynamic_cast`-operaattori. Tämän piirteen avulla voidaan kirjoittaa siistimpää koodia, koska et todella tarvitse `psa`- tai `pca`-muuttujia, paitsi jos `dynamic_cast`-operaattorit, jotka alustavat ne, onnistuisivat, ja uudella syntaksilla näitä muuttujia ei tarvitse määrittää muunnokset sisältävien ehtolauseiden ulkopuolella. (Kohdassa 32 selvitetään, miksi yleensä ottaen haluat joka tapauksessa välttää haitallisia muuttujien määrityksiä.) Jos kääntäjäsi eivät vielä tue tätä uutta tapaa määrittää muuttujia, voit tehdä sen vanhalla tavalla:

```
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {

  SavingsAccount *psa;          // perinteinen määritys
  CheckingAccount *pca;         // perinteinen määritys

  if (psa = dynamic_cast<SavingsAccount*>(*p)) {
    psa->creditInterest();
  }

  else if (pca = dynamic_cast<CheckingAccount*>(*p)) {
    pca->creditInterest();
  }

  else {
    error("Unknown account type!");
  }
}
```

Asioiden hienossa skeemassa, paikkaan, johon tietysti sijoitat muuttujien, kuten `psa` ja `pca` määritykset, on vähän jälkiseuraamuksia. Tärkeää on tämä: `if-then-else` -tyylinen ohjelmointi, johon muunnos alaspäin väistämättä johtaa, on suunnattomasti huonompaa kuin virtuaalifunktioiden käyttäminen, ja sinun kannattaisi säästää niiden käyttö tilanteisiin, jolloin ei todellakaan ole vaihtoehtoja. Jos sinulla on onnea, et tule koskaan kohtaamaan näin kolkkoa ja ankeaa ohjelmointimaisemaa.

## Kohta 40: Mallinna *omistaa* tai *on toteutettu jonkin ehdoilla* kerrosajattelun kautta.

*Kerrosajattelu* on prosessi, jossa rakennetaan yksi luokka toisen luokan päälle niin, että kerroksessa olevassa luokassa on kerrostetun luokan olio tietojäsenenä. Esimerkiksi:

```
class Address { ... };          // jossa joku asuu

class PhoneNumber { ... };

class Person {
public:
  ...

private:
  string name;                  // kerrosolio
  Address address;              // samat sanat
  PhoneNumber voiceNumber;      // samat sanat
  PhoneNumber faxNumber;        // samat sanat
};
```

Tässä esimerkissä `Person`-luokan sanotaan olevan kerrostettuna luokkien `string`, `Address ja PhoneNumber` päällä, koska se sisältää näiden tyyppien tietojäseniä. Termillä kerrosajattelu on useita synonyymejä. Se tunnetaan myös nimellä *asetelma* (composition), *säilöntä* (containment) ja *upotus* (embedding).

Kohdassa 35 selvitetään, että julkinen periytyminen tarkoittaa samaa kuin *on eräänlainen*. Vastakohtaisesti kerrosajattelu tarkoittaa joko samaa kuin *omistaa* tai *on toteutettu jonkin ehdoilla*.

Edellä mainittu `Person`-luokka esittää *omistaa*-suhteen. `Person`-oliolla on nimi, osoite ja puhelinnumerot puhelin- ja fax-tietoliikenteelle. Et sanoisi, että henkilö *on* nimi tai että henkilö *on* osoite. Sanoisit, että henkilö *omistaa* nimen ja henkilö *omistaa* osoitteen ja niin edelleen. Useimmilla ihmisillä ei ole hankaluuksia tämän erikoispiirteen kanssa, joten *on eräänlainen*- ja *omistaa* -roolien välinen epäjärjestys on suhteellisen harvinaista.

Jokseenkin vaivalloisempaa on *on eräänlainen*- ja *on toteutettu jonkin ehdoilla* -suhteiden välinen ero. Oletetaan esimerkiksi, että tarvitset luokille mallin, jolla esität mielivaltaisen joukon olioita, toisin sanoen kokoelman ilman kaksoiskappaleita. Koska uudelleenkäyttö on hieno asia, ja koska viisaasti luit Kohdassa 49 olevan yleiskuvauksen C++-kirjastosta, otat vaistomaisesti ensimmäiseksi käyttöön kirjaston `set`-mallin. Miksi kirjoittaa uusi malli, kun voit käyttää vakiintunutta mallia, jonka joku muu on kirjoittanut?

Kun uppoudut `set`-mallin dokumentaatioon, huomaat kuitenkin rajoituksen, jota ilman ohjelmasi ei voi elää: `set`-malli vaatii, että sen sisältämien elementtien täytyy olla *täydellisesti järjestyksessä*, toisin sanoen jokaiselle joukon a:n ja b:n oliopareista täytyy olla määritettävissä, että `a<b` tai `b<a`. Tämä edellytys on helppo tyydyttää

useille tyypeille, ja se, että olioilla on täydellinen järjestys, suo `set`-mallille mahdollisuuden tarjota eräitä suoritukseen liittyviä puoleensavetäviä takeita. (Katso Kohta 49, jos haluat lukea lisätietoja suoritukseen liittyvistä takeista peruskirjastossa.) Tarpeesi liittyy kuitenkin johonkin yleisempään: `set`-tyyppinen luokka, jossa olioiden ei tarvitse olla täysin järjestyksessä, niiden pitää vain olla sitä, mitä C++-standardi värikkäästi kutsuu nimellä "vertailukelpoinen samanarvoisuus" (EqualityComparable): saman tyyppisille `a`- ja `b`-olioille on mahdollista määrittää, onko lause `a==b` totta. Tämä paljon vähäisempi vaatimus sopii paremmin tyypeille, jotka esittävät värin tapaisia asioita. Onko punainen vähemmän kuin vihreä tai onko vihreä vähemmän kuin punainen? Näyttää siltä, että sinun pitää kuitenkin kirjoittaa ohjelmallesi oma malli.

Uudelleenkäyttö *on* silti hieno asia. Tiedät tietorakenteiden eksperttinä, että joukkojen toteuttamisen lähes rajoituksettomista valinnoista eräs erityisen yksinkertainen tapa on ottaa käyttöön linkitetyt listat. Mutta arvaa mitä? `list`-malli (joka luo linkitettyjen listojen luokat) sijaitsee peruskirjastossa! Päätät käyttää sitä (uudelleen).

Päätät nimenomaan niin, että syntyvä `Set`-mallisi periytyy `list`-mallista. Tämä tarkoittaa sitä, että `Set<T>` periytyy `list<T>`:stä. `Set`-oliosta tulee toteutuksessasi itse asiassa `list`-olio. Esittelet täten `Set`-mallisi tähän tapaan:

```
// väärä tapa käyttää luetteloa Set-luokalle
template<class T>
class Set: public list<T> { ... };
```

Kaikki voi vaikuttaa hienolta ja korealta tällä hetkellä, mutta itse asiassa jotain on kuitenkin pielessä. Kuten Kohdassa 35 selitetään, jos `D` on `B`, kaikki mitä on `B`:ssä totta, on myös totta `D`:ssä. `list`-olio voi kuitenkin sisältää kaksoiskappaleita, joten jos arvo 3051 lisätään `list<int>`-listaan kahdesti, tuo lista sisältää kaksi kopiota luvusta 3051. Vastakohtaisesti voi olla, että `Set` ei sisällä kaksoiskappaleita, joten jos arvo 3051 lisätään `Set<int>`-listaan kahdesti, joukko sisältää vain yhden kopion arvosta. Täten on valheellista väittää, että `Set` *on eräänlainen* `list`, koska eräät asiat, jotka ovat totta `list`-olioille, eivät ole totta `Set`-olioille.

Koska näiden kahden luokan välinen suhde ei ole *on eräänlainen*, julkinen periytyvyys on väärä tapa mallintaa tätä suhdetta. Oikea tapa on ymmärtää, että `Set`-olio *voidaan toteuttaa* `list`-olion *ehdoilla*:

```
// oikea tapa käyttää luetteloa Set-luokalle
template<class T>
class Set {
public:
  bool member(const T& item) const;

  void insert(const T& item);
  void remove(const T& item);

  int cardinality() const;

private:
  list<T> rep;                  // sarjan esittely
};
```

`Set`-olion jäsenfunktiot voivat vankasti nojautua `list`-mallin ja peruskirjaston muiden osien jo tarjoamaan toiminnallisuuteen, joten toteutusta ei ole vaikea kirjoittaa eikä kovin innostavaa lukea:

```
template<class T>
bool Set<T>::member(const T& item) const
{ return find(rep.begin(), rep.end(), item) != rep.end(); }

template<class T>
void Set<T>::insert(const T& item)
{ if (!member(item)) rep.push_back(item); }

template<class T>
void Set<T>::remove(const T& item)
{
  list<T>::iterator it =
    find(rep.begin(), rep.end(), item);

  if (it != rep.end()) rep.erase(it);
}

template<class T>
int Set<T>::cardinality() const
{ return rep.size(); }
```

Nämä funktiot ovat tarpeeksi yksinkertaisia niin, että ne ovat järkeviä ehdokkaita avointen funktioiden määrittämiselle, vaikkakin tiedän, että haluat lukea uudelleen Kohdan 33 keskustelun ennen kuin teet mitään sitovia avointen funktioiden esittelyyn liittyviä päätöksiä. (Edellä mainitussa koodissa funktiot, kuten `find`, `begin`, `end`, `push_back` ja niin edelleen, ovat osa peruskirjaston kehystä, jonka avulla työskennellään `listin` kaltaisten mallien kanssa. Löydät yleisesityksen tästä kehyksestä Kohdassa 49.

Kannattaa huomata, että `Set`-luokan rajapinta epäonnistuu valmiin ja minimaalisen rajapinnan testissä (katso Kohta 18). Mitä tulee valmiuteen, ensisijainen laiminlyönti on tapa, jolla iteroidaan joukon sisällön läpi. Iterointi voi olla tarpeellista monille sovelluksille (ja tämä on peruskirjaston kaikkien jäsenten käytettävissä, mukaan lukien `Set`). Lisähaitta on se, että `Set` epäonnistuu noudattaessaan säiliöluokan sopimuk-

sia, joita peruskirjasto on vaalinut (katso Kohta 49), ja tämä tekee vaikeammaksi hyötyä kirjaston muista osista, kun työskentelet `Set`-tyyppisten luokkien kanssa.

`Set`-luokan rajapinnasta saivartelun ei pitäisi kuitenkaan antaa varjostaa sitä, minkä `Set` kieltämättä teki oikein: `Set`- ja `list`-mallien suhde. Tämä suhde ei ole *on eräänlainen* (vaikka alustavasti näytti siltä, että se voisi olla), se on *toteutettu jonkin ehdoilla*, ja kerrosajattelun käytöstä tämän suhteen toteutuksessa kuka tahansa luokkien suunnittelija voi hyvällä syyllä olla ylpeä.

Sattumalta, kun käytät kerrosajattelua luodessasi yhteyden kahden luokan välillä, luot kääntämisen aikaisen riippuvuuden näiden kahden luokan välille. Jos haluat tietoa siitä, miksi tämän pitäisi huolestuttaa sinua, tai mitä voit tehdä rauhoittaaksesi huolesi, lue Kohta 34.

## Kohta 41: Eriytä periytyvyys ja mallit.

Tutki kahta seuraavaa työtavan ongelmaa:

- Tietojenkäsittelyopin harras opiskelija kun olet, haluat luoda luokat, joilla esitetään oliopinoja. Tarvitset useita erilaisia luokkia, koska jokaisen pinon täytyy olla homogeeninen, toisin sanoen siinä voi olla vain yhden tyyppisiä olioita. Sinulla voisi esimerkiksi olla `int`-tyypeistä koostuva pinojen luokka, toinen `string`-tyyppisille pinoille, kolmas `string`-tyyppisten pinojen pinoille ja niin edelleen. Olet kiinnostunut tukemaan vain minimaalista rajapintaa luokkaan (katso Kohta 18), joten rajoitat toimintosi pinon luomiseen, pinon tuhoamiseen, olioiden työntämiseen pinoon, olioiden poimimiseen pinosta ja päättämiseen siitä, onko pino tyhjä. Jätät tämän harjoituksen kohdalla huomioimatta peruskirjaston luokat (mukaan lukien `stack` - katso Kohta 49), koska hingut sitä kokemusta, että kirjoitat koodin itse. Uudelleenkäyttö on hieno asia, ei ole mitään siihen verrattavaa kuin siihen uppoutuminen ja kätesi likaaminen.
- Olet kissojen harras ystävä ja haluat suunnitella luokat, joilla esitetään kissoja. Tarvitset useita erilaisia luokkia, koska jokainen kissarotu on hieman erilainen. Kuten kaikki oliot, kissat voidaan luoda ja tuhota, mutta kuten kaikki kissojen rakastajat tietävät, ainoat asiat, mitä kissat tekevät, ovat syöminen ja nukkuminen. Kuitenkin jokainen kissarotu syö ja nukkuu omalla hellyttävällä tavallaan.

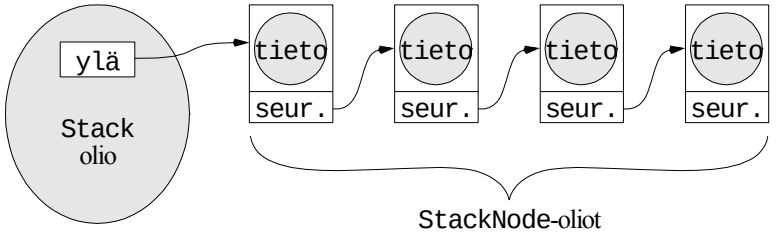
Nämä kaksi ongelmien määritystä tuntuvat samankaltaisilta, mutta niistä on silti lopputuloksena erilaiset ohjelmistosuunnitelmat. Miksi?

Vastauksen täytyy liittyä jokaisen luokan käyttäytymiseen ja käsiteltävissä olevan olion *tyypin* suhteeseen. Olet sekä pinojen että kissojen tapauksessa tekemisissä eri tyyppisten lajien kanssa (pinot, jotka sisältävät T-tyyppisiä objekteja, kissat, joiden rotu on T), mutta sinun täytyy kysyä itseltäsi tätä: vaikuttaako tyyppi T luokan T *käyttäytymiseen*? Jos T *ei* vaikuta *käyttäytymiseen*, sinun täytyy käyttää mallia. Jos T vaikuttaa käyttäytymiseen, voit käyttää virtuaalifunktioita, ja käytät sen tähden periytyvyyttä.

Tässä on tapa, jolla pystyisit määrittämään linkitetyn listan toteutuksen Stack-luokalle, olettaen, että pinottavat oliot ovat tyyppiä T:

```
class Stack {
public:
  Stack();
  ~Stack();

  void push(const T& object);
  T pop();

  bool empty() const;          // onko pino tyhjä?

private:
  struct StackNode {           // linkitetyn listan solmu
    T data;                    // tässä solmussa ol. tieto
    StackNode *next;           // listan seuraava solmu

    // StackNode-muodostin alustaa molemmat kentät
    StackNode(const T& newData, StackNode *nextNode)
    : data(newData), next(nextNode) {}
  };

  StackNode *top;              // pinon yläosa

  Stack(const Stack& rhs);          // estä kopiointi ja sijoi-
  Stack& operator=(const Stack& rhs); // tus (kts. Kohta 28)
};
```

Stack-oliot voisivat täten muodostaa tietorakenteita, jotka näyttävät tältä:




Itse linkitetty lista koostuu StackNode-olioista, mutta se on Stack-luokan toteutuksen yksityiskohta, joten StackNode on esitelty private-tyyppisenä Stack-oliona. Huomaa, että StackNode-oliolla on muodostinfunktio, joka varmistaa sen, että kaikki sen kentät on alustettu kunnolla. Se, että osaat vaikka unissasi kirjoittaa linkitettyjä listoja, ei ole syy siihen, että jätät pois teknologiset kehitykset kuten muodostinfunktiot.

Tässä on järkevä ensimmäinen leikkaus siitä, kuinka voisit toteuttaa `Stack`-olion jäsenfunktiot. Kuten monien prototyyppitoteutusten kanssa (ja liian monien tuotannossa olevien ohjelmien kanssa), mitään virheidentarkistusta ei ole, koska prototyyppisessä maailmassa mikään ei koskaan mene pieleen.

```
Stack::Stack(): top(0) {}      // yläosa alust. null-arv.

void Stack::push(const T& object)
{
  top = new StackNode(object, top);   // pane uusi solmu
}                                     // listan etuosaan

T Stack::pop()
{
  StackNode *topOfStack = top;  // muista yläosan solmu
  top = top->next;

  T data = topOfStack->data;    // muista solmun tieto
  delete topOfStack;

  return data;
}

Stack::~Stack()               // poista kaikki pinosta
{
  while (top) {
    StackNode *toDie = top;     // ota osoitin yläsolmuun
    top = top->next;            // siirry seuraavaan solm.
    delete toDie;               // poista ed. yläsolmu
  }
}

bool Stack::empty() const
{ return top == 0; }
```

Näissä toteutuksissa ei ole mitään pysyvää. Itse asiassa niiden ainoa kiinnostava asia on tämä: pystyt kirjoittamaan jokaisen jäsenfunktion, vaikka et pääasiallisesti tiedä mitään `T`-oliosta. (Oletat, että voit kutsua `T`:n kopiomuodostinta, mutta kuten Kohdassa 45 selitetään, se on aika järkevä olettamus.) Koodi, jonka kirjoitat muodostamiselle, tuhoamiselle, työntämiselle, noutamiselle ja päätökselle siitä, onko pino tyhjä, on sama huolimatta siitä, mikä `T` on. Lukuunottamatta olettamusta, että voit kutsua `T`:n kopiomuodostinta, `stack`-olion käyttäytyminen ei riipu `T`-luokasta millään tavalla. Tämä on malliluokan tunnistettavissa oleva luonteenpiirre: käyttäytyminen ei riipu tyypistä.

`Stack`-luokan muuntaminen malliksi on muuten niin yksinkertaista, että jopa Dilberin pystytukkainen pomo voisi tehdä sen:

```
template<class T> class Stack {
  ...                           // täsmälleen sama kuin ed.
};
```

Mutta siirrytään kissoihin. Miksi mallit eivät toimi kissoissa?

Lue speksi uudelleen ja huomaa vaatimus: "Jokainen kissarotu syö ja nukkuu omalla hellyttävällä tavallaan". Tämä tarkoittaa sitä, että sinun täytyy toteuttaa jokaisen kissatyypin *erilainen käyttäytyminen*. Et voi pelkästään kirjoittaa yhtä funktiota, joka hoitaa kaikki kissat, voit vain *määrittää rajapinnan* funktiolle, joka jokaisen kissatyypin täytyy toteuttaa. Ahaa! Tapa, jolla vain funktion *rajapinta* julkaistaan, on esitellä puhdas virtuaalifunktio (katso Kohta 36):

```
class Cat {
public:
  virtual ~Cat();              // katso Kohta 14

  virtual void eat() = 0;      // kaikki kissat syövät
  virtual void sleep() = 0;    // kaikki kissat nukkuvat
};
```

`Cat`-luokan aliluokkien - kuten `Siamese` ja `BritishShortHairedTabby` - täytyy tietenkin määrittää perimänsä rajapinnan `eat`- ja `sleep`-funktioiden rajapinnat uudelleen:

```
class Siamese: public Cat {
public:
  void eat();
  void sleep();

  ...

};

class BritishShortHairedTabby: public Cat {
public:
  void eat();
  void sleep();

  ...

};
```

Tiedät nyt, miksi mallit toimivat `Stack`-luokalle ja miksi ne eivät toimi `Cat`-luokalle. Tiedät myös, miksi periytyvyys toimii `Cat`-luokalle. Ainoa jäljellejäävä kysymys on, miksi periytyvyys ei toimi `Stack`-luokalle. Nähdäksesi miksi, yritä esitellä `Stack`-hierarkian juuriluokka, yksittäinen luokka, josta kaikki pinoluokat voivat periä:

```
class Stack {                  // kaiken kattava pino
public:
  virtual void push(const ??? object) = 0;
  virtual ??? pop() = 0;

  ...

};
```

Vaikeus on nyt varmaan selvä. Mitä tyyppejä aiot esitellä puhtaille virtuaalifunktioille kuten `push` ja `pop`? Muista, että jokaisen aliluokan täytyy määrittää uudelleen perimänsä virtuaalifunktiot *täsmälleen* samoilla parametrityypeillä ja paluuarvoilla, jotka ovat yhtenäisiä kantaluokan määritysten kanssa. `int`-tyypeistä koostuva pino haluaa valitettavasti suorittaa `push`- ja `pop`-operaatiot `int`-olioihin, siinä missä sanokaamme pino `Cats`-olioita haluaa suorittaa `push`- ja `pop`-komennot `Cat`-olioihin. Kuinka `Stack`-luokka voi esitellä sen puhtaat virtuaalifunktiot sellaisella tavalla, että asiakkaat voivat luoda pinoja sekä `int`-tyypeistä että `Cat`-olioiden pinoista? Kylmä ja kova totuus on, että se ei voikaan, ja tämän takia periytyvyys on sopimaton pinojen luontiin.

Mutta ehkä olet nuuskivaa tyyppiä. Ajattelet ehkä, että voit olla ovelampi kuin kääntäjäsi käyttämällä geneerisiä (`void*`) osoittimia. Käy ilmi, että geneeriset osoittimet eivät ole paljon avuksi. Et pysty vaikuttamaan siihen vaatimukseen, että periytetyissä luokissa olevat virtuaalifunktion esittelyt eivät koskaan saa olla ristiriidassa sen kantaluokassa olevan esittelyn kanssa. Geneeriset osoittimet voivat kuitenkin auttaa sinua toisessa ongelmassa, eli siinä, joka liittyy malleista luotujen luokkien tehokkuuteen. Jos haluat lukea yksityiskohtia, katso Kohta 42.

Nyt kun olemme vapautuneet pinoista ja kissoista, voimme luoda yhteenvedon tämän Kohdan harjoituksista seuraavasti:

- Mallia tulisi käyttää luomaan luokkien kokoelma silloin, kun olioiden tyyppi *ei vaikuta* luokan funktioiden käyttäytymiseen.
- Periytyvyyttä tulisi käyttää luokkien kokoelmaan silloin, kun olioiden tyyppi *vaikuttaa* luokkien käyttäytymiseen.

Sisäistä nämä kaksi pientä alakohtaa, ja sinulla on hyvät eväät matkallasi periytyvyyden ja mallien valinnan hallintaan.

## Kohta 42: Käytä yksityistä periytyvyyttä ymmärtäväisesti.

Kohdassa 35 esitetään, että C++-kieli käsittelee julkisen periytyvyyden *on eräänlainen* -suhteena. Se tekee tämän näyttämällä, että kääntäjät, silloin kun niille annetaan hierarkia, jossa `Student`-luokka julkisesti periytyy `Person`-luokasta, muuntavat `Student`-luokat implisiittisesti `Person`-luokiksi silloin, kun se on tarpeellista funktiokutsun onnistumiselle. Kannattaa toistaa osa tuosta esimerkistä käyttämällä yksityistä periytyvyyttä julkisen periytyvyyden sijasta:

```
class Person { ... };

class Student:                  // tällä kertaa käytämme
  private Person { ... };       // yksityistä periytyvyyttä

void dance(const Person& p);    // jokainen osaa tanssia

void study(const Student& s);   // vain opiskelijat opiske-
                                // levat
```

```
Person p;                    // p on Person
Student s;                   // s on Student

dance(p);                    // toimii, p on Person

dance(s);                    // virhe! Student ei ole
                             // Person
```

Yksityinen periytyminen ei selvästikään tarkoita samaa kuin *on eräänlainen*. Mitä se sitten tarkoittaa?

"Vau!" sanot. "Ennen kuin pääsemme tarkoitukseen, käsitelkäämme käyttäytyminen. Kuinka yksityinen periytyvyys käyttäytyy?" Olet juuri nähnyt yksityistä periytyvyyttä hallitsevan ensimmäisen säännön toiminnassa: julkiselle periytyvyydelle vastakohtana kääntäjät *eivät* yleisesti muunna periytettyä luokkaoliota (kuten `Student`) kantaluokan olioksi (kuten `Person`), jos luokkien välinen periytymissuhde on yksityinen. Tämän takia `s`-olion tanssiinkutsu epäonnistuu. Toinen sääntö on, että jäsenistä, jotka on periytetty yksityisestä kantaluokasta, tulee periytetyn luokan yksityisiä jäseniä, vaikka ne olisivat kantaluokassa suojattuja tai julkisia. Se käyttäytymisestä.

Tämä johtaa meidät tarkoitukseen. Yksityinen periytyvyys tarkoittaa samaa kuin *on toteutettu jonkin ehdoilla*. Jos teet niin, että olio `D` periytyy yksityisesti luokasta `B`, teet niin, koska olet kiinnostunut hyötymään osasta koodia, joka on jo kirjoitettu luokalle `B`, et siksi, että `B`-tyyppisten olioiden ja `D`-tyyppisten olioiden välillä olisi mitään käsitteellistä suhdetta. Yksityinen periytyvyys on tällaisena puhtaasti toteutustekniikka. Jos käytetään Kohdassa 36 esiteltyjä termejä, yksityinen periytyvyys tarkoittaa sitä, että vain toteutus pitäisi periyttää; rajapinta pitäisi jättää huomioonottamatta. Jos `D` periytyy yksityisesti `B`:stä, se tarkoittaa, että `D`-oliot toteutetaan `B`-olioiden ehdoilla, ei muuta. Yksityinen periytyvyys ei tarkoita mitään ohjelmistosuunnittelun aikana, vain ohjelmiston *toteutuksen* aikana.

Tosiasia, että yksityinen periytyvyys tarkoittaa samaa kuin *on toteutettu jonkin ehdoilla,* on hieman häiritsevää, koska Kohdassa 40 tähdennetään, että kerrosajattelu voi tarkoittaa samaa asiaa. Kuinka sinun oletetaan valitsevan niiden välillä? Vastaus on yksinkertainen: käytä kerrosajattelua aina kun voit, käytä yksityistä periytyvyyttä silloin, kun sinun on pakko. Koska on pakko? Silloin, kun suojatut jäsenet ja/tai virtuaalifunktiot astuvat kuvaan - mutta lisää siitä hetken kuluttua.

Kohdassa 41 nähdään tapa kirjoittaa `Stack`-malli, joka luo luokat, jotka tallentavat eri tyyppiset oliot. Haluat varmaan tutustua tähän Kohtaan nyt. Mallit ovat yksi C++-kielen käyttökelpoisimmista piirteistä, mutta kun alat käyttää niitä säännöllisesti, huomaat, että jos instantioit mallin kaksitoista kertaa, tulet luultavasti instantioimaan koodin mallille kaksitoista kertaa. `Stack`-mallin tapauksessa koodi, joka muodostaa `Stack<int>`-:n jäsenfunktiot, tulee olemaan kokonaan erillään koodista, joka muodostaa `Stack<double>`:n jäsenfunktiot. Joskus tämä ei ole vältettävissä, mutta tällainen koodin replikointi tulee olemaan olemassa, vaikka mallifunktiot pys-

tyisivät itse asiassa jakamaan koodin. Tuloksena olevalle oliokoodin koolle on olemassa nimi: mallin indusoima *koodipöhöttymä*. Tämä ei ole hyvä asia.

Eräille luokkatyypeille voidaan käyttää geneerisiä osoittimia koodipöhöttymän välttämiseksi. Luokat, joille tämä työtapa on sovellettavissa, varastoivat osoittimet olioiden sijasta ja ne toteutetaan:

1. Luomalla yksittäinen luokka, joka tallentaa `void*`-tyyppiset osoittimet olioihin.
2. Luomalla joukko lisäluokkia, joiden ainoa tarkoitus on pakottaa voimakas tyypittäminen. Nämä kaikki luokat käyttävät itse työhön askeleessa 1 olevaa geneeristä luokkaa.

Tässä on esimerkki, jossa käytetään Kohdan 41 ei-malli-pohjaista `Stack`-luokkaa, paitsi että tässä se varastoi geneerisiä osoittimia olioiden sijasta:

```
class GenericStack {
public:
  GenericStack();
  ~GenericStack();

  void push(void *object);
  void * pop();

  bool empty() const;

private:
  struct StackNode {
    void *data;                 // tässä solm. oleva tieto
    StackNode *next;            // listan seuraava solmu

    StackNode(void *newData, StackNode *nextNode)
    : data(newData), next(nextNode) {}
  };

  StackNode *top;                         // pinon yläosa

  GenericStack(const GenericStack& rhs);  // estä kopiointi ja
  GenericStack&                           // sijoitus (katso
    operator=(const GenericStack& rhs);   // Kohta 27)
};
```

Koska tämä luokka tallentaa olioiden sijasta osoittimet, on mahdollista, että useammasta kuin yhdestä pinosta osoitetaan olioon (toisin sanoen ne on työnnetty useisiin pinoihin). Näin ollen on kriittisen tärkeää, että `pop` ja luokan muodostinfunktio *eivät* poista minkään tuhottavan `StackNode`-olion tieto-osoitinta, vaikka niiden täytyy jatkaa itse `StackNode`-olion poistamista. `StackNode`-oliolle on joka tapauksessa varattu muistia `GenericStack`-luokan sisällä, joten ne täytyy myös poistaa varaamasta muistia tuon luokan sisältä. Tästä on tuloksena, että Kohdassa 41 oleva `Stack`-luokan toteutus riittää melkein täydellisesti `GenericStack`-luokalle. Ainoat muutokset, jotka sinun pitää tehdä, koskevat `void*`-osoittimien korvaamista `T`:lle.

`GenericStack`-luokalla itse on vähän hyötykäyttöä - sitä on liian helppo käyttää väärin. Asiakas voisi esimerkiksi sijoittaa erehdyksessä osoittimen `Cat`-olioon pinoon, joka on tarkoitettu vain `int`-tyyppien osoittimien varastoksi, ja kääntäjät hyväksyisivät tämän iloisesti. Silloin kun puhutaan `void*`-tyyppisistä parametreistä, osoitin on osoitin.

Saavuttaaksesi jälleen tyyppiturvallisuuden, johon olet tottunut, luot rajapintaluokat `GenericStack`-luokkaan, tähän tyyliin:

```
class IntStack {              // int-tyypp. luokan rajap.
public:
  void push(int *intPtr) { s.push(intPtr); }
  int * pop() { return static_cast<int*>(s.pop()); }
  bool empty() const { return s.empty(); }

private:
  GenericStack s;             // toteutus
};

class CatStack {              // kissojen luok. rajapinta
public:
  void push(Cat *catPtr) { s.push(catPtr); }
  Cat * pop() { return static_cast<Cat*>(s.pop()); }
  bool empty() const { return s.empty(); }

private:
  GenericStack s;             // toteutus
};
```

Kuten voit nähdä, `IntStack`- ja `CatStack`-luokat ovat hyviä vain siksi, että ne pakottavat vahvan tyypittämisen. `IntStack`-pinoon voidaan työntää vain `int`-tyyppiset osoittimet tai ne voidaan vetää sieltä, ja `CatStack`-luokkaan voidaan työntää tai sieltä voidaan vetää vain `Cat`-osoittimia. Sekä `IntStack` että `CatStack` on toteutettu `GenericStack`-luokan ehdoilla, suhteella, joka on ilmaistu kerrosajattelun avulla (katso Kohta 40), ja `IntStack` ja `CatStack` jakavat niiden funktioiden koodin `GenericStack`-luokassa, jotka itse asiassa toteuttavat niiden käyttäytymisen. Lisäksi tosiasia, että kaikki `IntStack`- ja `CatStack`-jäsenfunktiot on (implisiittisesti) esitelty avoimina funktioina, tarkoittaa sitä, että näiden rajapintaluokkien suorituksenaikainen kustannus on nolla markkaa.

Mutta mitä jos mahdolliset asiakkaat eivät tajua tätä? Mitä jos he erehdyksessä uskovat, että `GenericStack`-luokan käyttö on tehokkaampaa, tai mitä jos he ovat pelkästään villejä ja hurjia, ja uskovat, että vain nynnyt tarvitsevat turvaverkkoja? Mikä estää heitä jättämästä huomioimatta `IntStack`- ja `CatStack`-luokat ja siirtymästä suoraan `GenericStack`-luokkaan, jolloin he olisivat vapaita tekemään sen tyyppisiä tyyppivirheitä, joita C++-kieli nimenomaan suunniteltiin estämään?

Ei mikään. Ei mikään estä sitä. Mutta ehkä jonkin pitäisi.

Mainitsin tämän Kohdan alussa, että julkisen periytyvyyden avulla voidaan vaihtoehtoisesti vakuuttaa *on toteutettu jonkin ehdoilla* -tyyppinen luokkien välinen suhde. Tässä tapauksessa tuo tekniikka tarjoaa edun kerrosajattelulle, koska se sallii sinun ilmaista ajatuksen, että `GenericStack` ei ole tarpeeksi turvallinen yleiseen käyttöön, ja että sitä pitäisi käyttää vain muiden luokkien toteuttamiseen. Määrität sen suojaamalla `GenericStack`-luokan jäsenfunktiot:

```
class GenericStack {
protected:
  GenericStack();
  ~GenericStack();

  void push(void *object);
  void * pop();

  bool empty() const;

private:
  ...                           // sama kuin edellä
};

GenericStack s;                 // virhe! muodostinfunktio
                                // on suojattu


class IntStack: private GenericStack {
public:
  void push(int *intPtr) { GenericStack::push(intPtr); }
  int * pop() { return static_cast<int*>(GenericStack::pop()); }
  bool empty() const { return GenericStack::empty(); }
};

class CatStack: private GenericStack {
public:
  void push(Cat *catPtr) { GenericStack::push(catPtr); }
  Cat * pop() { return static_cast<Cat*>(GenericStack::pop()); }
  bool empty() const { return GenericStack::empty(); }
};

IntStack is;                    // toimii

CatStack cs;                    // toimii myös
```

Kuten kerrosajattelu, julkiseen periytyvyyteen perustuva toteutus välttää koodin kaksinkertaistumista, koska tyyppiturvalliset rajapintaluokat eivät koostu muusta kuin avointen funktioiden kutsuista alla oleviin `GenericStack`-funktioihin.

Tyyppiturvallisten rajapintojen rakentaminen `GenericStack`-luokan päälle on aika ovela temppu, mutta kaikkien noiden rajapintojen luokkien kirjoittaminen käsin on kauhean epämukavaa. Onneksi sinun ei tarvitse kirjoittaa niitä käsin. Niiden luo-

miseksi automaattisesti voidaan käyttää malleja. Tässä on malli, jolla luodaan tyyppiturvalliset pinojen rajapinnat käyttäen yksityistä periytyvyyttä:

```
template<class T>
class Stack: private GenericStack {
public:
  void push(T *objectPtr) { GenericStack::push(objectPtr); }
  T * pop() { return static_cast<T*>(GenericStack::pop()); }
  bool empty() const { return GenericStack::empty(); }
};
```

Tämä on hämmästyttävää koodia, vaikka et sitä ehkä heti käsitäkään. Kääntäjät luovat mallin avulla automaattisesti niin monta rajapintaluokkaa kuin tarvitset. Koska nämä luokat ovat tyyppiturvallisia, asiakkaan tyyppivirheet tunnistetaan kääntämisen aikana. Koska `GenericStack`-luokan jäsenfunktiot ovat suojattuja ja rajapinnan luokat käyttävät luokkaa yksityisenä kantaluokkana, asiakkaat eivät pysty jättämään rajapinnan luokkia huomiotta. Koska rajapinnan luokan jokainen jäsenfunktio on esitelty (implisiittisesti) avoimena funktiona, suorituksen aikaista kustannusta ei synny, kun käytetään tyyppiturvallisia luokkia; luotu koodi on täsmälleen samaa, kuin jos asiakkaat ohjelmoisivat `GenericStack`-luokan suoraan (olettaen, että kääntäjät kunnioittavat `inline`-pyyntöä - katso Kohta 33). Ja koska `GenericStack` käyttää `void*`-tyyppisiä osoittimia, maksat vain yhdestä kopiosta koodia, jolla käsitellään pinoja, riippumatta siitä, kuinka monen tyyppistä pinoa käytät ohjelmassasi. Lyhyesti sanottuna, saat tämän työtavan avulla koodin, joka on sekä maksimaalisen tehokas että maksimaalisen tyyppiturvallinen. On vaikeaa pistää paremmaksi kuin tämä.

Yksi tämän kirjan ohjelauseista on se, että C++-kielen piirteet vaikuttavat toisiinsa merkittävillä tavoilla. Toivon, että olet samaa mieltä, että tämä esimerkki on aika merkittävä.

Tästä esimerkistä voidaan johtaa se oivallus, että sitä ei olisi voitu saada aikaiseksi kerrosajattelun avulla. Vain periytyvyys antaa pääsyn suojattuihin jäseniin, ja vain periytyvyys sallii sen, että virtuaalifunktiot määritetään uudelleen. (Esimerkki siitä, kuinka virtuaalifunktioiden olemassaolo voi motivoida yksityisen periytyvyyden käytön, löytyy Kohdasta 43.) Koska virtuaalifunktiot ja suojatut jäsenet ovat olemassa, yksityinen periytyvyys on joskus ainoa käytännöllinen tapa ilmaista *on toteutettu ehdoilla* -suhde luokkien välillä. Tästä on tuloksena, että sinun ei kannata pelätä käyttää yksityistä periytyvyyttä silloin, kun se on sopivin käytettävissä oleva toteutustekniikka. Kerrosajattelu on kuitenkin samaan aikaan yleisesti ottaen suositeltavin tekniikka, joten sinun kannattaa ottaa se käyttöön aina kun voit.

## Kohta 43: Käytä moniperintää ymmärtäväisesti.

Riippuen siitä kuka puhuu, moniperintä (MI) on joko jumalaisen inspiraation tai pirun manifestityön tuote.

Ne, jotka kannattavat sitä, ylistävät sitä olennaisen luonnollisena mallina tosielämän ongelmiin, kun taas arvostelijat ovat yhtä mieltä siitä, että se on hidas, vaikea toteuttaa, eikä yhtään tehokkaampi kuin yksittäinen periytyvyys. Hämmentävää kyllä, oliosuuntautuneiden ohjelmointikielten maailma on yhä jakautuneena tästä asiasta: C++, Eiffel ja Common LISP Object System (CLOS) sisältävät moniperinnän; Smalltalk, Objective C ja Object Pascal eivät sisällä; ja Java tukee vain rajoittunutta muotoa siitä. Mihin köyhän ja ponnistelevan ohjelmoijan kannattaa uskoa?

Ennen kuin uskot mitään, sinun pitää selvittää muutama asia. Yksi kiistämätön tosiasia C++-kielen moniperinnästä (MI) on, että se avaa Pandoran lippaan monimutkaisuudet, joita ei yksinkertaisesti ole olemassa yksittäisen periytyvyyden alla. Perustavin näistä monimutkaisista asioista on tietenkin monimerkityksisyys (katso Kohta 26). Jos periytetty luokka perii jäsenen nimen useammasta kuin yhdestä kantaluokasta, mikä tahansa viittaus tuohon nimeen on monimerkityksellistä; sinun täytyy implisiittisesti kertoa, mitä jäsentä tarkoitat. Tässä on esimerkki, joka perustuu ARM:ssa käytyyn keskusteluun (katso Kohta 50):

```
class Lottery {
public:
  virtual int draw();

  ...

};

class GraphicalObject {
public:
  virtual int draw();

  ...

};

class LotterySimulation: public Lottery,
                         public GraphicalObject {

  ...                           // ei esittele draw-funkt.

};

LotterySimulation *pls = new LotterySimulation;

pls->draw();                    // virhe! – monimerkityks.
pls->Lottery::draw();           // toimii
pls->GraphicalObject::draw();   // toimii
```

Tämä näyttää kömpelöltä, mutta ainakin se toimii. Kömpelyyttä on valitettavasti vaikea eliminoida. Vaikka yksi periytetyistä `draw`-funktioista olisi `private`-tyyppinen ja täten luoksepääsemätön, monimerkityksisyys säilyisi. (Tähän on hyvä syy, mutta täydellinen selitys tilanteesta on luettavissa Kohdasta 26, joten en toista sitä tässä.)

Jäsenten kelpuuttaminen eksplisiittisesti on kuitenkin enemmän kuin kömpelöä ja se on lisäksi rajoittavaa. Kun jäsen kelpuutetaan eksplisiittisesti luokan nimellä, funktio ei toimi enää virtuaalisena. Kutsuttu funktio on sen sijaan tarkalleen sama kuin se, jonka määrität, vaikka olio, jonka kohdalla sitä pyydetään avuksi, on periytetystä luokasta:

```
class SpecialLotterySimulation: public LotterySimulation {
public:
  virtual int draw();

  ...

};

pls = new SpecialLotterySimulation;

pls->draw();                    // virhe! – yhä monimerkit.
pls->Lottery::draw();           // kutsuu Lottery::draw
pls->GraphicalObject::draw();   // kuts. GraphicalObject::draw
```

Huomaa tässä tapauksessa, että vaikka `pls` osoittaa `SpecialLotterySimulation`-olioon, tuossa luokassa määritettyä `draw`-funktiota ei ole mitään tapaa (muunnos alaspäin puuttuu - katso Kohta 39) pyytää avuksi.

Mutta odota, lisää seuraa. Sekä `Lottery-` että `GraphicalObject`-luokissa olevat `draw`-funktiot on esitelty virtuaalifunktioina niin, että aliluokat voivat määrittää ne uudelleen (katso Kohta 36), mutta mitä jos `LotterySimulation` haluaisi määrittää ne molemmat uudelleen? Epämiellyttävä totuus on, että se ei voi, koska luokalla sallitaan olevan vain yksi `draw`-niminen funktio, joka vastaanottaa argumentteja. (Tähän sääntöön on olemassa erikoispoikkeus, jos yksi funktioista on tyypiltään `const` ja toinen ei ole - katso Kohta 21.)

Tämän hankaluuden ajateltiin yhdessä vaiheessa olevan niin vakava asia, että se oikeuttaisi muutokseen kielessä. ARM:ssa keskustellaan mahdollisuudesta, jolla sallittaisiin perittyjen virtuaalifunktioiden "uudelleennimeäminen", mutta sitten huomattiin, että ongelma voidaan kiertää lisäämällä pari uutta luokkaa:

```
class AuxLottery: public Lottery {
public:
  virtual int lotteryDraw() = 0;

  virtual int draw() { return lotteryDraw(); }
};

class AuxGraphicalObject: public GraphicalObject {
public:
  virtual int graphicalObjectDraw() = 0;

  virtual int draw() { return graphicalObjectDraw(); }
};
```

```
class LotterySimulation: public AuxLottery,
                         public AuxGraphicalObject {
public:
  virtual int lotteryDraw();
  virtual int graphicalObjectDraw();

  ...
};
```

Nämä molemmat kaksi uuttaa luokkaa `AuxLottery` ja `AuxGraphicalObject` esittelevät pääasiallisesti uuden nimen `draw`-funktiolle, jonka molemmat perivät. Tämä uusi nimi on puhtaan virtuaalifunktion muotoinen, tässä tapauksessa `lotteryDraw` ja `graphicalObjectDraw`; funktiot ovat puhtaasti virtuaalisia niin, että konkreettisten aliluokkien täytyy määrittää ne uudelleen. Jokainen luokka määrittää lisäksi sen `draw`-funktion uudelleen, jonka se perii pyytääkseen itselleen avuksi uutta, puhdasta virtuaalifunktiota. Verkkovaikutus on, että yksinäinen, monimerkityksinen nimi `draw` on tämän luokkahierarkian sisällä jaettu tehokkaasti kahteen monimerkityksettömään, mutta toiminnallisesti samaa merkitsevään nimeen: `lotteryDraw` ja `graphicalObjectDraw`:

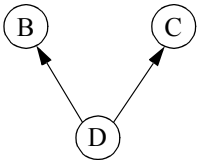
```
LotterySimulation *pls = new LotterySimulation;

Lottery *pl = pls;
GraphicalObject *pgo = pls;

// tämä kutsuu LotterySimulation::lotteryDraw
pl->draw();

// tämä kutsuu LotterySimulation::graphicalObjectDraw
pgo->draw();
```

Tämä strategia, joka on valmis ja jossa on sovellettu fiksusti puhtaita virtuaalifunktioita, yksinkertaisia virtuaalifunktiota ja avoimia funktioita (katso Kohta 33), jotka pitäisi opetella ulkoa. Se ratkaisee ensinnäkin ongelman, jonka voit kohdata jonain päivänä. Toiseksi se voi toimia muistuttajana sinulle komplikaatioista, jotka voivat nousta esiin silloin, kun käytetään moniperintää. Kyllä, tämä taktiikka toimii, mutta haluatko todella, että sinut pakotetaan esittelemään uusia luokkia vain siksi, että voit määrittää uudelleen virtuaalifunktion? Luokat `AuxLottery` ja `AuxGraphicalObject` ovat elintärkeitä tämän hierarkian oikealle toiminnalle, mutta ne eivät vastaa ongelma-alueen abstraktioon eivätkä toteutusalueen abstraktioon. Ne ovat olemassa puhtaasti toteutuksen välineenä - ei mitään muuta. Tiedät jo, että hyvä ohjelmisto on "laiteriippumaton". Tämä peukalosääntö pätee myös tässä.

Monimerkityksisyysongelma, niin kiinnostava kuin se onkin, tuskin raapaisee pinnalta niitä asioita, joita kohtaat, kun keimailet moniperinnän (MI) kanssa. Toinen ongelma tulee esiin siitä empiirisestä huomiosta, että periytyvyyshierarkialla, joka näyttää

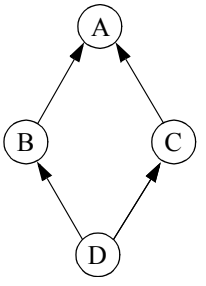
aluksi tältä,

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




on ahdistava taipumus kehkeytyä tämän näköiseksi:

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

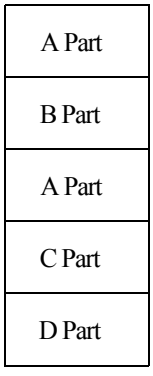



Totta tai ei, että timantit ovat tytön paras ystävä, mutta nyt on varmasti totta, että timantinmuotoinen periytyvyyshierarkia, kuten tämä, *ei* ole kovin ystävällinen. Jos luot tämänkaltaisen hierarkian, joudut heti vastakkain sen kysymyksen kanssa, tehdäänkö A:sta virtuaalinen kantaluokka, toisin sanoen, pitäisikö A:n periytyvyyden olla virtuaalista. Vastaus on käytännössä lähes poikkeuksetta, että sen pitäisi olla virtuaalista; vain harvoin haluat, että olio, joka on tyyppiä D, sisältäisi monia kopioita A:n tietojäsenistä. Edellä mainitut B ja X esittelevät tunnistaessaan tämän totuuden A:n virtuaalisena kantaluokkana.

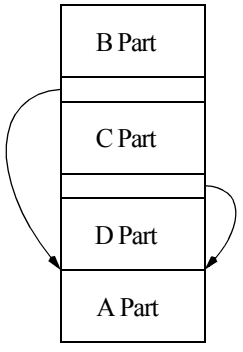
Valitettavasti samaan aikaan, kun määrität B:n ja C:n, et ehkä tiedä, pitäisikö minkään luokan päättää periytyä molemmista niistä, ja itse asiassa, sinun ei pitäisikään tietää tätä siinä järjestyksessä, että määrittäisit ne uudelleen. Joudut luokkien suunnittelijana kauheaan hämmennyksen tilaan. Jos *et* esittele A:ta B:n ja C:n virtuaalisena kantana, D:n myöhemmän suunnittelijan täytyy ehkä muuttaa B:n ja C:n määrityksiä, jotta pystyisi käyttämään niitä tehokkaasti. Tätä on poikkeuksetta mahdotonta hyväksyä, usein siksi, koska A:n, B:n ja C:n määritykset sisältävät vain lukuoikeudet. Näin olisi tapaus silloin, jos A, B ja C olisivat esimerkiksi kirjastossa, ja D olisi kirjaston asiakkaan kirjoittama.

Jos toisaalta *esittelet* A:n B:n ja C:n virtuaalisena kantana, tyrkytät tyypillisesti lisäkustannuksen, joka koskee näiden luokkien asiakkaiden tilaa ja aikaa. Tämä johtuu siitä, että virtuaaliset kantaluokat on usein toteutettu osoittimina olioihin, ennemminkin kuin itse olioina. On tarpeetonta sanoa, että muistissa olevien olioiden pohjakaava on kääntäjästä riippuvainen. Tosiasia on, että olion, joka on tyyppiä D A:n ollessa epävirtuaalisena kantana, muistin pohjakaava on tyypillisesti jatkuva sarja muistipaikkoja, siinä missä olion, joka on tyyppiä D, A:n ollessa virtuaalisena kantana,

muistin pohjakaava on joskus jatkuva sarja muistipaikkoja. Nämä sisältävät osoittimet muistipaikkoihin, joissa on virtuaalisen kantaluokan tietojäsenet:




Muistin yleinen pohjakaava D-oliolle, jossa A on epävirtuaalinen kantaluokka




Eräiden kääntäjien muistin pohjakaava D-oliosta, jossa A on virtuaalinen kantaluokka

Myös ne kääntäjät, jotka eivät käytä tätä määrättyä toteutusstrategiaa, tyrkyttävät tyypillisesti eräänlaista tilarangaistusta, kun käytetään virtuaalista periytyvyyttä.

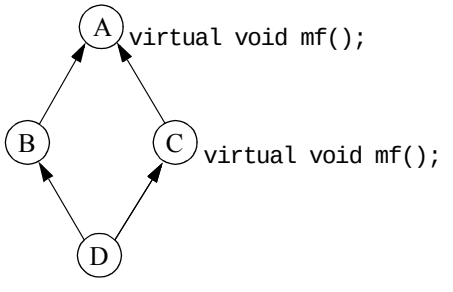
Näiden harkintojen valossa näyttäisi siltä, että tehokas luokkasuunnittelu, silloin kun käytetään moniperintää (MI), vaatii kirjastojen suunnittelijoilta selvänäköisyyttä. Kun nähdään, että maalaisjärjen käyttö on yhä harvinaisempaa näinä päivinä, olisi harhaanjohtamista, jos nojautuisit liian raskaasti kielen piirteeseen, joka vaatii suunnittelijoilta ei pelkästään sen, että he ennakoisivat tulevaisuuden tarpeet, vaan myös suorastaan profeetallisuutta.

Sama asia voitaisiin tietysti sanoa valinnasta virtuaali- ja epävirtuaalifunktioiden välillä kantaluokassa, mutta niiden välillä on kriittinen ero. Kohdassa 36 selvitetään, että virtuaalifunktiolla on hyvin määritetty korkean tason merkitys, joka erottuu selvästi vastaavasti epävirtuaalifunktion hyvin määritetystä korkean tason merkityksestä, joten on mahdollista valita näiden kahden väliltä, jotka perustuvat siihen, miten haluat kommunikoida aliluokkien kirjoittajien välillä. Päätöksestä, pitäisikö kantaluokan olla virtuaalinen vai epävirtuaalinen, puuttuu kuitenkin hyvin määritetty korkean tason tarkoitus. Tuo päätös perustuu tavallisesti ennemminkin koko periytyvyyshierarkian rakenteeseen, ja sellaisena sitä ei voida tehdä, ennen kuin koko hierarkia tunnetaan. Jos sinun tarvitsee tietää tarkasti, kuinka luokkaasi tullaan käyttämään, ennen kuin voit määrittää sen oikein, tehokkaiden luokkien suunnitteleminen muodostuu erittäin vaikeaksi.

Kun olet selvittänyt monimerkityksisyyden ongelman ja kysymyksen, pitäisikö kantaluokkasi/-luokkiesi periytyvyyden olla virtuaalista, vastaan tulee yhä enemmän komp-

likaatioita. Sen sijaan, että heittäisin lisää löylyä kiukaalle, mainitsen vain kaksi muuta kysymystä, jotka kannattaa muistaa:

- **Muodostinargumenttien välittäminen virtuaalisille kantaluokille.** Kantaluokan muodostinfunktion argumentit määritetään epävirtuaalisen periytyvyyden alaisuudessa niiden luokkien jäsenten alustuslistassa, jotka ovat välittömästi periytyneet kantaluokasta. Koska yksittäiset periytyvyyshierarkiat tarvitsevat vain epävirtuaalisia kantoja, argumentit välitetään periytyvyyshierarkiaan erittäin luonnollisella tavalla: hierarkian *n*-tasolla olevat luokat välittävät argumentit luokille tasolla *n*-1. Argumentit on kuitenkin määritetty virtuaalisen kantaluokan muodostinfunktioille niiden luokkien alustuslistassa, jotka on viimeksi *periytetty* kannasta. Tästä on tuloksena, että luokka, joka alustaa virtuaalisen kannan ja voi omavaltaisesti olla kaukana siitä periytyvyyskaaviosta, sekä luokka, joka suorittaa alustuksen, voi muuttua, kun uusia luokkia lisätään hierarkiaan. (Tämä ongelma voidaan välttää erinomaisesti eliminoimalla tarve välittää muodostinargumentteja virtuaalikantoihin. Helpoin tapa tehdä tämä on välttää tietojäsenten sijoittamista tällaisiin luokkiin. Tämä on merkittävin osa Javan ratkaisussa ongelmaan: Javan virtuaalisia kantaluokkia (eli "Interfaces") on kielletty sisältämästä tietoa.)
- **Virtuaalifunktioiden valta-asema.** Juuri kun luulit, että olit selvittänyt kaiken monimerkityksisyyden, sinua koskevat säännöt muutetaan. Tutki uudelleen timantin muotoista periytyvyyskaaviota, joka koskee luokkia `A`, `B`, `C` ja `D`. Oletetaan, että `A` määrittää virtuaalisen jäsenfunktion `mf`, ja `C` määrittää sen uudelleen; `B` ja `D` eivät kuitenkaan määritä `mf`-funktiota uudelleen:




Aikaisemman keskustelun pohjalta odottaisit tämän olevan monimerkityksistä:

```
D *pd = new D;
pd->mf();                    // A::mf or C::mf?
```

Mitä `mf`-funktiota pitäisi kutsua `D`-oliolle, sitä, joka periytyy suoraan `C`:stä vai sitä, joka epäsuorasti periytyi (`B`:n kautta) `A`:sta? Vastaus riippuu siitä, kuinka `B` ja `C` periytyvät `A`:sta. Varsinkin, jos `A` on `B`:n tai `C`:n epävirtuaalinen kanta, kutsu on monimerkityksinen, mutta jos `A` on sekä `B`:n että `C`:n virtuaalinen kanta, `mf`-funktion uudelleenmäärityksen `C`:ssä sanotaan *dominoivan* `A`:n alkuperäisen

määrityksen `A`:ssa, ja kutsu `mf`-funktioon `pd`:n kautta johtaa (monimerkityksettömästi) `C::mf`-funktioon. Jos istut alas ja selvität tämän kaiken, käy ilmi, että tämä on haluamasi käyttäytyminen, mutta on aika tuskallista istua alas ja selvittää tämä kaikki järkevästi.

Olet jo varmasti samaa mieltä, että moniperintä (MI) voi johtaa komplikaatioihin. Ehkä olet vakuuttunut siitä, että kukaan täysijärkinen ei käytä sitä koskaan. Ehkä olet valmistautunut ehdottamaan kansainväliselle C++-kielen standardointikomitealle, että moniperintä poistettaisiin kielestä, tai ainakin ehdottamaan johtajallesi, että yrityksesi ohjelmoijia kiellettäisiin fyysisesti käyttämästä sitä.

Ehkä olet liian hätiköivä.

Kannattaa muistaa, että vaikka C++-kielen suunnittelija ei aikonut tehdä moniperinnästä vaikeaa käyttää, ilmeni vain se, että yritys saada kaikki palat toimimaan yhdessä enemmän tai vähemmän järkevällä tavalla on luonnostaan perinyt eräiden monimutkaisuuksien esittelyn. Olet ehkä edellä olevasta keskustelusta huomannut, että osa näistä monimutkaisuuksista ilmenee siinä yhteydessä, kun käytetään virtuaalisia kantaluokkia. Jos voit välttää virtuaalikantojen käyttämistä - tämä tarkoittaa sitä, että jos voit välttää kuolettavan timanttisen periytyvyyskaavion luonnin - asioista tulee paljon hallittavimpia.

Kohdassa 34 kuvataan, kuinka esimerkiksi *Protocol-luokka* on olemassa vain määrittääkseen rajapinnan periytetyille luokille; sillä ei ole tietojäseniä, ei muodostinfunktioita, virtuaalituhoajafunktiota (katso Kohta 14) eikä joukkoa puhtaita virtuaalifunktioita, jotka määrittävät rajapinnan. Protocol-tyyppinen `Person`-luokka voisi näyttää tältä:

```
class Person {
public:
  virtual ~Person();

  virtual string name() const = 0;
  virtual string birthDate() const = 0;
  virtual string address() const = 0;
  virtual string nationality() const = 0;
};
```

Tämän luokan asiakkaiden täytyy ohjelmoida `Person`-luokan osoittimien ja viittausten ehdoilla, koska abstrakteja luokkia ei voida instantioida.

Jos halutaan luoda olioita, joita voidaan käsitellä `Person`-olioina, `Person`-luokan asiakkaiden täytyy käyttää *tehdasvalmisteisia funktioita* (katso Kohta 34) alustaakseen tuon luokan konkreettiset aliluokat:

```
// tehdasfunktio, jolla luodaan Person-olio
// tietokannan yksilöllisestä ID-tunnuksesta
Person * makePerson(DatabaseID personIdentifier);
```

```
DatabaseID askUserForDatabaseID();

DatabaseID pid = askUserForDatabaseID();
Person *pp = makePerson(pid);  // luo olio, joka tukee
                               // Person-rajapintaa
...                            // manipuloi *pp
                               // Personin jäsenfunktion
                               // kautta
delete pp;                     // poista olio sitten, kun
                               // sitä ei enää tarvita
```

Tämä suorastaan kerjää kysymystä: kuinka `makePerson` luo olioita, joihin se palauttaa osoittimia? Täytyy selvästi olla olemassa jokin konkreettinen luokka, joka on periytynyt `Person`-luokasta, jonka `makePerson` voi instantioida.

Oletetaan, että tämän luokan nimi on `MyPerson`. `MyPerson`-luokan täytyy konkreettisena luokkana tarjota toteutukset puhtaille virtuaalifunktioille, jotka se perii `Person`-luokasta. Se voisi kirjoittaa ne tyhjästä, mutta olisi parempaa ohjelmistosuunnittelua hyödyntää olemassaolevia komponentteja, jotka jo tekevät useimmat tai kaikki tarvittavat asiat. Oletetaan esimerkiksi, että vanhuuttaan nariseva tietokantaspesifinen `PersonInfo` on jo olemassa, tarjoten sen tärkeimmän, mitä `MyPerson` tarvitsee:

```
class PersonInfo {
public:
  PersonInfo(DatabaseID pid);
  virtual ~PersonInfo();

  virtual const char * theName() const;
  virtual const char * theBirthDate() const;
  virtual const char * theAddress() const;
  virtual const char * theNationality() const;

  virtual const char * valueDelimOpen() const;     // katso
  virtual const char * valueDelimClose() const;    // ed.

  ...
};
```

Voit huomata, että tämä on vanha luokka, koska jäsenfunktiot palauttavat `const char*` -tyyppiset muuttujat merkkijono-olioiden sijasta. Jos kenkä kuitenkin sopii, niin miksi et käyttäisi sitä? Tämän luokan jäsenfunktioiden nimet viittaavat siihen, että tulos tulee olemaan aika mukava.

Tulet huomaamaan, että `PersonInfo` on kuitenkin suunniteltu helpottamaan prosessia, jossa tietokannan kentät tulostetaan eri muodoissa, ja jokaisen kentän arvon loppu on erotettu erityisillä merkkijonoilla. Oletuksena on, että kentän arvojen avaavat

ja sulkevat erottimet ovat hakasulkeet, joten kenttänimi "Kippurahäntäinen puoliapina" muotoiltaisiin tähän tapaan:

```
[Kippurahäntäinen puoliapina]
```

Kun tunnistetaan se tosiasia, että `PersonInfo`-luokan asiakkaat eivät yleisesti ottaen halua käyttää hakasulkeita, virtuaalifunktiot `valueDelimOpen` ja `valueDelimClose` sallivat periytettyjen luokkien määrittää omat avaavat ja sulkevat erotinmerkkien merkkijonot. `PersonInfo`-luokan `theName`, `theBirthDate`, `theAddress`- ja `theNationality`-muuttujien toteutukset kutsuvat näitä virtuaalifunktioita lisätäkseen sopivat erottimet palauttamiinsa arvoihin. Kun `PersonInfo::name`-funktio on esimerkkinä, koodi näyttää tältä:

```
const char * PersonInfo::valueDelimOpen() const
{
  return "[";                  // olet. ol. avaava erotin
}

const char * PersonInfo::valueDelimClose() const
{
  return "]";                  // oletuksena oleva
                               // sulkeva erotinmerkki
}

const char * PersonInfo::theName() const
{
  // varaa puskurista paluuarvolle. Koska tämä on tyypil-
  // tään static, se alustetaan automaattisesti nollilla
  static char value[MAX_FORMATTED_FIELD_VALUE_LENGTH];

  // kirjoita avaava erotinmerkki
  strcpy(value, valueDelimOpen());

  lisää value-muuttujassa olevaan merkkijonoon tämän olion
nimikenttä

  // kirjoita sulkeva erotinmerkki
  strcat(value, valueDelimClose());

  return value;
}
```

`PersonInfo::theName`-funktion työtavasta voitaisiin kinastella (varsinkin kiinteän kokoisen staattisen puskurin käytöstä - katso Kohta 23), mutta laita hiusten halkomiset syrjään ja keskity sen sijaan tähän: `theName` kutsuu `valueDelimOpen`-funktiota luodakseen avaavan erotinmerkin merkkijonoon, jonka se palauttaa, sitten se luo itse nimen arvon, kutsuu sitten `valueDelimClose`-funktiota. Koska `valueDelimOpen` ja `valueDelimClose` ovat virtuaalifunktioita, `theName`:n palauttama tulos ei ole pelkästään riippuvainen `PersonInfo`-luokasta, vaan myös `PersonInfo`-luokasta periytetyistä luokista.

Nämä ovat hyviä uutisia `MyPerson`-luokan toteuttajalle, koska jos käytät hämäriä ilmauksia `Person`-luokan dokumentaatiossa, huomaat, että `name`:n ja sen sisar-

jäsenfunktioiden arvot vaaditaan palauttamaan koristelemattomina arvoina, eli erotinmerkkejä ei sallita. Tämä tarkoittaa sitä, että jos henkilö on kotoisin Madagaskarista, kutsu tämän henkilön `nationality`-funktioon palauttaisi "Madagaskar", ei "[Madagaskar]".

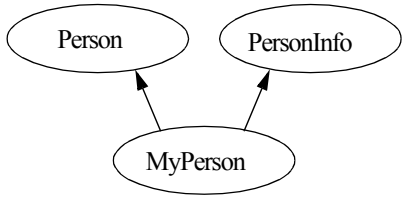
`MyPerson`- ja `PersonInfo`-luokkien välinen suhde on se, että `PersonInfo`-luokalla sattuu olemaan funktioita, jotka tekevät `MyPerson`-luokasta helpomman toteuttaa. Siinä kaikki. Näkyvillä ei ole minkäänlaista *on eräänlainen*- tai *omistaa*-suhdetta. Niiden suhde on täten *toteutettu jonkin ehdoilla* ja tiedämme, että se voidaan esittää kahdella tavalla: kerrosajattelun avulla (layering) (katso Kohta 40) tai yksityisen periytyvyyden avulla (katso Kohta 42). Kohta 42 osoittaa, että kerrosajattelu on yleisesti ottaen suositeltavampi työtapa, mutta yksityinen periytyvyys on välttämätön, jos virtuaalifunktiot tullaan määrittelemään uudelleen. `MyPerson`-luokan tarvitsee tässä tapauksessa määrittää uudelleen `valueDelimOpen` ja `valueDelimClose` -funktiot, joten kerrosajattelu ei käy ja yksityistä periytyvyyttä pitää käyttää: `MyPerson`-luokan täytyy periytyä yksityisesti `PersonInfo`-luokasta.

Mutta `MyPerson`-luokan täytyy myös toteuttaa `Person`-luokan rajapinta, ja tässä vaaditaan julkista periytyvyyttä. Tämä johtaa yhteen moniperinnän järkevään sovellustapaan: yhdistä rajapinnan julkinen periytyvyys toteutuksen yksityiseen periytyvyyteen:

```
class Person {                  // tämä luokka määrittelee
public:                         // toteutettavan
  virtual ~Person();            // rajapinnan

  virtual string name() const = 0;
  virtual string birthDate() const = 0;
  virtual string address() const = 0;
  virtual string nationality() const = 0;
};

class DatabaseID { ... };       // käytetään edellä; yksi-
                                // tyiskohdat eivät tärk.

class PersonInfo {              // tällä luokalla on funkt.
public:                         // jotka ovat käytännöll.
  PersonInfo(DatabaseID pid);   // toteut. Personin rajap.
  virtual ~PersonInfo();

  virtual const char * theName() const;
  virtual const char * theBirthDate() const;
  virtual const char * theAddress() const;
  virtual const char * theNationality() const;

  virtual const char * valueDelimOpen() const;
  virtual const char * valueDelimClose() const;

  ...

};
```

```
class MyPerson: public Person,        // huomaa moniperin-
                private PersonInfo {  // nän käyttö
public:
  MyPerson(DatabaseID pid): PersonInfo(pid) {}

  // per. virt. erotinmerkkifunktioid. uudelleenmääritykset
  const char * valueDelimOpen() const { return ""; }
  const char * valueDelimClose() const { return ""; }

  // Person-luokan vaadittujen jäsenfunktioiden
  // toteutukset
  string name() const
  { return PersonInfo::theName(); }

  string birthDate() const
  { return PersonInfo::theBirthDate(); }

  string address() const
  { return PersonInfo::theAddress(); }

  string nationality() const
  { return PersonInfo::theNationality(); }
};
```

Graafisesti tämä näyttää tältä:




Tämän tyyppinen esimerkki esittää, että MI voi olla sekä käytännöllinen että tajuttava, vaikka ei olekaan mikään yllätys, että pelätty timantin muotoinen periytyvyyskaavio on silmäänpistävästi poissa.

Yritä kuitenkin välttää houkutusta. Voit joskus langeta siihen ansaan, että käytät moniperintää (MI) tehdäksesi nopean korjauksen periytyvyyshierarkiaan, joka olisi toiminut paremmin perinteisemmällä uudelleensuunnittelulla. Oletetaan esimerkiksi, että työskentelet hierarkian kanssa, jolla animoidaan sarjakuvahenkilöitä. Ainakin käsitteellisesti on järkevää, että kuka tahansa henkilö tanssii ja laulaa, mutta tapa, jolla jokaisen tyypin mukainen henkilö suorittaa nämä aktiviteetit, vaihtelee. Laulamisen ja tanssimisen oletuskäytös on lisäksi ei tehdä mitään.

Tämä kaikki voidaan kertoa C++-kielellä tähän tyyliin:

```
class CartoonCharacter {
public:
  virtual void dance() {}
  virtual void sing() {}
};
```

Virtuaalifunktiot mallintavat luonnollisesti rajoituksen, että tanssiminen ja laulaminen ovat järkeviä toimintoja kaikille `CartoonCharacter`-olioille. Älä-tee-mitään-tyyppinen oletuskäyttäytyminen ilmaistaan luokassa tyhjillä määrityksillä (katso Kohta 36).

Oletetaan, että heinäsirkka on määrätyn tyyppinen sarjakuvahahmo, joka tanssii ja laulaa omalla määrätyllä tavallaan:

```
class Grasshopper: public CartoonCharacter {
public:
  virtual void dance();         // määritys on muualla
  virtual void sing();          // määritys on muualla
};
```

Oletetaan seuraavaksi, että kun olet toteuttanut `Grasshopper`-luokan, päätät, että tarvitset myös luokan sirkoille:

```
class Cricket: public CartoonCharacter {
public:
  virtual void dance();
  virtual void sing();
};
```

Kun ryhdyt toteuttamaan `Cricket`-luokkaa, tajuat, että voit käyttää uudelleen paljoa `Grasshopper`-luokalle kirjoittamaasi koodia. Sitä pitää kuitenkin hieman säätää sieltä sun täältä, ottaen huomioon heinäsirkkojen ja sirkkojen välillä olevat erot laulamisessa ja tanssimisessa. Sinulle juolahtaa yhtäkkiä mieleen älykäs tapa, jolla voit käyttää uudelleen koodiasi: toteutat `Cricket`-luokan `Grasshopper`-luokan ehdoilla, ja käytät virtuaalifunktioita, joilla sallit `Cricket`-luokan muokata `Grasshopper`-luokan käyttäytymistä!

Huomaat heti, että nämä kaksoisvaatimukset - *on toteutettu jonkin ehdoilla* -suhde ja kyky määrittää virtuaalifunktiot uudelleen - tarkoittavat sitä, että `Cricket`-luokan on periydyttävä yksityisesti `Grasshopper`-luokasta, mutta sirkka on tietysti silti sarjakuvahahmo, joten määrität `Cricket`-luokan uudelleen niin, että se periytyy sekä `Grasshopper`-luokasta että `CartoonCharacter`-luokasta:

```
class Cricket: public CartoonCharacter,
               private Grasshopper {
public:
  virtual void dance();
  virtual void sing();
};
```

Valmistaudut sitten tekemään tarvittavat muunnokset `Grasshopper`-luokkaan. On tärkeää, että esittelet joitakin uusia virtuaalifunktioita, jotka `Cricket` voi määrittää uudelleen:

```
class Grasshopper: public CartoonCharacter {
public:
  virtual void dance();
  virtual void sing();

protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();

  virtual void singCustomization();
};
```

Heinäsirkkojen tanssiminen on nyt määritetty näin:

```
void Grasshopper::dance()
{
  suorita tanssin yleiset toiminnot;

  danceCustomization1();

  suorita lisää tanssin yleisiä toimintoja;

  danceCustomization2();

  suorita loput tanssin yleiset toiminnot;
}
```

Heinäsirkan laulaminen on samalla tavoin orkesteroitu.

Selvää on, että `Cricket`-luokka täytyy päivittää niin, että se ottaa huomioon uudet virtuaalifunktiot, jotka sen pitää määrittää uudelleen:

```
class Cricket: public CartoonCharacter,
               private Grasshopper {
public:
  virtual void dance() { Grasshopper::dance(); }
  virtual void sing() { Grasshopper::sing(); }

protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();

  virtual void singCustomization();
};
```

Tämä tuntuu toimivan hienosti. Kun `Cricket`-oliota käsketään tanssimaan, se suorittaa `Grasshopper`-luokassa olevan yleisen tanssikoodin, sitten se suorittaa `Cricket`-luokassa olevan tanssin asiakaskohtaisen koodin, jatkaa sitten suorittaen `Grasshopper::dance`-funktion koodin ja niin edelleen.

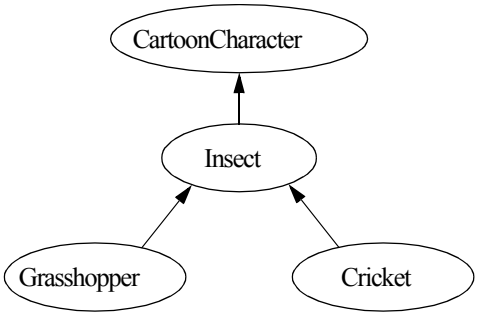
Suunnittelussasi on kuitenkin vakava kauneusvirhe, ja se on se, että olet juossut päättömästi Okkamilaisen partaterään, mikä on tietysti huono ajatus minkä tahansa partaterän kanssa, ja varsinkin silloin, kun se kuuluu Vilhelm Okkamilaiselle. Okkaismi julistaa, että olevaista ei pitäisi kertoa yli tarpeen, ja tässä tapauksessa kysymyksessä

olevat olevaiset ovat periytyvyyden suhteet. Jos uskot, että moniperintä on monimutkaisempi kuin yksittäinen periytyvyys (ja toivon että uskot niin), `Cricket`-luokan suunnittelu on sitten tarpeettoman monimutkainen.

Ongelma on pohjimmiltaan siinä, että ei ole totta, että `Cricket`-luokka on toteutettu `Grasshopper`-luokan ehdoilla. `Cricket`-luokka ja `Grasshopper`-luokka pikemminkin jakavat yhteisen koodin. Ne jakavat varsinkin sen koodin, joka määrittää sen tanssimisen ja laulamisen käyttäytymisen, joka on yhteinen sekä heinäsirkoille että sirkoille.

Tapa, jolla kerrotaan, että kahdella luokalla on jotain yhteistä, ei ole se, että yksi luokka periytyy toisesta, vaan se, että ne molemmat periytyvät samasta kantaluokasta. Heinäsirkkojen ja sirkkojen yhteinen koodi ei kuulu `Grasshopper`-luokkaan, eikä se kuulu myöskään `Cricket`-luokkaan. Se kuuluu uuteen luokkaan, josta ne molemmat periytyvät, esimerkiksi `Insect`:

```
class CartoonCharacter { ... };

class Insect: public CartoonCharacter {
public:
  virtual void dance();         // yhteinen koodi sekä
  virtual void sing();          // heinäsirk. että sirkoille

protected:
  virtual void danceCustomization1() = 0;
  virtual void danceCustomization2() = 0;

  virtual void singCustomization() = 0;
};

class Grasshopper: public Insect {
protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();

  virtual void singCustomization();
};

class Cricket: public Insect {
protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();

  virtual void singCustomization();
};
```




Huomaa, kuinka paljon siistimpi tämä työtapa on. Mukana on vain yksittäistä periytyvyyttä, ja lisäksi vain *julkista* periytyvyyttä käytetään. `Grasshopper` ja `Cricket` määrittävät vain asiakaskohtaiset funktiot; ne perivät tanssin ja laulun funktiot muuttamattomina `Insect`-luokasta. Vilhelm Okkamilainen olisi ylpeä.

Vaikka tämä työtapa on siistimpi kuin se, jossa oli mukana moniperintä (MI), se voi aluksi tuntua hieman huonolaatuisemmalta. Verrattuna MI-työtapaan, tämä yksittäisperiytyvyys-arkkitehtuuri aiheuttaa aivan uuden luokan esittelyn, luokan, joka on tarpeeton, jos moniperintää käytetään. Miksi esitellä ylimääräinen luokka, jos sinun ei tarvitse?

Tämä johtaa sinut kasvotusten moniperinnän viettelevän luonteen kanssa. MI tuntuu pinnalta katsoen olevan helpoin kulkureitti. Se ei lisää uusia luokkia, ja vaikka se aiheuttaa joidenkin uusien virtuaalifunktioiden lisäämisen `Grasshopper`-luokkaan, nuo luokat olisi joka tapauksessa pitänyt lisätä jonnekin.

Kuvittele ohjelmoijaa, joka ylläpitää laajaa C++-luokkakirjastoa, eli kirjastoa, johon uusi luokka pitää lisätä, samaan tapaan kuin `Cricket`-luokka pitää lisätä olemassaolevaan `CartoonCharacter/Grasshopper`-hierarkiaan. Ohjelmoija tietää, että laaja asiakkaiden joukko käyttää olemassaolevaa hierarkiaa, joten mitä suurempi muutos kirjastoon, sitä suurempi häiriö asiakkaille. Ohjelmoija on päättänyt minimoida tuon kaltaisen häiriön. Kun ohjelmoija miettii vaihtoehtoja, hän tajuaa, että jos yksittäinen yksityisen periytyvyyden linkki `Grasshopper`-luokasta lisätään `Cricket`-luokkaan, hierarkiaan ei tarvita muita muutoksia. Ohjelmoija hymyilee ajatukselle, tyytyväisenä siihen tulevaisuudenkuvaan, että toiminnallisuuteen oli saatu laaja lisäys vain pienellä kustannuksella, joka johtuu monimutkaisuuden kasvamisesta.

Kuvittele seuraavaksi, että tuo ylläpitävä ohjelmoija olet sinä. Vältä viettelystä.

## Kohta 44: Sano mitä tarkoitat; ymmärrä mitä sanot.

Tämän osan periytyvyyttä ja oliopohjaista suunnittelua käsittelevässä esittelyssä korostin ymmärtämisen tärkeyttä siitä, mitä erilaiset olioperäiset konstruktiot *tarkoittavat*. Tämä tarkoittaa eri asiaa kuin se, että tunnet kielen säännöt. C++-kielen säännöt määräävät esimerkiksi, että jos luokka `D` periytyy julkisesti luokasta `B`, `D`-osoittimesta suoritetaan normaali konversio osoittimeen `B`; `B`-luokan julkiset jäsenfunktiot ovat periytyneet `D`-luokan julkisina jäsenfunktioina, ja niin edelleen. Tämä kaikki on totta, mutta on lähes hyödytöntä yrittää muuntaa työtapasi C++-kieleen. Sinun täytyy sen sijaan ymmärtää, että julkinen periytyvyys tarkoittaa samaa kuin *on eräänlainen*, ja että jos `D` periytyy julkisesti `B`:-luokasta, jokainen `D`-tyyppinen olio myös *on eräänlainen* `B`-tyypin olio. Täten, jos käytät *on eräänlainen* -testiä työskentelyssäsi, tiedät, että sinun pitää käyttää julkista periytyvyyttä.

Se, että sanot mitä tarkoitat, on vain puolet taistelusta. Kolikon toinen puoli on se, että ymmärrät mitä sanot, ja se on yhtä lailla tärkeää. On esimerkiksi vastuutonta, jollei suorastaan moraalitonta kuljeskella ympäriinsä esittelemässä epävirtuaalisia jäsenfunktioita tunnistamatta sitä, että kun teet niin, tyrkytät rajoitteita aliluokille. Kun esittelet epävirtuaalisen jäsenfunktion, se mitä todella sanot on, että funktio esittää erikoistumisen ohittavan vakion, ja olisi tuhoisaa, jos et tietäisi sitä.

Julkisen periytyvyyden ja *on eräänlainen* -testin samanarvoisuus ja epävirtuaalisten jäsenfunktioiden ja erikoistumisen ohittavan vakion käsitteen samanarvoisuus ovat esimerkkejä siitä, kuinka eräät C++-kielen rakenteet vastaavat suunnittelutason ajatuksiin. Alla olevassa listassa on yhteenveto tärkeimmistä kartoituksista.

- **Yleinen kantaluokka tarkoittaa yhteisiä erikoisuuksia**. Jos luokka `D1` ja luokka `D2` esittelevät molemmat luokan `B` kantana, `D1` ja `D2` perivät yleiset tietojäsenet ja/tai `B`:n yleiset jäsenfunktiot. Katso Kohta 43.
- **Julkinen periytyvyys tarkoittaa samaa kuin *on eräänlainen*.** Jos luokka `D` periytyy julkisesti luokasta `B`, jokainen `D`-tyypin olio on myös tyypin `B` olio, mutta ei päin vastoin. Katso Kohta 35.
- **Yksityinen periytyvyys tarkoittaa samaa kuin käsite *on toteutettu jonkin ehdoilla*.** Jos luokka `D` periytyy yksityisesti luokasta `B`, oliot, joiden tyyppi on `D`, on yksinkertaisesti toteutettu `B`-tyyppisten olioiden ehdoilla; olioiden, joiden tyypit ovat `B` ja `D`, välillä ei ole olemassa käsitteellistä suhdetta. Katso Kohta 42.
- **Kerrosajattelu tarkoittaa samaa kuin *on eräänlainen* ja *on toteutettu jonkin ehdoilla*.** Jos luokka `A` sisältää tietojäsenen, jonka tyyppi on `B`, `A`-tyyppisillä olioilla on joko `B`-tyyppinen komponentti tai ne on toteutettu `B`-tyyppisen olion ehdoilla. Katso Kohta 40.

Seuraavat kartoitukset pätevät vain silloin kun käytetään julkista periytyvyyttä:

- **Puhdas virtuaalifunktio tarkoittaa sitä, että vain funktion rajapinta on periytynyt.** Jos luokka `C` esittelee puhtaan virtuaalisen jäsenfunktion `mf`, `C`-luokan aliluokkien täytyy periä rajapinta `mf`-funktiolle, ja `C`-luokan konkreettisten aliluokkien täytyy tarjota omat toteutukset sille. Katso Kohta 36.
- **Yksinkertainen virtuaalifunktio tarkoittaa sitä, että sekä funktion rajapinta että oletustoteutus ovat periytyneet.** Jos luokka `C` esittelee yksinkertaisen (ei puhtaan) virtuaalifunktion `mf`, `C`:n aliluokkien täytyy periä rajapinta `mf`-funktiolle, ja ne voivat myös valitessaan periä oletustoteutuksen. Katso Kohta 36.
- **Epävirtuaalinen funktio tarkoittaa sitä, että sekä funktion rajapinta että pakollinen toteutus on peritty.** Jos luokka `C` esittelee epävirtuaalisen jäsenfunktion `mf`, `C`-luokan aliluokkien täytyy periä sekä rajapinta että sen toteutus `mf`-funktiolle. Tästä on seurauksena, että `mf` määrittää vakion `C`:n erikoistumisen ohi. Katso Kohta 36.