Olio-ohjelmoinnin toteuttaminen Fortran 90 -kielellä

Florian Berger

20.12.2002

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

Tiivistelmä

Olio-ohjelmointi on ohjelmointimenetelmä, jonka toteuttaminen ei välttämättä edellytä puhdasta oliokieltä. Tutkielmassani vertailen olio-ohjelmoinnin perusrakenteiden toteuttamista proseduraalisilla ohjelmointikielillä. Tämän jälkeen kuvaan olio-ohjelmoinnin toteuttamista hybridikielellä käyttäen esimerkkinä C++-kieltä. Lopuksi tutkin erityisesti Fortran 90 -kielen soveltuvuutta olio-ohjelmointiin. Tutkielmani perustuu alan kirjallisuuteen.

Avainsanat: Olio-ohjelmointi, proseduraaliset kielet, C++-kieli, Fortran 90 -kieli.

Sisältö

Liitteet:

1. ”CHenkilo”-luokan lopullinen toteutus Fortran 90:llä.
2. ”CHenkilo”-luokan lopullinen toteutus C++-kielellä.
3. ”COpiskelija”-luokan lopullinen toteutus Fortran 90:llä.
4. ”COpiskelija”-luokan lopullinen toteutus C++-kielellä.
5. ”COpettaja”-luokan lopullinen toteutus Fortran 90:llä.
6. ”COpettaja”-luokan lopullinen toteutus C++-kielellä.
7. ”CPoly”-luokan lopullinen toteutus Fortran 90:lla.
8. ”CPoly”-luokka automaattisesti luotuna Fortran 90:lle.

1. Johdanto

Olio-ohjelmointi on käytännöllinen ohjelmointimenetelmä suurten ja monimutkaisten, varsinkin todellista maailmaa mallintavien sovellusten luomiseen. Kuitenkin vielä nykyäänkin moni sovellus toteutetaan esimerkiksi Fortran- tai C-kielillä, eivätkä tekijät edes pidä välttämättömänä kielen vaihtamista modernimpaan kieleen. Kielen vaihtamisen kynnyksenä voi olla usein uuden kielen ja ohjelmointityylin opettelun vaikeus. Muun muassa tämän takia olio-ohjelmointi ei ole saavuttanut suurta suosiota kaikkien ohjelmoijien keskuudessa. Esimerkiksi C++-kieli on hyvin laaja ja vaikeasti hallittava kieli (Koskimies, 2000).

Olio-ohjelmointia voidaan kuitenkin tulkita monella eri tavalla. Useasti olio-ohjelmointi yhdistetään pelkästään kieliin, kuten C++ ja Java. Olio-ohjelmointi on kuitenkin ensisijaisesti ohjelmointitekniikka, eikä vain jonkin tietyn ohjelmointikielen ominaisuus. Tutkielmassani olenkin halunnut tarkastella olio-ohjelmoinnin toteuttamista kielillä, jotka eivät ole puhtaita oliokieliä. Puhdas oliokieli, esimerkiksi Smalltalk tai Java, käsittelee kaiken tietonsa olioiden kautta. Tutkielmassani tarkastelen erityisesti proseduraalisen Fortran 90 -kielen soveltuvuutta olio-ohjelmointiin.

Tutkielman rakenne on seuraava: luvussa kaksi esittelen olio-ohjelmoinnin keskeisiä käsitteitä pohjautuen Wegnerin (1987) näkemykseen olio-ohjelmoinnista. Wegnerin (1987) mukaan olio-ohjelmoinnin keskeisiin käsitteisiin kuuluvat olion, luokan, periytymisen, monimuotoisuuden ja myöhäisen sidonnan käsitteet.

Luvussa kolme esittelen olio-ohjelmoinnin toteuttamista perinteisillä proseduraalisilla ohjelmointikielillä: Pascalilla, Fortran 77:llä ja C-kielellä. Käsittelyni pohjana on luvussa kaksi esiteltyjen olio-ohjelmoinnin keskeisten rakenteiden toteuttaminen valitsemillani kielillä. Valitsin nämä kielet, koska Meyer (1988) on luokitellut nämä tärkeimmiksi proseduraalisiksi ohjelmointikieliksi. Lisäksi Pascal- ja C-kieli ovat tunnetusti niin sanottuja peruskieliä, mutta niitä ei pidetä olio-ohjelmointikielinä, joten käsittelin tutkimuksessani pääpiirteittäin olio-ohjelmoinnin peruskäsitteiden soveltuvuutta näille kielille. Samoin pidin tärkeänä kuvata Fortran 77 -kielen soveltuvuutta olio-ohjelmointikieleksi, koska se edeltää Fortran 90 -kieltä, jota tarkastelen tutkielmassani lähemmin. Fortran 90 -kieltä ei myöskään ole mielletty olio-ohjelmointikieleksi, joten olio-ohjelmoinnin keskeisten rakenteiden toteuttamisen tutkiminen

perinteisillä proseduraalisilla kielillä rakentaa pohjaa luvussa viisi tarkemmin käsiteltävälle Fortran 90 -kielen soveltuvuudelle olio-ohjelmointiin.

Luvussa neljä esittelen C++-hybridikielen, joka on proseduraalisen ja puhtaan olio-ohjelmointikielen välissä. Laajennan luvussa kaksi esiteltyjen ominaisuuksien lisäksi käsittelyäni olio-ohjelmointia tukeviin teknisiin ominaisuuksiin, vaikka ne eivät olekaan edellytyksiä olio-ohjelmoinnille. Käytännössä oliokielen tulee kuitenkin sisältää nämä ominaisuudet.

Luvussa viisi keskityn olio-ohjelmoinnin toteuttamiseen Fortran 90 -kielellä. Esittelen kuinka luvussa kaksi ja neljä esitellyt olio-ohjelmoinnin keskeiset rakenteet ja C++-kielen tekniset ominaisuudet voidaan toteuttaa Fortran 90 -kielellä.

2. Olio-ohjelmointi

Olio-ohjelmointi on ohjelmointimenetelmä – paradigma, jonka mukaisesti kirjoitetaan ohjelmia ongelmien ratkaisemiseksi. Sen lähtökohtana on kohdemaailman tietomalli, jossa tunnistetut pysyvät objektit kuvataan olioilla, joihin liittyvät sekä objektien ominaisuudet että niiden toiminnallinen puoli. Olio-ohjelmointi sisältää hyvin vähän teknisiä ominaisuuksia. Olio-ohjelmointi on termi, jonka eri ihmiset tulkitsevat eri tavalla. Mielestäni Wegner (1987) on tulkinnut olio-ohjelmoinnin käsitettä selkeästi: suunnittelun lähtökohtana ovat oliot, jotka on ryhmitelty luokkiin määrittelyn tekemiseksi. Perinteisen tietoelementtien välisen riippuvuuden lisäksi luokilla on mahdollisuus kuvata periytymissuhteita. Tekniset ominaisuudet, kuten myöhäinen sidonta ja monimuotoisuus, lisätään alemman tason suunnittelussa ja toteutuksessa (Wegner, 1987).

2.1 Olio-ohjelmoinnin synty

Perinteiset tietokonekielet perustuvat yleensä kahteen tekijään – tietoon eli dataan sekä algoritmeihin. Data koostuu tiedosta, jota ohjelma käyttää ja käsittelee. Algoritmit ovat ohjelman käyttämiä toimintatapoja.

Tietokonekielien ensimmäinen muoto oli konekieli. Varsin pian symbolinen konekieli korvasi konekielen binäärisen esitysmuodon. Symbolinen konekieli esittää komennot symboleina, jotka symbolisen konekielen kääntäjä muuttaa konekieliseksi. Esimerkiksi käsky: ”load R3, 54”, joka tarkoittaa ”laita rekisteriin numero kolme tieto muistipaikasta 54” on huomattavasti luettavampi kuin sitä vastaava jono bittejä. Vaikkakin konekieli oli luettavampaa kuin pelkkä bittimuotoinen koodi, se vain ilmaisi samat asiat eri tavalla. Näin ollen ohjelmoijan täytyi edelleen ajatella hyvin koneläheisesti. Seuraava askel olivat proseduraaliset kielet, joiden alkeisoperaatiot olivat korkeamman tasoisia kuin konekielessä (Ben-Ari, 1996).

Proseduraalisen ohjelmoinnin painopiste on algoritmeissa, joita käytetään laskennan suorittamiseen (Brookshear, 1994). Proseduraalinen ohjelmointityyli oli ensimmäinen, jolla ohjelmien koodia voitiin esittää samaan tapaan kuin pseudokoodia. Tämä onnistui luomalla korkean tason aliohjelmia, joiden sisältöä tarkennettiin alemman tason käskyillä. Ohjelmointikielet tukevat proseduraalista ohjelmointia välittämällä argumentteja aliohjelmille ja palauttamalla arvoja aliohjelmista. Proseduraaliseen ohjelmointityyliin perustuvan ohjelmiston suunnittelun

lähtökohtana on siis korkeimman tason toiminta, jota asteittain tarkennetaan. Tarkennusta jatketaan, kunnes ohjelmisto on jaettu pieniin, helposti ohjelmoitaviin aliohjelmiin.

Proseduraalinen ohjelmointityyli on käyttökelpoinen pienissä sovelluksissa, mutta suurissa sovelluksissa ongelmaksi muodostuu suurien tietomäärien hallinta. Ohjelmien suunnittellussa onkin painotettu tiedon organisointia pelkän aliohjelmien suunnittelemisen sijasta. Suurten ohjelmistojen hallitseminen on vaikeaa, ellei ohjelmistoa voida organisoida eli jakaa pienempiin järkevämmin hallittaviin kokonaisuuksiin. Näitä hallittavia kokonaisuuksia kutsutaan yleensä *moduuleiksi* (Ben-Ari, 1996 ja Brookshear, 1994). Teknisesti moduuli on joukko toisiinsa liittyviä aliohjelmia ja niiden käsittelemiä tietoja.

Moduuleista voidaan johtaa *abstraktin tietotyypin* käsite. Abstrakti tietotyyppi määrittelee ilmentymiinsä sovellettavat operaatiot. Operaatioiden toteutustapa ja niiden käyttämät tietorakenteet eivät kuitenkaan kuulu abstraktin tietotyypin määritelmään. Moduulin sisäinen toteutus voidaan piilottaa kyseistä moduulia käyttäviltä ohjelmayksiköiltä. Tätä menetelmää kutsutaan myös tietojen piilottamisen periaatteeksi. Tällöin voidaan toteuttaa abstrakteja tietorakenteita, joissa käyttäjä näkee tietorakenteelle määritetyt operaatiot, mutta ei pääse vaikuttamaan niiden sisäiseen toteutukseen. Näin ohjelmoijat voivat halutessaan muuttaa toteutuksen yksityiskohtia ohjelmiston toiminnallisuuden muuttumatta.

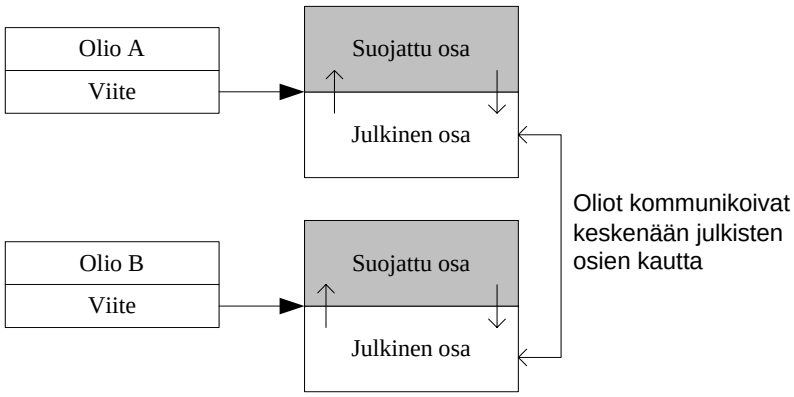
2.2 Olio

Olioita voidaan tarkastella usealla eri tavalla. Olen valinnut tarkasteluni pohjaksi Koskimiehen (2000) ja Wegnerin (1987) näkemykset, koska mielestäni molemmat tuovat tärkeitä ja toisistaan poikkeavia näkökulmia olioiden määrittelyyn. Koskimies (2000) on käsitellyt oliota hyvin yleisellä tasolla; pyrkien sen avulla mallintamaan ympäröivää maailmaa. Wegner (1987) puolestaan tarkastelee oliota teknisestä näkökulmasta, osana ohjelmointikieltä. Koskimiehen (2000) mukaan olio voidaan abstraktissa mielessä määritellä ympäristöstään erottuvana kokonaisuutena: sillä on oma identiteettinsä, sisäinen rakenteensa sekä suhteet tiettyyn ympäristöön. Olio ei ole puhtaasti toiminnallinen, kuten esimerkiksi aliohjelma, tai puhtaasti tietoa säilyttävä, kuten esimerkiksi tietue, vaan se sisältää molemmat ominaisuudet. Wegner (1987) puolestaan yksinkertaistaa olion joukoksi operaatioita sekä tilaksi, joka muistaa operaatioiden vaikutukset. Kolmas näkökulma on Korsonin ja McGregorin (1990) olion tekninen määrittely. Tällöin oliot ovat peruskokonaisuuksia oliopohjaisessa järjestelmässä. Ne käyttä-

vät sovelluksen muistiavaruutta ja niillä on esimerkiksi C-kielen tai Pascalin tietueita vastaava muistiosoite. Toisin sanoen bittien järjestys olion varaamassa muistissa määrittää olion tilan ja jokaiseen olioon liittyy joukko aliohjelmia, jotka määrittelevät olion toiminnallisuuden (Korson ja McGregor, 1990).

Olio-ohjelmointia käsittelevässä kirjallisuudessa (muun muassa Koskimies, 2000) olioiden keskeisiksi ominaisuuksiksi on määritelty oliolle ominaisten toimintojen suoritus, tiedon tallennus olion attribuutteihin, identifioiva tunniste ja suojaus.

Olion tulee pystyä suorittamaan oliolle ominaiset toiminnot. Näitä toimintoja kutsutaan *operaatioiksi*. Kullakin operaatiolla on nimi, mahdollisesti parametreja sekä toiminnan määrittelevä runko. Lisäksi operaatio voi olla arvon palauttava aliohjelma (Koskimies, 2000). Wegnerin (1987) mukaan olion operaatioita voi verrata tavallisiin aliohjelmiin, jotka ovat riippuvaisia ainoastaan parametreistaan; aliohjelma toimii aina samalla tavalla samoilla parametreilla. Sen sijaan olion operaatiot voivat olla riippuvaisia parametrien lisäksi olion omasta tilasta. Olio pystyy oppimaan operaatioiden toimintaa seuraamalla (Wegner, 1987).



Kuva 2.1 Olion viite ja suojaus.

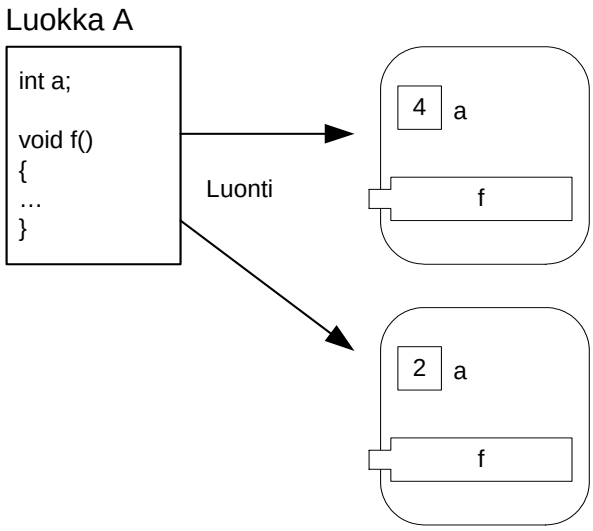
Oliolla on kyky tallentaa tietoa *attribuutteihinsa* eli nimettyihin tietokenttiin. Operaatioiden suoritus voi muuttaa attribuuttien arvoja. Olion attribuutit ja operaatiot muodostavat olion piirteet. Olion luonnin yhteydessä voidaan yleensä olion attribuutit alustaa asettamalla niille sopivat arvot.

Jokaisella oliolla on sen identifioiva tunniste, joka voi olla myös *viite* (kuva 2.1). Tunniste on se pysyvä tunnus, jonka järjestelmä antaa oliolle olion syntymisen yhteydessä. Kahdella eri oliolla on eri tunniste, vaikka ne olisivat toistensa kopioita.

Olio on *suojattu* kokonaisuus, jonka käyttö on rajattu tiettyihin muotoihin (kuva 2.1). Yleensä vain tietyt olion piirteet ovat käytettävissä olion ulkopuolelta. Olion omat operaatiot toimivat suojauksen sisäpuolella. Ne pääsevät vapaasti käyttämään olion omia attribuutteja ja muita operaatioita.

2.3 Luokka

Olio-ohjelmointi perustuu *luokkien* määrittelemiseen. Oliot rakentuvat samanlaisiksi kuin se luokka, johon ne kuuluvat (kuva 2.2). Luokka määrittelee sen tiedon ja ne operaatiot, jotka kuuluvat kaikille kyseisen luokan mukaan luoduille olioille. Toisin sanoen luokka määrittelee sen mukaan luodun olion ominaisuudet ja toiminnallisuuden (Klump, 2001). Luokan operaation tapauksessa ei ole oleellista, ajatteleeko sen sijaitsevan luokassa vai oliossa, koska operaatio on vakio kaikille luokan olioille. Luokka ei kuitenkaan suoraan määrittele olion toteutustapaa tietokoneessa, vaan tämä määräytyy käytettävän ohjelmointikielen toteutuksen mukaan (Koskimies, 2000).



Kuva 2.2 Luokan mallin mukaan luodaan useita olioita. Olioilla on samat attribuutit (a) ja operaatiot (f) (Koskimies, 2000).

Abstrakti tietotyyppi on hyvin lähellä yleistä luokan käsitettä. Abstrakti tietotyyppi määrittelee käyttäjilleen vain ilmentymiinsä sovellettavat operaatiot, mutta ei niiden toteutustapaa eikä toteutukseen tarvittavia tietorakenteita. Samanlaisen tietoabstraktion saavuttamiseksi luokan määrittelyssä käytetään tiedon kätkentää. Ominaisuudet, joita luokan käyttäjän ei ole tarkoitus käyttää, määritellään joko yksityisiksi tai suojatuiksi. Yksityiset tai suojatut ominaisuudet ovat luokan käyttäjälle saavuttamattomia. Käyttäjän käytettävissä olevat ominaisuudet voidaan määrittää julkisiksi. Tietoabstraktion säilymisen takia on tärkeää, että olio pystyy suojaamaan sisäisen tietonsa ulkopuolisilta käyttäjiltä. Näin olio tarjoaa itsestään ulkopuolisille käyttäjille yleisen määritelmän, joka kertoo mitkä asiat olio pystyy tekemään, mutta ei sitä, miten nämä asiat toteutetaan.

Luokkien ja olioiden ero voidaan esittää myös Meyerin (1988) tavalla. Tällöin luokka on tyyppi, jonka ilmentymä on olio. Luokka on staattinen, koska se on osa koodia. Vastaavasti olio on dynaaminen, sillä se luodaan sovelluksen suorituksen aikana. Luokkien ei tarvitse käytännössä esiintyä sovelluksen suorituksen aikana.

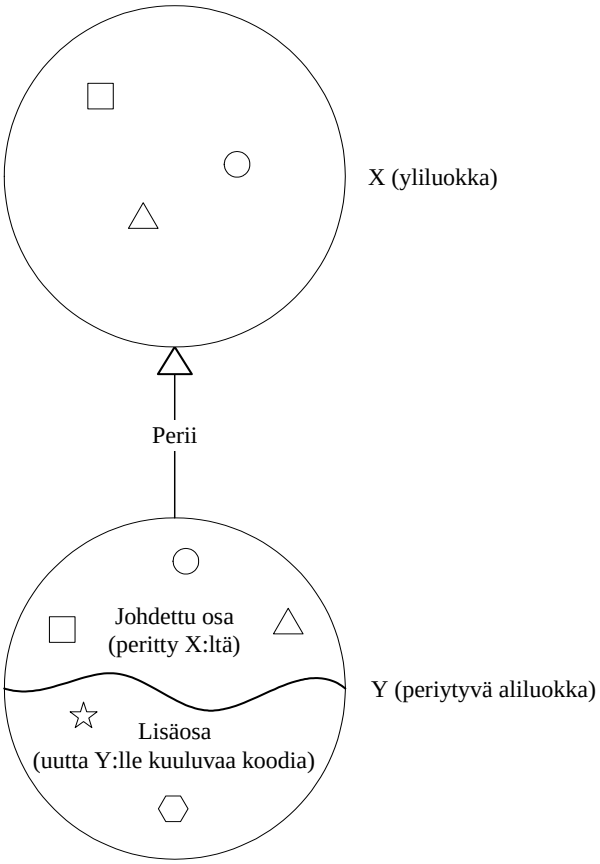
2.4 Periytyminen

Yksi oliopohjaisen ohjelmoinnin keskeisistä päämääristä on koodin uudelleenkäytettävyys. Uutta, laajaa ohjelmaa luodessa on kätevää, jos voidaan käyttää valmista ja jo testattua koodia sen sijaan, että kaikki kirjoitettaisiin alusta (Korson ja McGregor, 1990). Tämä säästää aikaa ja vähentää virheitä.

Perinteiset kirjastot sisältävät valmiiksi käännettyä koodia, jota käyttäjä voi hyödyntää omissa sovelluksissaan. Tällaiset kirjastot ovat kuitenkin rajoittuneita, sillä yleensä niiden mukana ei tule lähdekoodia, eikä käyttäjä siis voi muokata tai laajentaa lähdekoodia omia tarpeitaan vastaavaksi. Perinteisiä kirjastoja käytettäessä käyttäjän on muokattava oma koodinsa yhteensopivaksi kirjastojen koodin kanssa. Toisaalta, vaikka lähdekoodi olisikin mukana, on aina mahdollisuus, että käyttäjä tekee varomattomia muutoksia, joiden vaikutuksia on hyvin vaikea arvata.

Periytyminen mahdollistaa uusien luokkien johtamisen vanhoista (Kuva 2.3). Tällöin uusi luokka perii kaikki vanhan niin sanotun yliluokan ominaisuudet ja operaatiot. Periytyminen on transitiivista eli aliluokka perii ominaisuutensa yliluokalta ja samalla siis myös yliluokkaa

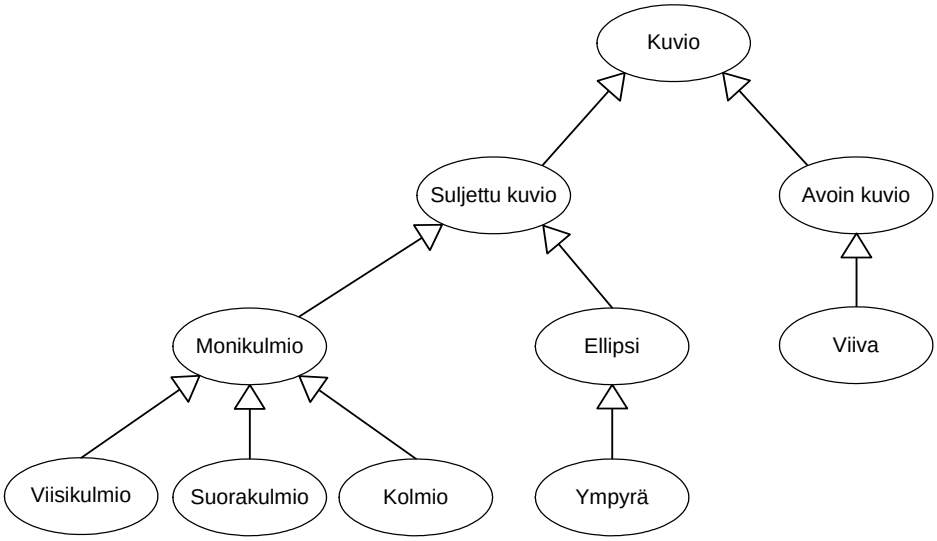
mahdollisesti edeltäneiden yliluokkien ominaisuudet. Uuteen luokkaan voidaan lisätä uusia ominaisuuksia ja operaatioita, ja on jopa mahdollista määritellä periytyneiden ominaisuuksien ja operaatioiden toiminta uudelleen (Korson ja McGregor, 1990).



Kuva 2.3 Periytyminen (Korson ja McGregor, 1990).

Mikäli luokka perii ominaisuutensa vain yhdeltä yliluokalta, kutsutaan tätä *yksittäisperiytymiseksi*. Yksittäisperiytymisessä yliluokkasuhteet antavat luokille puumaisen rakenteen, jossa kukin luokka esitetään solmuna siten, että solmun vanhempana on luokan yliluokka (kuva 2.4). On myös mahdollista, että luokka perii ominaisuutensa useammalta yliluokalta. Tällöin on kyseessä *moniperiytyminen*. Moniperiytymisessä luokkahierarkia ei enää muodosta puuta vaan syklittömän verkon. Syklittömyys seuraa siitä, että luokka ei voi periä itseään suoraan tai epäsuorasti.

Periytyminen vaikeuttaa kuitenkin koodiin perehtymistä, sillä aliluokka saattaa olla periytynyt useasta yliluokasta. Tarkasteltaessa luokan aliluokan toimintaa joudutaan tarkastelemaan myös yliluokkien toteutusta.



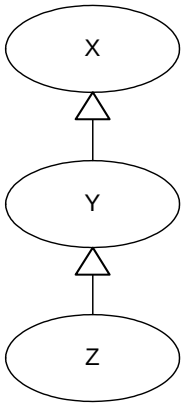
Kuva 2.4 Esimerkki periytymishierarkiasta (Korson ja McGregor, 1990).

2.5 Monimuotoisuus

Monimuotoisuudella tarkoitetaan sitä, että sovelluksessa esiintyvä kohde voi tarkoittaa useita erilaisia asioita. Oliokielessä monimuotoisuus tarkoittaa sitä, että monimuotoinen viite voi viitata usean eri luokan olioihin, kunhan luokkien oliot kuuluvat samaan periytymishierarkiaan (periytymispuuhun). Näin monimuotoinen viite voi siis viitata siihen luokkaan, josta se on lähtöisin, tai samassa periytymispuun haarassa oleviin sitä alempiin luokkiin (Korson ja McGregor, 1990). Esimerkiksi Monikulmion viite voi viitata Monikulmioon, tai Viisikulmioon, Suorakulmioon tai Kolmioon, mutta ei esimerkiksi Ympyrään (kuva 2.4).

Monimuotoiseen viitteeseen liittyvät staattisen ja dynaamisen luokan käsitteet. Monimuotoisen viitteen *dynaaminen luokka* voi vaihdella ohjelman suorituksen aikana. *Staattinen luokka* määrittelee kaikki ne mahdolliset luokat, joihin dynaaminen luokka voi viitata ohjelman suorituksen aikana. Ohjelmointikielen kääntäjä lukee valmiiksi annetun staattisen luokan sovelluksen käännöksen aikana. Esimerkiksi, kun ohjelman koodissa on esitelty viite, jonka tyypiksi on annettu Monikulmio (kuva 2.4), on Monikulmio tällöin viitteen staattinen luokka. Tämä määrittely perustuu ohjelmointikielen kääntäjän tekemään periytymishierarkian analysointiin. Dynaaminen luokka voi liikkua vain staattisen luokan määräämissä rajoissa; staattinen luokka ja kaikki siitä periytyvät luokat kelpaavat.

```
#include <iostream>

using namespace std;

class X
{
public:
    virtual void Tulosta(void)
    {
        cout << "X\n";
    }
};

class Y : public X
{
public:
    virtual void Tulosta(void)
    {
        cout << "Y\n";
    }
};

class Z : public Y
{
public:
    virtual void Tulosta(void)
    {
        cout << "Z\n";
    }
};

void main(void)
{
    X Xobj, *pX=NULL;
    Y Yobj;
    Z Zobj;

    pX=&Xobj;
    pX->Tulosta();
    pX=&Yobj;       // Lauseke 1
    pX->Tulosta();  // Lauseke 2
    pX=&Zobj;       // Lauseke 3
    pX->Tulosta();  // Lauseke 4
}


--- Ohjelman tuottama tuloste ---
X
Y
Z
```



Kuva 2.5 Monimuotoisuus ja myöhäinen sidonta.

Kuvassa 2.5 on toimiva C++-kielinen ohjelma, jossa esiintyy monimuotoisuutta. Ennen ensimmäistä lauseketta pX:n sekä staattinen että dynaaminen luokka on X. Ensimmäisen lausekkeen jälkeen pX:n staattinen luokka on pysynyt samana, mutta dynaaminen luokka on muuttunut luokaksi Y. Kolmannen lausekkeen jälkeen pX:n staattinen luokka on edelleen X, mutta dynaaminen luokka on Z.

2.6 Myöhäinen sidonta

Tavanomaisissa ohjelmointikielissä, kuten Pascalissa, aliohjelman toteutus voidaan jo kääntäessä liittää kutsuttavan aliohjelman määrittelyyn. Tämä on mahdollista siksi, että ennalta tiedetään kutsuttavan aliohjelman toteutus. Tätä kutsutaan staattiseksi sidonnaksi.

Myöhäisessä sidonnassa operaation (eli olion aliohjelman) toteutustapaa ei tiedetä ennen sovelluksen suorituksen aikaista operaation kutsua. Olioympäristössä myöhäinen sidonta liittyy periytymiseen ja monimuotoisuuteen: todellinen operaatio on riippuvainen monimuotoisen viitteen dynaamisesta luokasta (Korson ja McGregor, 1990).

Kuvassa 2.5 pX:n staattinen luokka on X, mutta sen dynaaminen luokka voi olla X, Y tai Z. Luokka Y on perinyt luokan X ominaisuudet ja määritellyt operaation ”Tulosta” toiminnallisuuden itse. Vastaavasti luokka Z perii Y:n ominaisuudet ja määrittää ”Tulosta” operaation toiminnallisuuden uudelleen. Mikäli pX käyttää operaatiota ”Tulosta”, on operaatio riippuvainen pX:n dynaamisesta luokasta. Toisessa lausekkeessa pX:n dynaaminen luokka on Y ja operaation ”Tulosta” toteutus on määritelty luokassa Y. Näin ohjelma tulostaa kirjaimen ’Y’. Neljännessä lausekkeessa vastaavasti pX:n dynaaminen luokka on Z ja operaation ”Tulosta” toteutus on määritelty luokassa Z.

Monimuotoisuuden ja myöhäisen sidonnan edut tulevat esille seuraavassa esimerkissä (Korson ja McGregor, 1990). Oletetaan, että kuvan 2.4 ”Kuvio”-luokalla on operaatio ”Piirrä”. Näin ollen jokaisella ”Kuvio”-luokasta periytyneellä luokalla on ”Piirrä”-operaatio. Jokaisen kuvion ulkoasu on kuitenkin erilainen, joten ”Piirrä”-operaation toiminnallisuus on joka luokalle erilainen. Ajatellaan kuvan 2.4 luokkahierarkian käyttämistä graafisessa sovelluksessa. Näytöllä olisi vaihteleva määrä kuvan 2.4 kuvioita. Kaikki kuviot voisivat olla yhdessä taulukossa, jossa jokainen taulukon alkio on viite luokkaan ”Kuvio”. Monimuotoisuuden avulla alkion viite voi viitata kaikkiin kuvan 2.4 olioihin. Näin kaikki oliot voitaisiin piirtää myöhäisen sidonnan avulla, eli käymällä kaikki taulukon alkiot läpi ja kutsumalla ”Piirrä”-operaatioita jokaisen alkion viitteelle.

3. Olio-ohjelmoinnin keskeisten rakenteiden toteuttaminen proseduraalisella kielellä

Tietty ohjelmointikieli ei välttämättä ole toista parempi siksi, että siinä on ominaisuus, joka toisesta puuttuu. Tärkeintä ei ole se, millaisia ominaisuuksia kielessä on, vaan se, että ominaisuudet riittävät halutun ohjelmointityylin noudattamiseen. Tutkijoilla on erilaisia näkemyksiä siitä miten eri ohjelmointikielet soveltuvat olio-ohjelmointiin. Esittelen tässä luvussa olio-ohjelmoinnin toteuttamista kolmella perinteisellä ohjelmointikielellä. Käsittelyni pohjana on Meyerin (1988) näkemys olio-ohjelmoinnin toteuttamisesta, joten valitsin samat ohjelmointikielet kuin hän: Pascalin, Fortran 77:n ja C-kielen. Olio-ohjelmoinnin toteuttaminen proseduraalisella kielellä johtaa yleensä kielen ominaisuuksien käyttöön tavalla, joka ei luultavasti ole niiden alkuperäinen tarkoitus. Tämä saattaa johtaa koodin sekavuuteen ja aiheuttaa ylimääräisiä virheitä (Meyer, 1988).

3.1 Pascal

Pascal-ohjelmointikieli on nimetty ranskalaisen matemaatikon ja keksijän Blaise Pascalin (1623 – 1662) mukaan. Kielen julkaisi Niklaus Wirth vuonna 1971. Pascal-ohjelmointikieli esitteli monia myöhemmin käytettyjä piirteitä, kuten tiedon vahvan tyypityksen, vapaamuotoisen syntaksin ja ohjausrakenteet. Nykyään Pascal-ohjelmointikieltä käytetään paljon tietojenkäsittelytieteen alkeisopetuksessa, koska se havainnollistaa miten ohjelma tulee rakentaa pala palalta, eikä se salli useissa muissa kielissä mahdollisia oikopolkuja. Aloittelija voi aloittaa luomalla yksinkertaisia ohjelmia ja taitojen kehittyessä lisätä erikoisempia piirteitä ohjelmiinsa (Brookshear, 1994).

Meyerin (1988) mukaan standardi Pascalilla ei voida esittää mitään muuta kuin rajoitettua tietorakenteen käsittelyä, mikä periaatteessa voidaan toteuttaa millä tahansa ohjelmointikielellä. Pascalin rajoitteena on ohjelmien rakenne, joka poikkeaa täysin oliopohjaisista. Pascal-ohjelma koostuu jonosta koodin osia, jotka esiintyvät muuttumattomassa järjestyksessä: otsikot, vakiot, tyypit, muuttujat, aliohjelmat (proseduurit ja funktiot) ja pääohjelma. Aliohjelmat sisältävät samanlaisen rakenteen. Aliohjelmien sisällä esitellyt muuttujat ovat näkyviä vain aliohjelmalle itselleen, kun taas ennen aliohjelmia esitellyt muuttujat ovat näkyviä kaikille aliohjelmille.

Tämä rakenne helpottaa Pascal-kielen kääntämistä, mutta se on Meyerin (1988) mielestä liian rajoittava olio-ohjelmointiin. Esimerkkinä Meyer käyttää abstraktin tietorakenteen toteuttamista Pascalilla. Tietorakenteena on pino toteutettuna taulukolla. Tietorakennetta varten tarvittaisiin: vakioita, kuten taulukon koko; tyyppejä, kuten pinon toteutusta kuvaava tietue; muuttujia, kuten osoitin pinon päällimmäiseen alkioon; sekä muutamia aliohjelmia, joissa on kuvattu tietorakenteen toiminta. Pascal-kielisessä ohjelmassa kaikki edellä mainitut elementit ovat ympäri ohjelmakoodia. Useampaa abstraktia tietorakennetta käytettäessä tiedot limittyvät toisiinsa, eivätkä voi muodostaa erillisiä selkeitä kokonaisuuksia.

Jacky ja Kalet (1987) kuitenkin esittävät artikkelissaan standardi Pascalilla toteutetun oliopohjaisen mallin. He olivat menestyksekkäästi käyttäneet malliaan yli 30 000 koodiriviä sisältävän sovelluksen luomiseen. Siis täysin päinvastainen tulos kuin Meyerilla (1988). Jackyn ja Kaletin mallilla pystytään esittämään oliot, luokat, operaatiot ja yksinkertainen periyttäminen.

Jackyn ja Kaletin mallissa oliot ovat Pascal-tietueiden ilmentymiä. Nämä tietueet määrittävät olion tilan. Olion luokka määritellään olion tietueen esittelyssä ja luokan nimi on lueteltu tyyppi (kuva 3.1, ”luokat = (X, Y)”). On tärkeää, että jokaista eri luokkaa varten ei ole määritelty omaa tietuetta, jotta olio-ohjelmointia voitaisiin käyttää vahvasti tyypitetyssä Pascalissa. Sen sijaan kaikki oliot ovat saman perustietueen ilmentymiä. Erilaisia luokkia saadaan määrittelemällä *muunneltava tietue* (kuva 3.1, ”case valinta …”). Luokan nimi esiintyy tietueen kentässä case-lauseen sisällä määrittämässä sen mistä luokasta on kyse. Esimerkiksi kuva 3.1: tietueessa on kaksi luokkaa X ja Y. Molemmat sisältävät kokonaislukumuuttujan nimeltä tunnus. Lisäksi X-luokka sisältää xmuuttuja nimisen kokonaislukumuuttujan ja Y-luokka sisältää ymuuttuja nimisen reaalilukumuuttujan.

```
type
    luokat = (X, Y);

    luokka = record
           tunnus : integer;

           case valinta : luokat of
                X: (xmja : integer);
                Y: (ymja : real);
    end;
```

Kuva 3.1 Kaksi esimerkkiluokkaa.

Muunneltavien tietueiden toteutuksessa kannattaa huomioida mahdollinen ylimääräinen muistinkulutus (Ben-Ari, 1996). Muunneltava tietue varaa aina muistia suurimman mahdollisen muunnelmansa mukaan. Esimerkiksi, jos tietueella on kaksi muunnelmaa, joista toisella on yksi kokonaislukutyyppinen muuttuja ja toisella sadan alkion kokonaislukutaulukko, niin kääntäjä varaa muistia aina sadan alkion kokonaislukutaulukolle.

Olio-ohjelmoinnissa olioita voidaan luoda ja tuhota tarpeen mukaan. Jotta tämä onnistuisi Pascal-kielessä, on käytettävä osoittimia olioiden hyödyntämiseen ja dynaamiseen luomiseen. Olio-ohjelmoinnissa oliot ovat yleensä suhteellisen pitkäikäisiä, sillä useimpien olioiden on tarkoitus palvella useammissa operaatioissa. Yleensä Pascalissa aliohjelman paikallinen muuttuja tai olio tuhoutuu aliohjelman loputtua. Tämän takia olioita täytyy käsitellä pääohjelmassa esiteltyjen osoittimien kautta. Nämä osoittimet ovat pysyviä jokaiselle aliohjelmalle.

Sen sijaan, että kaikille mahdollisille olioille olisi Pascalissa oma osoitin, Jackyn ja Kaletin mallissa oliot kerätään dynaamiseen listaan tai puuhun. Tällainen tietorakenne on hyödyllinen, koska näin kaikkiin olioihin päästään käsiksi joko listan ensimmäisen alkion tai puun juuren kautta. Edellä mainitut tietorakenteet ovat heterogeenisiä, koska kaikki oliot ovat saman tietueen ilmentymiä. Esimerkiksi lista-tietorakenteen tapauksessa kuvassa 3.1 olevaan luokkatietueeseen täytyy lisätä osoitin seuraavaan luokkaan (kuva 3.2).

```
type
    luokat = (X, Y);

    luokka_osoitin = ^luokka;

    luokka = record
           tunnus : integer;
           seuraava : luokka_osoitin;

           case valinta : luokat of
                X: (xmja : integer);
                Y: (ymja : real);
     end;
```

Kuva 3.2 Luokka, jossa on osoitin seuraavaan luokkaan.

Jackyn ja Kaletin mallissa olion operaatioita käytetään viestien avulla; viesti määrittää mitä operaatiota kutsutaan. Luokan kaikkia operaatioita vastaa proseduuri, joka välittää viestit. Proseduurin nimen alkuosa on aina sama ja loppuosa koostuu luokan nimestä. Esimerkiksi luokan ”Kuvio” viestiproseduuri olisi muotoa: ”invoke_kuvio”. Proseduurin ensimmäinen parametri on tietue, joka sisältää olion osoittimen ja viestin, jonka avulla kutsutaan oikeata operaatiota (kuva 3.3).

```
operaatiot=(LUO, TUHOA, TULOSTA);

viesti_tietue=record
    kohde_olio : luokka_osoitin;
    operaatio : operaatiot;
end;

...

procedure invoke_X(viesti : viesti_tietue);
begin
     case viesti.operaatio of
          LUO: ; { Olion luonti }
          TUHOA: ; { Olion tuhoaminen }
          TULOSTA: ; { Tulostaa olion tiedot }
     end;
end;
```

Kuva 3.3 Viestin rakenne ja luokan operaatioita.

Olio-ohjelmoinnissa on tärkeää, että kaikki luokan operaatiot on koottu yhteen. Jackyn ja Kaletin mallissa operaatio, jota halutaan kutsua, annetaan parametrina oliolle eli toisin sanoen olion luokan viestiproseduurille. Tämä eroaa proseduraalisesta ohjelmointityylistä, jossa tietorakenne annetaan parametrina proseduurille. Proseduraalisessa tyylissä jokaista erinimistä operaatiota vastaa oma proseduurinsa. Mikäli operaatio kuuluu useammalle luokalle, saatetaan proseduurin sisällä vielä joutua tekemään valintoja eri luokkien välillä. Edellä kuvatun viestipohjaisen menetelmän etu voidaan havaita uutta luokkaa lisättäessä. Tällöin uutta luokkaa varten riittää yhden uuden viestiproseduurin luominen, eikä se ole riippuvainen muiden luokkien proseduureista.

Olioiden operaatioita ei kuitenkaan käytetä kutsumalla suoraan ”invoke”-proseduuria. Sen sijaan pääohjelmassa on viestijono, jota pääohjelma tarkkailee jatkuvasti. Viestijonon seuraavaksi käsittelyvuorossa oleva viesti määrittää minkä luokan ”invoke”-proseduuria pääohjelma kutsuu. Mikäli olioiden operaatioita halutaan käyttää suoraan, on tätä tarkoitusta varten yleinen ”send”-proseduuri, joka luo uuden viestin ja lisää sen viestijonoon. Yleinen viestinlähetysmekanismi on tärkeä muun muassa olioiden keskinäisten operaatioiden kutsumisessa. Jackyn ja Kaletin mielestä samanlainen proseduuri kaikille luokille on monimuotoisuutta, koska eri olioiden operaatioita kutsutaan samalla tavalla.

Jackyn ja Kaletin mallissa periytyminen toimii hyvin alkeellisesti ja oikeastaan vain teorian tasolla. Kaikkien luokkien voidaan teoriassa ajatella periytyneen samasta tietueesta, koska kaikki luokat perivät ne tietueen osat, jotka eivät kuulu muunneltavaan osaan tietueesta (case-lauseke). Koodi, joka käsittelee näitä yhteisiä tietueen osia, toimii kaikille luokille samalla tavalla. Näin ollen kaikkien luokkien voidaan ajatella periytyneen yhdestä yliluokasta.

Olioiden ominaisuudet ovat kaikkialla käytettävissä ja muokattavissa, koska kaikkiin olioihin päästään käsiksi pääohjelmassa sijaitsevan dynaamisen tietorakenteen kautta. Tämä on ongelmallista. Jackyn ja Kaletin malli ei kuitenkaan esitä mitään keinoa, millä suojata luokkien ominaisuudet, vaan vastuu jää ohjelmoijalle.

3.2 Fortran 77

Fortran on vanhin ja eniten käytetty tieteellisen ja teknisen laskennan ohjelmointikieli. Fortran-kielen kehitti 1950-luvulla IBM:n kehitysryhmä, jota johti John Backus. Fortranin alku oli vaikea, koska ohjelmoijat eivät uskoneet konekieltä korkeamman tason kielen kääntäjän pystyvän laatimaan yhtä tehokasta koodia kuin he itse. Fortran oli aluksi puutteellinen niin kieleltään kuin moderneilta piirteiltään, kuten tiedon ja moduulien rakenteellisuudelta, koska Fortranin lähtökohtana oli tehokkuus, eikä esimerkiksi suurien tietomäärien hallinta (Ben-Ari, 1996).

Fortran oli kuitenkin monilta osiltaan parempi kuin konekieli. Fortranilla rekisterien käyttö ja konekieliset käskyt piilotettiin, joten ohjelmista tuli siirrettävämpiä, luotettavampia ja nopeampia kehittää. Koska ohjelmointi tietokoneiden alkuaikoina keskittyi tieteellisten ongelmien ratkontaan, tuli Fortranista de facto standardikieli, jota käytettiin tieteessä ja tekniikassa. Fortrania on aika ajoin päivitetty (1966, 1977, 1990 ja 1995) vastaamaan modernin ohjelmistokehityksen vaatimuksia (Ben-Ari, 1996).

Fortran 77:llä voidaan Meyerin (1988) mukaan saavuttaa rajoitettu suojaus, jolla voidaan toteuttaa suojattuja tietorakenteita. Fortran 77 -ohjelma koostuu pääohjelmasta ja aliohjelmista, jotka on Pascalin tavoin jaettu kahteen ryhmään: funktioihin ja proseduureihin. Yleensä abstrakti tietorakenne toteutetaan tarjoamalla tietorakenteen aliohjelmien esittely, mutta ei toteutusta. Fortran 77:ssä tietorakenteen toteutus voidaan jakaa aliohjelmien kesken käyttämällä Fortran 77:n *”COMMON”-lohkoja* (Korpela et al., 1985). Tällä tekniikalla on kuitenkin haittapuolensa, koska yhteisten muuttujien päivittäminen ”COMMON”-lohkoissa jää aliohjelmien vastuulle. ”COMMON”-lohkot eivät sisällä varsinaisia muuttujia, vaan muuttujien tieto tallennetaan tavuissa. Fortran 77 -kieli ei rajoita sitä, missä muodossa tieto luetaan ”COMMON”-lohkoista. Tietoa voi välittää aliohjelmasta toiseen myös aliohjelman parametrien ja paluuarvon kautta, mutta se ei aina ole käytännöllistä, koska hierarkkisessa aliohjelmarakenteessa parametrien määrä voi kasvaa hyvin suureksi.

Meyerin (1988) mielestä kapselointiin sopii ”COMMON”-lohkoja paremmin Fortran 77:ssä esitelty uusi ominaisuus: aliohjelmien vaihtoehtoiset aloituskohdat eli *”ENTRY”-lause*. Vaihtoehtoista aloituskohtaa eli ”sisäänkäyntiä” käyttämällä voidaan yhdistää kaksi tai useampia samantapaisia aliohjelmia yhdeksi aliohjelmaksi (Korpela et al., 1985). ”ENTRY”-lauseita voidaan kutsua aivan kuten muitakin aliohjelmia ja niiden parametrit voivat vaihdella. Kaikki aliohjelman ”ENTRY”-lauseet jakavat aliohjelman pysyvät tiedot, jotka Fortran 77:ssä on määritelty ”SAVE”-lausekkeella.

Abstrakti moduuli voidaan toteuttaa aliohjelmalla, jolla on vaihtoehtoisia aloituskohtia. Itse aliohjelmaa ei kuitenkaan koskaan kutsuta, vaan moduulia hallitaan vaihtoehtoisilla aloituskohdilla. Kuvassa 3.4 on esimerkki edellä mainittua tekniikkaa käyttävän abstraktin reaalilukupinon toiminnasta. Kuvassa 3.5 on abstraktin reaalilukupinon toteutus.

```
C     -- STACK EXAMPLE --
      LOGICAL OK
      REAL X

      OK=CREATE()
      OK=PUSH(4.5)
      OK=PUSH(-7.88)
      X=TOP()
      OK=POP()
```

Kuva 3.4 Abstraktin reaalilukupinon käyttö Fortran 77:llä (Meyer 1988).

Fortran 77:ssä aliohjelman ja sen vaihtoehtoisten aloituskohtien tyypin tulee olla joko funktio tai proseduuri, kuitenkin niin, että saman aliohjelman sisällä esiintyy vain yhtä tyyppiä (Korpela et al. 1985). Koska aliohjelmien ”EMPTY” ja ”TOP” on järkevää olla funktioita, ovat kaikki moduulin aliohjelmat funktiotyyppisiä eli ne palauttavat tuloksenaan arvon. Kuitenkin esimerkiksi ”CREATE”-aliohjelmassa palautusarvolla ei ole väliä, joten se voisi myös olla proseduurityyppinen aliohjelma, jos toteutus koostuisi proseduurityyppiä olevista aliohjelmista.

```
C     -- IMPLEMENTATION OF ONE ABSTRACT STACK OF REALS --
      INTEGER FUNCTION RSTACK()
      PARAMETER (SIZE=1000)
      REAL IMPL(SIZE)
      INTEGER LAST
      SAVE IMPL, LAST

C     -- DECLARATION OF ENTRY POINTS --
      LOGICAL CREATE
      LOGICAL PUSH
      LOGICAL POP
      REAL TOP
      LOGICAL EMPTY

      REAL X

C       -- STACK CREATION --
        ENTRY CREATE()
          CREATE=.TRUE.
          LAST=0
        RETURN

C       -- PUSHING ELEMENTS --
        ENTRY PUSH(X)
          IF(LAST .NE. SIZE) THEN
            PUSH=.TRUE.
            LAST=LAST+1
            IMPL(LAST)=X
          ELSE
            PUSH=.FALSE.
          END IF
        RETURN

C       -- POPPING STACK --
        ENTRY POP()
          IF(LAST .NE. 0) THEN
            POP=.TRUE.
            LAST=LAST-1
          ELSE
            POP=.FALSE.
          END IF
        RETURN

C       -- ACCESSING TOP --
        ENTRY TOP()
          IF(LAST .NE. 0) THEN
            TOP=IMPL(LAST)
          ELSE
            PRINT *, 'TOP: EMPTY STACK'
          END IF
        RETURN

C       -- DETERMINING IF STACK IS EMPTY --
        ENTRY EMPTY()
          EMPTY=(LAST .EQ. 0)
        RETURN
      END
```

Kuva 3.5 Abstraktin reaalilukupinon toteutus Fortran 77:llä (Meyer 1988).

Vaihtoehtoisilla aloituskohdilla voidaan kapselointia emuloida Fortran 77 -ympäristössä. Menetelmällä on kuitenkin myös rajoitteensa. Ensinnäkin moduulin sisäiset kutsut eivät ole mahdollisia. Fortran 77 tulkitsee toisen vaihtoehtoisen aloituskohdan kutsumisen saman aliohjelman sisällä rekursioksi, mitä Fortran 77 ei hallitse (Korpela et al. 1985). Toiseksi menetelmä on puhtaasti staattinen. Kuva 3.5 esittää yhden abstraktin pinon toteutuksen, mutta ei abstraktin pinon tietotyypin toteutusta. Abstraktin pinon tietotyypin avulla voitaisiin dynaamisesti

luoda ennalta määräämätön määrä pinoja. Tämä ei onnistu staattisessa menetelmässä. Kuitenkin ennalta määrätty määrä objekteja voitaisiin staattisessakin menetelmässä toteuttaa esimerkiksi taulukon avulla.

3.3 C-kieli

1970-luvun alussa Dennis Ritchie kehitti C-ohjelmointikielen UNIX-käyttöjärjestelmän ohjelmointiin. Käyttöjärjestelmät ohjelmoitiin perinteisesti konekielellä, koska korkeamman tason ohjelmointikieliä pidettiin tehottomina. C-kieli tuo yleisemmälle tasolle konekielisen ohjelmoinnin yksityiskohdat tarjoamalla rakenteellista ohjelmointia ja tietorakenteita, säilyttäen samalla konekielisen tason joustavuuden, kuten osoittimet ja bittioperaatiot. Koska UNIX oli helposti yliopistojen saatavilla ja se oli kirjoitettu siirrettävällä ohjelmointikielellä konekielen sijasta, siitä tuli hyvin suosittu akateemisissa yhteisöissä ja tutkimusinstituuteissa. C-kieli ja UNIX yleistyivät, kun uudet tietokoneet ja sovellukset levisivät näistä instituuteista kaupallisille markkinoille (Ben-Ari, 1996).

Meyerin (1988) mielestä olio-ohjelmoinnin toteuttaminen C-kielellä on mahdollista. Kuten millä tahansa muullakin ohjelmointikielellä, voidaan C-kielellä toteuttaa ohjelmointia, jossa tietorakennetta voidaan käyttää vain aliohjelmien kautta. C-kielessä tiedosto on käännettävä yksikkö; se voi sisältää aliohjelmia ja tietoja. Osa aliohjelmista voi olla piilotettuna muilta aliohjelmilta ja toiset puolestaan voivat olla näkyvissä muille aliohjelmille. Edellä mainituilla ominaisuuksilla voidaan C-kielessä toteuttaa kapselointia: tiedosto voi sisältää yhden tai useamman abstraktin objektin tai abstraktin tietotyypin toteutuksen.

Käytännössä C-kielessä usein käytetty menetelmä eroaa olio-ohjelmoinnin suunnittelun periaatteista. Useimmat C-kielen ohjelmat käyttävät otsikkotiedostoja, jotka kuvaavat jaettuja tietorakenteita. Mikäli C-kielinen tiedosto tarvitsee otsikkotiedostossa esiteltyjä tietorakenteita, se pyytää esikääntäjää sisällyttämään otsikkotiedoston itseensä. Tämä tapahtuu käyttämällä esikääntäjän ohjauskäskyä ”#include”, jolloin tiedosto saa otsikkotiedoston tietorakenteet käyttöönsä. Esimerkiksi: pinon tietorakenteen määrittelyn sisältävän otsikkotiedoston nimi on ”stack.h” ja pinon toteutuksen sisältävä ”stack.c”-tiedosto pyytää esikääntäjää sisällyttämään ”stack.h”- otsikkotiedoston itseensä seuraavalla käskyllä: #include ”stack.h”. Käsitteellisesti saman voisi toteuttaa kopioimalla koko otsikkotiedoston sisältö siihen kohtaan mihin se halutaan sisällyttää.

Olio-ohjelmoinnissa jokaisen olion voidaan ajatella pitävän sille sopivat muuttujat ja operaatiot mukanaan. Tämä on vain käsitteellinen ajatus, mutta toteutettaessa se käytännössä saadaan aikaiseksi mielenkiintoinen C-kielinen tekniikka. C-kielessä tietueet voivat muuttujien lisäksi sisältää osoittimia aliohjelmiin (kuva 3.6).

```
#define MAX 100

typedef struct
{
    int last;
    float impl[MAX];
    void (*pop)();
    void (*push)();
    float (*top)();
    int (*empty)();
} REAL_STACK;
```

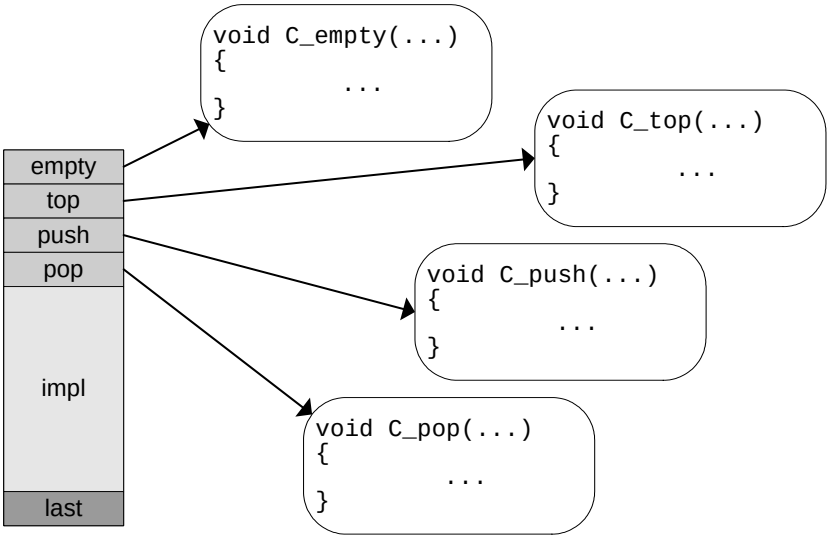
Kuva 3.6 Tietue, joka sisältää muuttujia ja osoittimia aliohjelmiin (Meyer 1988).

Kuvassa 3.6 on ”REAL_STACK”-niminen tietue, joka voisi kuvata luokkaa pino. Tietueen kaksi ensimmäistä muuttujaa ovat kokonaislukumuuttuja ja reaalilukutaulukko, muut muuttujat ovat osoittimia aliohjelmiin. Jokainen tietue täytyy alustaa siten, että aliohjelmien osoittimet osoittavat oikeisiin aliohjelmiin. Kuvassa 3.7 on esimerkki aliohjelman osoittimen asettamisesta.

```
REAL_STACK stack;

void C_pop(REAL_STACK s)
{
    /* toteutus */
}

void main(void)
{
    stack.pop=C_pop;
}
```

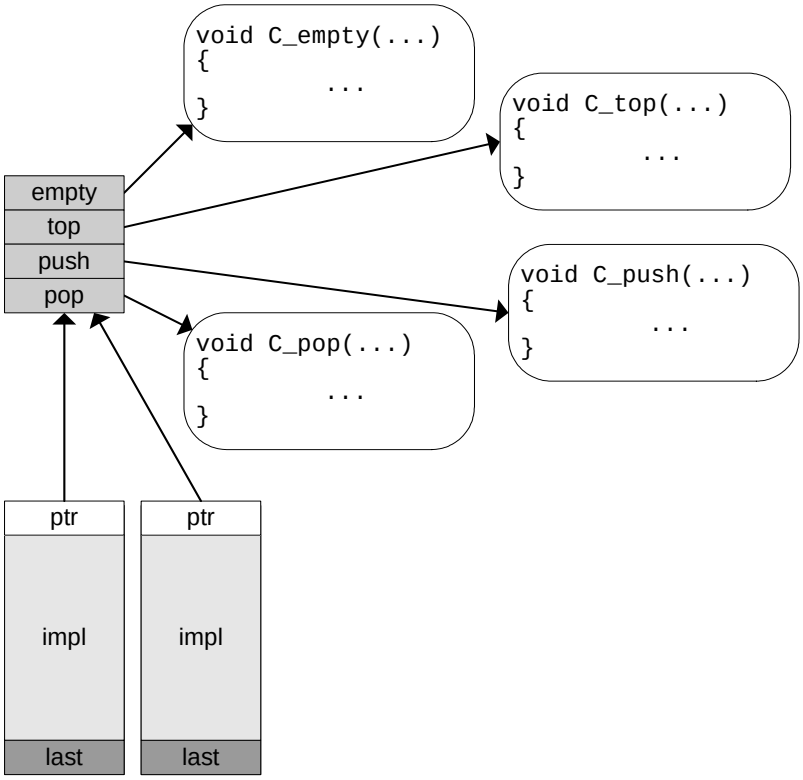
Kuva 3.7 ”REAL_STACK”-tietueen osoittimen asetus.

Voidakseen muokata pinoa ”C_pop”-aliohjelma tarvitsee parametrikseen pinon tietueen. Tämä pätee kaikille ”REAL_STACK”-nimisen tietueen aliohjelmille. Edellä kuvattu tekniikka toimii, mutta ei välttämättä ole muistinkäytön kannalta paras ratkaisu. Kuten kuvasta 3.8 nähdään, jokainen luokka sisältää osoittimet luokan kaikkiin aliohjelmiin.



Kuva 3.8 C-kielinen olio, joka sisältää osoittimia aliohjelmiin (Meyer 1988).

Tietuekohtaista muistin kulutusta voidaan kuitenkin vähentää. Tämä tapahtuu korvaamalla aliohjelmien osoittimet dynaamisesti varatulla listalla tai taulukolla, jota kutsutaan luokan kuvaukseksi. Luokan kuvaus on mahdollinen, koska jokainen olio käyttää luokalle yhteisiä aliohjelmia. Käyttämällä luokan kuvausta voidaan luokan aliohjelmien osoittimet korvata osoittimella operaatioiden kuvaukseen (kuva 3.9).



Kuva 3.9 Aliohjelmien jakaminen olioiden kesken (Meyer 1988).

Edellä kuvattua menetelmää on käytetty olio-ohjelmoinnin toteutuksen pohjana C-kielessä. Mikäli haluttaisiin hyödyntää periytymistä, tarvittaisiin Meyerin mielestä edistyneempiä tekniikoita. Meyerin (1988) mielestä olio-ohjelmoinnin toteuttaminen C-kielellä on mahdollista, mutta ei välttämättä suositeltavaa.

Stroustrup (1997) esittää yksinkertaisen periytymisrakenteen käyttäen C-kielen tietueita. Hänen mukaansa periytyminen voidaan toteuttaa sijoittamalla yliluokan olio aliluokkaan ja lisäämällä siihen aliluokan omat erikoisominaisuudet.

```
typedef struct
{
    char etunimi[16], sukunimi[16];
    int palkka;
} Tyontekija;

typedef struct
{
    Tyontekija tt;
    int alaisia;
} Johtaja;
```

Kuva 3.10 C-kielellä toteutettu alkeellinen periytyminen.

Kuvassa 3.10 on kaksi tietuetta, jotka kuvaavat yrityksen työntekijöitä. Johtaja on myös työntekijä, ja nämä tiedot on talletettu ”tt”-tietueeseen. Tämä voi olla selvää ohjelmoijalle, mutta kääntäjä ja muut työkalut eivät osaa mitenkään päätellä, että johtaja on myös työntekijä.

4. Olio-ohjelmoinnin toteuttaminen hybridikielellä

C++-kieli sai alkunsa Bell Laboratories-yhtiössä, jossa Bjarne Stroustrup kehitti sen 1980-luvun alussa. C++-kieli on C-kielen laajennus, joka lisää siihen tiedon abstraktoinnin ja olio-ohjelmoinnin piirteitä. C++-kieli laajentaa C-kielen syntaksia Simula-ohjelmointikielestä omaksutulla luokan käsitteellä. C++-kielen suunnittelun pohjana oli C-kielen syntaksin, tehokkuuden ja siirrettävyyden säilyttäminen (Stroustrup, 1994).

C++-kieli ei ole puhdas oliokieli, vaan hybridikieli, koska sillä tehty ohjelma ei ole koskaan täydellisesti ja pelkästään olioperustainen (Koskimies, 2000). C++-kieli tukee olio-ohjelmointia, mutta ei välttämättä edellytä sitä. Ohjelmoijat, jotka ovat aikaisemmin käyttäneet esimerkiksi proseduraalista ohjelmointityyliä, voivat vähitellen opetella ohjelmointia C++-kielellä ja hyödyntää monia C++-kielen etuja. Esimerkiksi C++-kielen standardikirjastojen käyttäminen on mahdollista C-kielen tietämystä hyödyntäen. Koska C++-kielen syntaksi on samanlainen kuin C-kielen, voidaan esimerkiksi C++-kielen standardikirjastojen käytössä hyödyntää samanlaista ohjelmointityyliä kuin C-kielessä.

Tarkastelen seuraavaksi kuinka osa olio-ohjelmoinnin keskeisistä rakenteista on toteutettu C++-kielellä. Näitä keskeisiä rakenteita ovat: oliot, luokat, operaatioiden ylikuormittaminen, suojaus, parametrien oletusarvot, periytyminen, monimuotoisuus ja myöhäinen sidonta. Tämän luvun tarkoituksena on antaa yleinen kuva C++-kielen ominaisuuksista, mutta ei kuvata kielen laajuuden vuoksi koko kieltä.

4.1 Luokat ja oliot C++:ssa

C++-kielessä oliot muistuttavat pitkälti perinteisen ohjelmoinnin muuttujia. C++-kielessä olion ominaisuuksia edustaa tietty muistialue, johon on tallennettu olion kulloinenkin tila, samaan tapaan kuin perinteisessä ohjelmoinnissa muuttujaa edustaa muistialue, johon on tallennettu muuttujan kulloinenkin arvo. C++-kielessä olion voi luoda esittelemällä luokan muuttujan tai käyttämällä ”new”-käskyä, joka luo olion ja palauttaa osoittimen luotuun olioon. Olioita voidaan ohjata aliohjelmien parametreiksi ja palauttaa aliohjelmien palautusarvojen kaltaisesti. Lisäksi olio voidaan sijoittaa toiseen olioon. Tällöin sijoituksen kohteena olevan olion attribuuteiksi asetetaan lähdeolion attribuutit.

Luokka on yksi tärkeimmistä C++-kielen käsitteistä (Stroustrup, 1994). C++-kielen luokalla on neljä ominaisuutta (Lippman, 1989). Ensinnäkin jokaisella luokalla on tietty joukko attribuutteja, jotka kuvaavat luokkaa. Attribuutteja voi olla yksi tai useampi tai ei yhtään. Toiseksi luokalla on tietty joukko operaatioita, joiden kohteena on luokan olio. Operaatioitakin voi olla yksi tai useampi tai ei yhtään. Kolmanneksi luokan eri osille on mahdollista määritellä eritasoisia suojausmääreitä. Neljänneksi luokalla täytyy olla yksilöivä nimi, joka palvelee tyypin määrittelynä.

Kuvassa 4.1 on C++-kielinen esimerkkiluokka, jonka nimi on ”CKuvio”. ”CKuvio”-luokka esittelee ”keskipiste”-attribuutin sekä erilaisia suojausmääreitä ja operaatioita. Operaatiolla ”Tulosta” on ”virtual”-määre, joka antaa ”CKuvio”-luokasta perityille aliluokille mahdollisuuden määrittää uudelleen operaation ”Tulosta” toiminta.

```
#include <iostream>

using namespace std;

class CKuvio
{
public: // Luokan julkiset osat
    CKuvio();
    ~CKuvio();

    // Siirra -operaatio on kuormitettu
    // Ilman sijaintia asettaa sijainniksi (0,0)
    // Sijainnin kanssa asettaa sijainniksi (x, y)
    void Siirra()               { sijx=0; sijy=0; }
    void Siirra(int x, int y)   { sijx=x; sijy=y; }

    virtual void Tulosta(bool sijainti=true);

protected: // Luokan suojatut osat
    int sijx;
    int sijy;
};

// Rakentaja
CKuvio::CKuvio()
{
    sijx=0;
    sijy=0;
}

//Purkaja
CKuvio::~CKuvio()
{
}

void CKuvio::Tulosta(bool sijainti)
{
    cout << "Kuvio\n";

    if(sijainti)
        cout << sijx << ", " << sijy << "\n";
}
```

Kuva 4.1 C++-kielinen esimerkkiluokka.

C++-kieli sisältää erityisen *rakentajaoperaation*, jota käytetään muodostettaessa uutta oliota. C++-kieli antaa valmiiksi rakentajaoperaation nimen ja syntaksin, ja ohjelmoija toteuttaa operaation. Rakentajaoperaatiolla on sama nimi kuin luokallakin. Esimerkiksi kuvan 4.1 ”CKuvio”-luokan rakentaja on ”CKuvio::CKuvio()”-operaatio, joka alustaa kuvion sijainnin.

Purkajaoperaatio puolestaan hoitaa olion hävittämiseen liittyvät lopputoimet, vapauttaa esimerkiksi varatun aputilan. Purkajaoperaatio ei varsinaisesti hävitä itse oliota, vaan C++-kieli aktivoi sen vasta olion hävittämisen yhteydessä. Esimerkiksi kuvan 4.1 ”CKuvio”-luokan purkaja on ”CKuvio::~CKuvio()”-operaatio, joka ei kyseisessä luokassa tee mitään. Luokalle voidaan määritellä useita rakentajia, mutta purkajia on aina vain yksi.

4.2 Operaatioiden ylikuormittaminen C++:ssa

Operaatioiden ylikuormittaminen mahdollistaa saman operaation kutsun erityyppisillä parametreilla. Kääntäjän kannalta kuormitettujen operaatioiden ainoa yhteinen ominaisuus on niiden nimi (kuva 4.1 ”Siirra”-operaatio). C++-kielen kääntäjän tehtävänä on analysoida operaation kutsun muotoa ja päättää, mitä operaatiota kutsu tarkoittaa. Kuormittaminen ratkaistaan siis käännösaikaisen informaation perusteella. Tämä informaatio koskee parametrien lukumäärää ja tyyppejä. Kuormitus on laiton, jos tämä tieto ei riitä yksilöimään mistä operaatioista on kyse (Koskimies, 2000).

Operaation palautustyyppiä ei oteta huomioon kuormitettua operaatiota valittaessa. Tarkoitus on nimittäin pitää operaation valinta kontekstista riippumattomana (Stroustrup, 1997). Esimerkiksi, jos ensimmäisen neliöjuuren laskevan operaation palautusarvo on reaaliluku ja toisen on kaksoistarkkuuden reaaliluku, ei C++-kielessä ole mahdollista palautusarvon perusteella valita kumpaa operaatiota kutsutaan.

4.3 Operaatioiden parametrien oletusarvot C++:ssa

Yleiskäyttöinen operaatio edellyttää usein enemmän argumentteja kuin yksinkertaisissa tapauksissa tarvitaan. Esimerkiksi operaatiossa, joka tulostaa kokonaisluvun, voisi olla järkevää antaa parametrina tieto siitä, missä lukujärjestelmässä luku halutaan tulostaa. Useimmissa tapauksissa käytetään kuitenkin vain kymmenjärjestelmän lukuja. Operaation ylikuormittamisella voitaisiin tehdä kaksi operaatiota, joista ensimmäinen ei saa parametrina lukujärjestel-

mää ja tulostaa luvun kymmenjärjestelmässä. Toinen operaatio saa parametrinaan myös halutun lukujärjestelmän ja tulostaa luvun halutussa järjestelmässä. Kuormittaminen ei kuitenkaan tee selväksi sitä, että halutaan yksi lyhyt tulostusoperaatio.

C++-kielessä operaatioiden parametreille voidaan määritellä myös *oletusarvoja*, joita käytetään, kun parametrille ei anneta arvoa (kuva 4.1 ”Tulosta”-operaatio) (Lippman, 1989). Parametrien oletusarvojen on oltava viimeisinä parametreina. Oletusarvoja käyttämällä voidaan tehdä luvun tulostusoperaatio, jossa lukujärjestelmäparametri saa arvon 10, jos parametria ei käytetä operaation kutsussa.

Operaatioiden kuormitus ja operaatioiden parametrien oletusarvot eivät ole oleellisia olio-ohjelmoinnin kannalta, mutta niistä on ilmeistä hyötyä. Luokalla voi olla tietyn operaation eri versioiden toteuttamista varten useita asiallisesti samaa tarkoittavia operaatioita samalla nimellä.

4.4 Luokan suojaus C++:ssa

C++-kielessä on kolme erilaista suojausmäärettä: ”*private*”, ”*protected*” ja ”*public*” (Lippman, 1989). Nämä määrittävät kuinka luokan attribuutteja ja operaatioita voi hyödyntää luokan sisältävässä ohjelmassa. Jos luokan attribuutti tai operaatio on määreeltään ”public”, ei siihen viittaamiselle ole mitään rajoitusta. Mikäli luokan attribuutti tai operaatio on määreeltään ”private”, siihen voi viitata vain luokan sisältä. Jos attribuutti tai operaatio halutaan antaa vain luokan jälkeläisluokkien, ei siis periytymishierarkian ulkopuolisten luokkien, käyttöön, sille annetaan määre ”protected”. Tyypillisesti luokan attribuutit ovat joko ”private”- tai ”protected”-tyyppisiä ja sovelluksen tarvitsemat operaatiot ”public”-tyyppisiä (kuva 4.1). Tällaista ”public- sekä joko ”private”- tai ”protected”-määrittelyä kutsutaan yleensä tiedon kätkennäksi C++-kielessä.

4.5 Periytyminen C++:ssa

C++-kieli tukee sekä yksittäis- että moniperiytymistä. Luokkien ei tarvitse olla periytyneitä toisista luokista, vaan ne voivat esiintyä itsenäisinä luokkina, toisin kuin puhtaassa oliokielessä, kuten Javassa. Esimerkiksi Java-kielessä kaikki luokat ovat periytyneitä yhdestä kantaluokasta (Ben-Ari, 1996). Moniperiytyminen C++-kielessä mahdollistaa sen, että periytyneessä

luokassa voivat yhdistyä usean luokan ominaisuudet. Periytyminen ilmaistaan luokan otsakkeessa luokan nimen jälkeen. Kuvassa 4.2 on ”CYmpyra”-luokka, joka perii kuvan 4.1 ”CKuvio”-luokan. Luokka ”CYmpyra” määrittelee uudelleen operaation ”Tulosta” ja lisää luokkaan yhden uuden yksityisen (”private”) attribuutin nimeltään ”halkaisija”. Lisäksi määrittelin ”halkaisija”-attribuutille kaksi operaatiota, joilla attribuutin tiedot voidaan lukea tai niitä voidaan muokata.

```
class CYmpyra : public CKuvio
{
public:
    CYmpyra();

    int Halkaisija() { return halkaisija; }
    void AsetaHalkaisija(int halk) { halkaisija=halk; }
    void Tulosta();

private:
    int halkaisija;
};

CYmpyra::CYmpyra()
{
    halkaisija=0;
}

void CYmpyra::Tulosta()
{
    cout << "Ympyrä\n";
}
```

Kuva 4.2 Luokan periytyminen C++-kielessä.

Yliluokan nimeen voidaan periytymisen yhteydessä liittää myös näkyvyysmääre, joka antaa perityille piirteille joko ”private”- tai ”public”-suojausmääreen. Kuvassa 4.2 yliluokka on määritelty ”public”-tyyppiseksi, joten periytyvä aliluokka pääsee käsittelemään yliluokan ”public”- ja ”protected”-piirteitä, mutta ei ”private”- piirteitä. Samalla yliluokan ”public”- ja ”protected”-piirteistä tulee aliluokan piirteitä. ”private”-piirteet eivät tässä tapauksessa lainkaan periydy uuteen luokkaan. Yliluokka voidaan kuitenkin määrittää myös ”private”-tyyppiseksi, jolloin yliluokan ”public”- ja ”protected”-piirteistä tulee periytyvän luokan ”private”-piirteitä.

4.6 Monimuotoisuus ja myöhäinen sidonta C++:ssa

Koskimiehen (2000) mukaan monimuotoisuus ja siihen liittyvä myöhäinen sidonta voidaan luoda osoitintyyppisillä arvoilla. C++-kieli mahdollistaa periytyneen luokan ilmentymän osoitteen sijoittamisen yliluokan osoittimeen. Kuvassa 4.3 on C++-kielinen ohjelmaesimerkki

monimuotoisuudesta ja myöhäisestä sidonnasta luokilla ”CKuvio” ja ”CYmpyra” sekä ohjelman tuottama tulostus.

```
CKuvio *p=NULL;

p=new CKuvio();
p->Siirra(1, 2);
p->Tulosta();
delete p;

p=new CYmpyra();
p->Siirra(3, 4);
p->Tulosta();
delete p;


--- Ohjelman tuottama tuloste ---
Kuvio
1, 2
Ympyrä
3, 4
```

Kuva 4.3 Monimuotoisuus ja myöhäinen sidonta luokilla “CKuvio” ja “CYmpyra”.

Kuvan 4.3 osoittimen ”p” perusluokka on ”CKuvio” tai jokin sen jälkeläisluokka. Olio luodaan ”new”-käskyllä, joka palauttaa osoittimen olioon. Kuten kuvasta 4.3 nähdään, voi osoitin ”p” osoittaa sekä perusluokkaan että sen jälkeläisluokkiin. Kun osoittimen ”p” osoittamaa oliota pyydetään suorittamaan operaatio, kirjoitetaan osoitinta tarkoittavan ilmaisun jälkeen nuolisymboli ”->”. Symbolin perään liitetään operaation kutsu. Tällöin ei Koskimiehen (2000) mukaan kyseisen olion ominaisuuksia välttämättä tunneta staattisesti, jolloin tarvittaessa sovelletaan myöhäistä sidontaa.

5. Olio-ohjelmoinnin toteuttaminen Fortran 90:llä

Fortran 90 on moderni ja tehokas kieli, jolla voidaan toteuttaa monia tärkeitä ohjelmointitekniikkoja, mukaan lukien olio-ohjelmointia. Fortran 90 -standardi oli merkittävä edistysaskel Fortran-kielen kehitykselle. Se toi Fortran-kielen muiden modernien ohjelmointikielten tasolle (Haataja et al., 1998). Tässä luvussa kuvaan olio-ohjelmoinnin toteuttamista Fortran 90 -kielellä, käyttäen pohjana luvussa kaksi esiteltyjä olio-ohjelmoinnin peruspiirteitä. Fortran 95 -standardi ei sisällä suuria muutoksia Fortran 90 -kieleen, ja varsinkaan kun se ei sisällä olio-ohjelmoinnin kannalta oleellisia lisäyksiä, olen tyytynyt käyttämään Fortran 90 -kieltä. Käsittelyni pohjana käytin Decykin, Nortonin ja Szymanskin Fortran 90 -kieltä käsitteleviä artikkeleita (1997a, 1997b, 1998).

5.1 Fortran 90:n olio-ohjelmointia tukevat piirteet

Aiemmista Fortran-versioista poiketen Fortran 90 sisältää mahdollisuuden käyttää rakenteisia tietotyyppejä taulukoiden ja skalaarien lisäksi. Rakenteisilla tietotyypeillä voi selkeyttää ohjelmaa kokoamalla yhteen rakenteeseen toisiinsa liittyviä tietoja.

Fortran 90 sisältää myös dynaamisen tilanvarauksen ja osoittimet (Haataja et al., 1998). Fortran 90:ssä osoitin voi osoittaa yksinkertaisen tietotyypin muuttujaan, taulukkoon, rakenteiseen tietotyyppiin tai nimeämättömään varattuun muistialueeseen. Dynaamisesti varatuilla taulukoilla päästään eroon aiempien Fortran-kielen versioiden hankalista työtilavektoreista. Ohjelman kokonaisuudessaan vaatimaa muistitilaa ei siis Fortran 90:ssä tarvitse enää varata ohjelman suorituksen alussa.

Olio-ohjelmoinnin kannalta Fortran 90:ssä tärkein uudistus ovat moduulirakenteet. Fortran 90:n moduulirakenteet ja moduulien sisäiset aliohjelmat mahdollistavat ohjelmakoodin jakamisen erillisiin, toisistaan riippumattomiin osiin (Haataja et al., 1998). Fortran 90:n moduuleissa voidaan määritellä muun muassa nimettyjä vakioita, muuttujia, rakenteisia tietotyyppejä ja aliohjelmia. Kunkin moduulin sisäinen toteutus voidaan piilottaa kyseistä moduulia käyttäviltä ohjelman osilta. Moduulien avulla voidaan myös korvata vanhoissa Fortran-ohjelmissa käytetyt ”COMMON”-lohkot. Yhdistämällä moduulit ja rakenteiset tietotyypit voidaan moduulin tietotyypille määritellä halutunlaisia aliohjelmia. Näin saadaan muodostettua abstrakteja tietotyyppejä, jotka koostuvat tiedosta ja tiedolle määritetyistä aliohjelmista.

Fortran 90 -moduulien tärkeisiin ominaisuuksiin kuuluvat myös suojaus, operaattoreiden määrittely ja geneeriset aliohjelmat (Haataja et al., 1998). Fortran 90:ssä moduuli suojataan määrittelemällä moduulin eri osat joko julkisiksi tai yksityisiksi. Oletusarvoisesti, jos moduulin osan määrittelyssä ei muuta sanota, osa on julkinen. Julkiset osat ovat moduulin käyttäjälle suoraan näkyviä, kun taas yksityisiin osiin käyttäjä ei voi suoraan viitata.

Moduulien avulla on mahdollista määritellä uusia operaattoreita ja antaa vanhoille operaattoreille uusia merkityksiä eri tietotyyppien yhteydessä. Ohjelmoija voi antaa uuden merkityksen numeerisille standardioperaattoreille, sijoitusoperaattorille sekä loogisille- ja vertailuoperaattoreille. Alkuperäisen tietotyypin valmiiksi määritetylle operaattorille ei kuitenkaan voi määrittää uutta merkitystä. Täten esimerkiksi kahden kokonaisluvun yhteenlaskun tulosta ei voi muuttaa.

Geneerinen aliohjelma voi toimia eri tavoin sen mukaan, minkä tyyppisiä argumentteja sille annetaan. Geneeriselle aliohjelmalle annetaan moduulissa yksi nimi, jota vastaa useampi aliohjelma. Tämä vastaa siis C++-kielen operaation ylikuormitusta. Nimeä vastaavien aliohjelmien täytyy olla joko palautusarvollisia tai palautusarvottomia aliohjelmia. Kutsuttaessa geneeristä aliohjelmaa kääntäjä valitsee aliohjelman parametrien tyypin perusteella. Tämän vuoksi geneerisen aliohjelman nimen alla olevien aliohjelmien parametrien tulee olla erotettavissa toisistaan. Geneerinen aliohjelma on julkinen, mutta sen eri aliohjelmia ei voi kutsua moduulin ulkopuolelta mikäli ne määritetään suojatuiksi.

5.2 Luokat ja oliot Fortran 90:ssä

Olio-ohjelmoinnissa luokka esittelee uuden tietotyypin ja sille soveltuvat aliohjelmat. Tietotyypin osat ovat luokan attribuutteja ja aliohjelmat ovat luokan operaatioita. Fortran 90 ei varsinaisesti sisällä luokan käsitettä, joten se määritellään moduulina, joka sisältää abstraktin tietotyypin ja tälle tietotyypille soveltuvia aliohjelmia (Decyk et al. 1997a). Kuvassa 5.1 on yksinkertainen henkilöä esittävä esimerkkiluokka Fortran 90:llä toteutettuna.

```
! Chenkilo-luokka
MODULE CHenkilo

    ! Luokan attribuutit
    TYPE Henkilo
        INTEGER :: ika              ! Henkilön ikä
        CHARACTER (LEN=16) :: nimi  ! Henkilön nimi
    END TYPE

    CONTAINS

    ! Luokan rakentaja
    SUBROUTINE Rakentaja(this)
        TYPE (Henkilo), INTENT (OUT) :: this
        this%ika=0
        this%nimi=''
    END SUBROUTINE
END MODULE
```

Kuva 5.1 Fortran 90:llä toteutettu luokka.

Luokka ”CHenkilo” sisältää kaksi attribuuttia: kokonaislukutyyppisen ”ika”-muuttujan ja merkkijonon ”nimi”. Lisäksi luokalla on yksi operaatio ”Rakentaja”, joka alustaa luokan asettamalla ”ika”-muuttujan arvoksi 0 ja ”nimi”-merkkijonon tyhjäksi. Fortran 90:ssä moduulin nimi ei voi olla sama kuin tyypin nimi, joten moduulin nimeen on tässä tapauksessa lisätty C-kirjain (Class). Fortran 90 -kääntäjä ei erottele pieniä ja isoja kirjaimia, mutta käyttämieni Fortran 90 -kielisten esimerkkien selventämiseksi olen kirjoittanut kielen varatut sanat isolla. Fortran 90 -kielen ohjelmakoodin kommentit alkavat huutomerkillä. Kuvassa 5.2 on esimerkkiohjelma, joka käyttää ”CHenkilo”-luokkaa.

```
PROGRAM main
    USE CHenkilo

    TYPE (Henkilo) :: A

    CALL Rakentaja(A)

END PROGRAM
```

Kuva 5.2 Luokan ”CHenkilo” käyttö.

Kuvan 5.2 ohjelmassa “USE”-lauseke esittelee ”CHenkilo”-luokan ohjelman käytettäväksi. Luokan operaatioita ei tarvitse erikseen esitellä ohjelmalle, koska Fortran 90 esittelee ne automaattisesti. Luokan ”CHenkilo” ilmentymä eli olio ”A” luodaan kahdella lausekkeella (kuva 5.3). Ensiksi olio ”A” esitellään ja varataan sen vaatima tila (ensimmäinen rivi). Tämän jälkeen olio ”A” alustetaan kutsumalla ”Rakentaja”-operaatiota (toinen rivi).

```
TYPE (Henkilo) :: A
CALL Rakentaja(A)
```

Kuva 5.3 Olion luonti.

Tämä eroaa esimerkiksi C++-kielestä, jossa olion luonti ja alustus voidaan hoitaa yhdellä kertaa (Decyk et al. 1997b). Luokan ”CHenkilo” operaatiosta ”Rakentaja” kannattaa huomata, että Fortran 90:ssä luokkaa edustava olio esiintyy aina luokan operaatioiden parametrina. Esimerkiksi kuvassa 5.3 luokan ”CHenkilo” olio ”A” edustaa luokkaa ”CHenkilo” ja on sen ”Rakentaja”-operaation parametri. Näin operaation koodi kohdistuu oikeaan olioon. Selvyyden vuoksi olio on luokan operaatioiden ensimmäinen parametri ja sen nimi on ”this”. C++-kielessä puolestaan olion viite on käytettävissä operaatioissa, mutta sitä ei erikseen esitellä (Decyk et al. 1997b). Tällöin esimerkiksi kuvassa 5.3 ”Rakentaja”-operaatiolle ei tarvitsisi erikseen antaa oliota parametrina. Edelleen C++-kielessä olio käsittelee suoraan luokan jäsenmuuttujia, kun taas Fortran 90:ssä luokan attribuutteja käytetään ”this”-parametrin ja ”%”-erikoismerkin avulla. Esimerkiksi C++-kielen lauseke ”ika=0;” olisi Fortran 90 -kielessä muotoa ”this%ika=0” (kuva 5.1).

Luokan ”CHenkilo” ”Rakentaja”-operaatiota voidaan kehittää siten, että jo alustettaessa oliota sen attribuuteille (ikä ja nimi) annetaan arvot. Kuvassa 5.1 esiteltyä ”Rakentaja”-operaatiota voidaan muokata kuvan 5.4 esittämällä tavalla.

```
    ! Luokan rakentaja
    SUBROUTINE Rakentaja(this, ika, nimi)
        TYPE (Henkilo), INTENT (OUT) :: this
        INTEGER, INTENT (IN) :: ika
        CHARACTER (LEN=*), INTENT (IN) :: nimi
        this%ika=ika
        this%nimi=nimi
    END SUBROUTINE
```

Kuva 5.4 Muokattu “Rakentaja”-operaatio.

Myöhemmin esittelen kuinka luokan operaatioita voidaan ylikuormittaa niin, että yhdellä luokalla voi olla useampia erilaisia operaatioita saman nimen alla (katso luku 5.3). Kuvassa 5.4 esittelemäni ”Rakentaja”-operaation ”ika”- ja ”nimi”-muuttujilla on ”INTENT (IN)”-määritys. Tämä määritys kertoo operaatiolle, ettei se voi muokata kyseisiä muuttujia. Kehitetyn ”Rakentaja”-operaation avulla voidaan olio luoda seuraavasti (kuva 5.5):

```
TYPE (Henkilo) :: A
CALL Rakentaja(A, 20, 'Kalle')
```

Kuva 5.5 Kehitetyn “Rakentaja”-operaation käyttö.

Tähän mennessä esitelty ”CHenkilo”-luokka käyttää kooltaan staattista merkkijonoa. Staattisessa tilanvarauksessa tilaa varataan yleensä joko liian vähän tai liian paljon. Fortran 90 mahdollistaa kuitenkin myös dynaamisen tilanvarauksen, jolloin staattisen merkkijonon sijasta voidaan käyttää dynaamista merkkitaulukkoa, jonka koko määritellään ohjelman suorituksen aikana. Fortran 90:ssä dynaaminen merkkitaulukko voidaan esitellä seuraavasti (kuva 5.6):

```
! Merkkitaulukon esittely
CHARACTER, DIMENSION (:), POINTER :: nimi
...
ALLOCATE(nimi(LEN('Kalle')))
...
DEALLOCATE(nimi)
```

Kuva 5.6 Dynaamisesti varatun merkkitaulukon esittely, varaaminen ja vapauttaminen.

Kuvan 5.6 dynaamisen merkkitaulukon tilanvaraus tehdään Fortran 90:ssä ”ALLOCATE”-nimisellä standardialiohjelmalla. Varattu merkkitaulukko puolestaan vapautetaan ”DEALLOCATE”-nimisellä standardialiohjelmalla. Kuvassa 5.6 on esimerkki merkkitaulukon varaamisesta ja vapauttamisesta.

C-kielen tyyppisten merkkitaulukoiden käyttäminen Fortran 90:ssä rajaa pois osan Fortran 90 -kielen merkkijonoja käyttävistä standardialiohjelmista. Esimerkiksi kuvan 5.6 ”nimi”-merkkitaulukon alustus merkkijonolla pitäisi tehdä lausekkeen yksi mukaisesti (kuva 5.7).

```
! Lauseke 1. Merkkijonon sijoitus merkkitaulukkoon
nimi1=(/'K','a','l','l','e'/)

! Lauseke 2. Merkkijonon sijoitus merkkijonoon
nimi2='Kalle'
```

Kuva 5.7 Merkkijonon sijoitus merkkitaulukkoon ja merkkijonoon Fortran 90:llä.

Käytettäessä Fortran 90 -kielen merkkijonoa sijoitus voitaisiin tehdä lausekkeen kaksi mukaisesti (kuva 5.7). Fortran 90 -kielessä sijoittamista voidaan yksinkertaistaa toteuttamalla operaatio ”Strcpy”, joka sijoittaa merkkijonon dynaamiseen merkkitaulukkoon. Kuvassa 5.8 on ”Strcpy”-operaation toteutus.

```
    ! Merkkijonon kopiointi merkkitaulukkoon
    SUBROUTINE CHenkilo_Strcpy(mtaulu, mjono)
        CHARACTER, DIMENSION (:), POINTER :: mtaulu
        CHARACTER (LEN=*), INTENT (IN) :: mjono
        INTEGER :: i

        DO i=1, MAX(LEN(mjono), SIZE(mtaulu))
            mtaulu(i)=mjono(i:i)
        END DO
    END SUBROUTINE
```

Kuva 5.8 Operaatio ”Strcpy”, joka sijoittaa merkkijonon merkkitaulukkoon.

Korvattaessa luokan ”CHenkilo” staattinen merkkijono dynaamisesti varatulla merkkitaulukolla tarvitaan luokassa myös ”Purkaja”-operaatiota, joka vapauttaa dynaamisesti varatun merkkitaulukon. Lisäksi luokkaan lisätään kuvan 5.8 ”Strcpy”-operaatio, jota ”Rakentaja”-operaatio hyödyntää.

```
! CHenkilo-luokka
MODULE CHenkilo

    ! Luokan attribuutit
    TYPE Henkilo
        INTEGER :: ika                                  ! Henkilön ikä
        CHARACTER, DIMENSION (:), POINTER :: nimi   ! Henkilön nimi
    END TYPE

    CONTAINS

    ! Luokan rakentaja
    SUBROUTINE Rakentaja(this, ika, nimi)
        TYPE (Henkilo), INTENT (OUT) :: this
        INTEGER, INTENT (IN) :: ika
        CHARACTER (LEN=*), INTENT (IN) :: nimi

        ! Muistin varaaminen
        ALLOCATE(this%nimi(LEN(nimi)))

        this%ika=ika
        CALL Strcpy(this%nimi, nimi)
    END SUBROUTINE

    ! Luokan purkaja
    SUBROUTINE Purkaja(this)
        TYPE (Henkilo), INTENT (OUT) :: this

        ! Muistin vapauttaminen
        DEALLOCATE(this%nimi)
    END SUBROUTINE

    ! Merkkijonon kopiointi merkkitaulukkoon
    SUBROUTINE Strcpy(mtaulu, mjono)
        CHARACTER, DIMENSION (:), POINTER :: mtaulu
        CHARACTER (LEN=*), INTENT (IN) :: mjono
        INTEGER :: i

        DO i=1, MAX(LEN(mjono), SIZE(mtaulu))
            mtaulu(i)=mjono(i:i)
        END DO
    END SUBROUTINE
END MODULE
```

Kuva 5.9 ”CHenkilo”-luokka dynaamisella merkkitaulukolla ja uusilla operaatioilla.

Mikäli sovelluksessa käytetään runsaasti merkkitaulukoita, on järkevintä sijoittaa merkkitaulukoita käsittelevät operaatiot omaan merkkitaulukkoluokkaansa. Tässä esimerkissä ”Strcpy”-operaatio sisällytetään kuitenkin yksinkertaisuuden vuoksi ”CHenkilo”-luokkaan. Kuvassa 5.9 on toteutettu ”CHenkilo”-luokka dynaamisella merkkitaulukolla hyödyntäen ”Strcpy”- ja ”Purkaja”-operaatioita.

5.3 Operaatioiden ylikuormittaminen Fortran 90:ssä

Operaatioiden ylikuormittaminen Fortran 90:ssä tapahtuu ”INTERFACE”-lohkon avulla. ”INTERFACE”-lohko on yksi geneerisen aliohjelman osa (katso 5.1). ”INTERFACE”-lohko täytyy esitellä ennen moduulin ”CONTAINS”-lausetta, joka määrittää moduulin toteutuksen alkamiskohdan. Esimerkiksi luokkaan ”CHenkilo” voitaisiin lisätä operaatio ”Vanhene”, jolla olisi kaksi erilaista toteutusta; parametriton ja parametrillinen. Ilman annettua parametria ”Vanhene”-operaatio käyttää parametritonta aliohjelmaa, joka kasvattaa ”CHenkilo”-luokan ”ika”-muuttujan arvoa yhdellä. Kokonaislukuparametrin kanssa ”Vanhene”-operaatio käyttää parametrillista aliohjelmaa, joka lisää ”ika”-muuttujaan parametrin ilmoittaman määrän. Fortran 90:ssä tällainen operaation ylikuormitus vaatii kaksi vaihetta. Ylikuormitettava operaatio täytyy esitellä ”INTERFACE”-lohkossa, jonka jälkeen päästään itse toteutukseen. C++-kielessä sitä vastoin riittäisi samannimisen operaation esittely eri parametreilla (Decyk et al. 1997b). Kuvassa 5.10 on ylikuormitetun ”Vanhene”-operaation esittely ”INTERFACE”-lohkossa ja sen toteutus.

```
INTERFACE Vanhene
    MODULE PROCEDURE CHenkilo_Vanhene
    MODULE PROCEDURE CHenkilo_Vanhene_Int
END INTERFACE

...

! Lisää ika-attribuuttiin 1
SUBROUTINE CHenkilo_Vanhene(this)
    TYPE (Henkilo), INTENT (OUT) :: this

    this%ika=this%ika+1
END SUBROUTINE

! Lisää ika-attribuuttiin arvo
SUBROUTINE CHenkilo_Vanhene_Int(this, arvo)
    TYPE (Henkilo), INTENT (OUT) :: this
    INTEGER, INTENT (IN) :: arvo

    this%ika=this%ika+arvo
END SUBROUTINE
```

Kuva 5.10 Ylikuormitettu operaatio “Vanhene”.

5.4 Operaatioiden parametrien oletusarvot Fortran 90:ssä

Fortran 90 ei mahdollista C++-kielen tapaan operaatioiden parametrien oletusarvojen automaattista asettamista, silloin kun parametreja ei anneta jo operaation kutsussa. Fortran 90 sisältää kuitenkin valinnaisia parametreja, joiden avulla osalle parametreista voidaan määrittää oletusarvoja (Decyk et al. 1997b). Fortran 90:n standardialiohjelmalla ”PRESENT” voidaan tarkistaa, onko operaatiolle jo annettu parametri vai käytetäänkö määriteltyä oletusarvoa.

```
! Tulostaa attribuutit
SUBROUTINE CHenkilo_Tulosta(this, tulosta_ika)
    TYPE (Henkilo), INTENT (IN) :: this
    LOGICAL, OPTIONAL, INTENT (IN) :: tulosta_ika

    WRITE (*,*) 'Henkilön nimi: ', this%nimi

    ! Tarkista onko parametri annettu
    IF(PRESENT(tulosta_ika)) THEN
        ! Tarkista parametrin arvo
        IF(tulosta_ika==.TRUE.) THEN
            WRITE (*,*) 'Ikä on: ', this%ika
        END IF
    END IF
END SUBROUTINE
```

Kuva 5.11 ”Tulosta”-operaatio, jolla on valinnainen boolean-parametri.

Kuvassa 5.11 on operaatio ”Tulosta”. Tämä operaatio tulostaa ”CHenkilo”-luokan ”nimi”-muuttujan sisällön sekä ”ika”-muuttujan arvon, mikäli valinnainen boolean-parametri ”tulosta_ika” on annettu ja se on tosi.

5.5 Luokan suojaus Fortran 90:ssä

“CHenkilo”-luokkaa voidaan edelleen kehittää lisäämällä siihen C++-kielen tyyppisiä suojausominaisuuksia. Edellä luvussa 5.1 esitellyn Fortran 90 -kielen moduulin eli luokan eri osien suojaus voidaan luokitella yksityiseksi tai julkiseksi. Kuten C++-kielessä, yksityiset osat ovat näkyviä vain luokan omille jäsenille ja julkiset osat taas näkyvät myös luokan ulkopuolelle. C++-kielen ”protected”-suojausta ei ole Fortran 90:ssä (Decyk et al. 1997b). Fortran 90 -luokan attribuutteja ei voi määritellä erikseen julkisiksi tai yksityisiksi, vaan kaikkien luokan attribuuttien täytyy olla jompaakumpaa (samaa) tyyppiä. Kuitenkin luokan operaatiot voidaan erikseen määritellä julkisiksi tai yksityisiksi (Haataja et al. 1998). ”CHenkilo”-luokkaa voidaan kehittää määrittämällä sen attribuutit yksityisiksi, jolloin niitä ei voi muokata luokan ulkopuolelta, sekä määrittämällä ylikuormitetun ”Vanhene”-operaation aliohjelmat yksityisiksi. Tällöin ”Vanhene”-operaatio on julkinen, mutta sen sisältämät aliohjelmat yksityisiä ja siis

käyttäjälle näkymättömissä. Kuvassa 5.12 on esimerkki operaatioiden ja attribuuttien määrittämisestä yksityisiksi.

```
! Yksityiset operaatiot
PRIVATE :: CHenkilo_Vanhene, CHenkilo_Vanhene_Int

...

! Luokan jäsenet
TYPE Henkilo
    PRIVATE
    INTEGER :: ika                                  ! Henkilön ikä
    CHARACTER, DIMENSION (:), POINTER :: nimi   ! Henkilön nimi
END TYPE
```

Kuva 5.12 Yksityiseksi määrittäminen.

”CHenkilo”-luokan attribuutit ovat suojattuja (yksityisiä), joten attribuuttien arvot luetaan ”CHenkilo”-luokan määrittelemien operaatioiden avulla. ”CHenkilo”-luokkaan voidaan lisätä kaksi operaatiota, joista toinen palauttaa ”ika”-attribuutin arvon ja toinen ”nimi”-attribuutin merkkijonona. Kuvassa 5.13 on ”CHenkilo”-luokan attribuutteja palauttavien operaatioiden toteutus.

```
INTEGER FUNCTION Ika(this)
    TYPE (Henkilo), INTENT (IN) :: this

    Ika=this%ika
END FUNCTION

FUNCTION Nimi(this) RESULT(mjono)
    TYPE (Henkilo), INTENT (IN) :: this
    CHARACTER (LEN=SIZE(this%nimi)) :: mjono

    DO i=1, SIZE(this%nimi)
        mjono(i:i)=this%nimi(i)
    END DO
END FUNCTION
```

Kuva 5.13 ”CHenkilo”-luokan attribuutteja palauttavat operaatiot.

Lopullinen versio kehitetystä ”CHenkilo”-luokasta on liitteessä yksi, ja liitteessä kaksi on vertailun vuoksi C++-kielellä toteutettu vastaava luokka. Kuvassa 5.14 on esimerkkiohjelma, joka hyödyntää ”CHenkilo”-luokkaa. Ohjelma esittelee ja luo kaksi oliota nimeltä ”A” ja ”B”, jotka sen jälkeen alustetaan ja tulostetaan. Tulostuksen jälkeen olioiden ”ika”-attribuuttia kasvatetaan ja toistetaan tulostus. Sen jälkeen tulostetaan ”A”-olion ”ika”-attribuutti ja ”B”-olion ”nimi”-attribuutti. Lopuksi kutsutaan ”CHenkilo”-luokan ”Purkaja”-operaatiota molemmille olioille.

```
PROGRAM main
    USE CHenkilo

    TYPE (Henkilo) :: A, B

    CALL Rakentaja(A, 20, 'Kalle')
    CALL Rakentaja(B, 15, 'Pasi')
    CALL Tulosta(A, .TRUE.)
    CALL Tulosta(B, .TRUE.)
    CALL Vanhene(A)
    CALL Vanhene(B, 5)
    CALL Tulosta(A, .TRUE.)
    CALL Tulosta(B, .TRUE.)
    WRITE (*,*) Ika(A)
    WRITE (*,*) Nimi(B)
    CALL Purkaja(A)
    CALL Purkaja(B)
END PROGRAM
```

Kuva 5.14 “CHenkilo”-luokkaa hyödyntävä esimerkkiohjelma.

5.6 Periytyminen Fortran 90:ssä

Fortran 90 ei suoranaisesti tue periytymistä, mutta periytymistä vastaava rakenne voidaan kuitenkin toteuttaa Fortran 90:ssä (Decyk et al. 1998). Luokan attribuutit periytetään esittelemällä yliluokan tietotyyppi aliluokan attribuuttina. Esimerkiksi, jos ”COpiskelija”-luokka periytyy ”CHenkilo”-luokasta, niin ”COpiskelija”-luokan attribuutit esitetään seuraavasti (kuva 5.15):

```
! Luokan attribuutit
TYPE Opiskelija
    TYPE (Henkilo) :: henkilo                    ! Perityt ominaisuudet
    INTEGER :: kursseja                          ! Monellako kurssilla
    CHARACTER (LEN=16), DIMENSION(10) :: kurssit ! Kurssien nimet
END TYPE
```

Kuva 5.15 ”COpiskelija”-luokka perii ”CHenkilo”-luokan attribuutit.

”COpiskelija”-luokka lisää ”CHenkilo”-luokasta perittyihin attribuutteihin kaksi attribuuttia. Nämä attribuutit ovat kokonaislukumuuttuja ”kursseja”, joka kertoo monellako kurssilla opiskelija on, ja staattinen merkkijonotaulukko ”kurssit”, joka sisältää kurssien nimet. C++-kielessä ei periytymisen yhteydessä tarvitse erikseen esitellä yliluokan perittäviä attribuutteja aliluokassa, vaan ne sisällytetään automaattisesti aliluokkaan (Decyk et al. 1997b). Alustettaessa Fortran 90 -kielellä ”COpiskelija”-luokan oliota, alustetaan aluksi yliluokka ja sen jälkeen luokan omat attribuutit. Kuvassa 5.16 on ”COpiskelija”-luokan ”Rakentaja”-operaation toteutus.

```
    SUBROUTINE COpiskelija_Rakentaja(this, ika, nimi)
        TYPE (Opiskelija), INTENT (OUT) :: this
        INTEGER, INTENT (IN) :: ika
        CHARACTER (LEN=*), INTENT (IN) :: nimi

        ! Yliluokan attribuuttien alustus
        CALL Rakentaja(this%henkilo, ika, nimi)

        this%kursseja=0
    END SUBROUTINE
```

Kuva 5.16 ”COpiskelija”-luokan ”Rakentaja”-operaation toteutus.

Tämä Fortran 90 -kielen rakenne jäljittelee C++-kielen rakentajien kutsua, jossa luokkahierarkian rakentajien kutsut suoritetaan ylhäältä alas järjestyksessä, aloittaen siis periytymishierarkian ylimmältä tasolta (Decyk et al. 1997b). Kuvassa 5.17 on esitetty alkeellinen ”COpiskelija”-luokan toteutus, joka perii ”CHenkilo”-luokan.

```
! Copiskelija-luokka
MODULE COpiskelija
    USE CHenkilo

    PRIVATE :: COpiskelija_Rakentaja, COpiskelija_Purkaja

    ! Luokan attribuutit
    TYPE Opiskelija
        TYPE (Henkilo) :: henkilo                    ! Perityt ominaisuudet
        INTEGER :: kursseja                          ! Monellako kursilla
        CHARACTER (LEN=16), DIMENSION(10) :: kurssit ! Kurssien nimet
    END TYPE

    INTERFACE Rakentaja
        MODULE PROCEDURE COpiskelija_Rakentaja
    END INTERFACE

    INTERFACE Purkaja
        MODULE PROCEDURE COpiskelija_Purkaja
    END INTERFACE

    CONTAINS

    ! Luokan rakentaja
    SUBROUTINE COpiskelija_Rakentaja(this, ika, nimi)
        TYPE (Opiskelija), INTENT (OUT) :: this
        INTEGER, INTENT (IN) :: ika
        CHARACTER (LEN=*), INTENT (IN) :: nimi

        ! Yliluokan attribuuttien alustus
        CALL Rakentaja(this%henkilo, ika, nimi)

        this%kursseja=0
    END SUBROUTINE

    ! Luokan purkaja
    SUBROUTINE COpiskelija_Purkaja(this)
        TYPE (Opiskelija), INTENT (OUT) :: this

        ! Yliluokan tuhoaminen
        CALL Purkaja(this%henkilo)
    END SUBROUTINE
END MODULE
```

Kuva 5.17 Alkeellinen “COpiskelija”-luokka, joka perii ”CHenkilo”-luokan.

C++-kielessä purkajien kutsut suoritetaan päinvastaisessa järjestyksessä kuin rakentajien kutsut, eli alhaalta ylös. Fortran 90:ssä purku tapahtuu samalla tavalla. Vastaavasti kuin ”COpiskelija”-luokan rakentajassa kutsutaan ”CHenkilo”-luokan rakentajaa, voidaan ”CHenkilo”-luokan osa ”COpiskelija”-luokassa tuhota ”COpiskelija”-luokan purkajassa kutsumalla ”CHenkilo”-luokan purkajaa.

Kuvan 5.17 esimerkissä ”USE”-lauseke periyttää ”CHenkilo”-luokan. ”COpiskelija”-luokan ”Rakentaja”- ja ”Purkaja”-operaatiot ovat samannimisiä kuin ”CHenkilo”-luokan vastaavat operaatiot. Samannimiset operaatiot eivät kuitenkaan muodosta ongelmaa, koska operaatiot ovat riippuvaisia oliosta. Kuvassa 5.18 on esimerkki ”COpiskelija”-luokan ”A”-olion luomisesta ja tuhoamisesta.

```
PROGRAM main
    USE COpiskelija

    TYPE (Opiskelija) :: A

    CALL Rakentaja(A, 25, "Simo")
    CALL Purkaja(A)
END PROGRAM
```

Kuva 5.18 “COpiskelija”-luokan ”A”-olion luonti ja tuhoaminen.

C++-kielessä luokan operaatioiden periytyminen on yksinkertaisempaa. Mikäli aliluokka ei tee muutoksia yliluokasta perittyyn operaatioon, ei aliluokan tarvitse tehdä operaatiolle mitään, ei edes esitellä sitä erikseen. C++-kielessä aliluokka käyttää suoraan yliluokan operaatioita, vaikka operaation kutsu kohdistuukin aliluokkaan. Mikäli aliluokka muokkaa yliluokasta perittyä operaatiota, on muunnellun operaation itsensä kutsuttava yliluokan operaatiota, jos siihen on tarvetta. Fortran 90:ssä peritty operaatio on aina esiteltävä riippumatta siitä muokataanko perittyä operaatiota vai ei (Decyk et al. 1997b). Fortran 90:ssä luokalla ei nimittäin ole kyseistä operaatiota, jos operaatiota ei ole esitelty aliluokassa. Lisäksi aliluokkaan perityn operaation on kutsuttava yliluokan operaatiota, vaikka perittyä operaatiota ei olisi muokattu aliluokassa (kuva 5.19). Tätä ei siis tarvitse tehdä C++-kielessä. Kuvassa 5.20 on luokan ”COpiskelija” ”Tulosta”-operaatio, jonka toiminnallisuutta olen laajentanut niin, että se tulostaa sekä ”CHenkilo”- että ”COpiskelija”-luokan tiedot. Aluksi kutsutaan ”CHenkilo”-yliluokan ”Tulosta”-operaatiota ja sen jälkeen suoritetaan luokan oma toteutus.

```
    SUBROUTINE COpiskelija_Tulosta(this, ika)
        TYPE (Opiskelija), INTENT (IN) :: this
        LOGICAL, OPTIONAL, INTENT (IN) :: ika

        CALL Tulosta(this%henkilo, ika)
    END SUBROUTINE
```

Kuva 5.19 Peritty operaatio, jota ei ole muokattu.

```
    SUBROUTINE COpiskelija_Tulosta(this, ika)
        TYPE (Opiskelija), INTENT (IN) :: this
        LOGICAL, OPTIONAL, INTENT (IN) :: ika

        CALL Tulosta(this%henkilo, ika)

        WRITE (*,*) Nimi(this%henkilo), ' on opiskelija'

        IF(this%kursseja<=0) THEN
            WRITE (*,*) Nimi(this%henkilo), ' ei opiskele'
        ELSE
            WRITE (*,*) Nimi(this%henkilo), ' on ', this%kursseja, ' kurssilla'
            WRITE (*,*) 'Kurssit:'

            DO i=1, this%kursseja
                WRITE (*,*) this%kurssit(i)
            END DO
        END IF
    END SUBROUTINE
```

Kuva 5.20 Peritty operaatio, jonka toiminnallisuutta on lisätty.

”COpiskelija”-luokassa täytyy olla mahdollisuus lisätä opiskelijan kurssien lukumäärää, jotta se olisi käytännöllisempi. Kuvassa 5.21 on operaatio, joka saa parametrinaan kurssin nimen ja lisää sen opiskelijan kursseihin. Operaatio lisää yksinkertaisesti kurssien lukumäärää yhdellä ja lisää kurssin nimen, joka annetaan siis parametrina, merkkijonotaulukkoon kurssien lukumäärän osoittamaan alkioon.

```
    SUBROUTINE COpiskelija_LisaaKurssi(this, kurssi)
        TYPE (Opiskelija), INTENT (INOUT) :: this
        CHARACTER (LEN=*), INTENT (IN) :: kurssi

        this%kursseja=this%kursseja+1
        this%kurssit(this%kursseja)=kurssi
    END SUBROUTINE
```

Kuva 5.21 Operaatio, joka lisää opiskelijalle kurssin.

Kuvassa 5.22 on luokan ”COpiskelija” testiohjelma ja sen tuottama tuloste. Liitteessä kolme on ”COpiskelija”-luokan lopullinen toteutus Fortran 90 -kielellä ja lisäksi liitteessä neljä on vertailun vuoksi C++-kielellä toteutettu vastaava luokka

```
! Testiohjelma
PROGRAM main
    USE COpiskelija

    TYPE (Opiskelija) :: A

    CALL Rakentaja(A, 25, "Simo")
    CALL LisaaKurssi(A, "Matematiikka")
    CALL LisaaKurssi(A, "Maantiede")
    CALL Tulosta(A, .TRUE.)
    CALL Purkaja(A)
END PROGRAM


--- Ohjelman tuottama tuloste ---
 Henkilön nimi: Simo
 Ikä on:           25
 Simo on opiskelija
 Simo on            2 kurssilla
 Kurssit:
 Matematiikka
 Maantiede
```

Kuva 5.22 “COpiskelija”-luokan testiohjelma ja ohjelman tuloste.

Tähän mennessä olen esitellyt periytymishierarkian, joka ei sisällä virtuaalisia operaatioita. Olen luonut uuden ”COpiskelija”-luokan, joka sisältää ”CHenkilo”-yliluokan attribuutit, sekä laajentanut aliluokan sisältämään yliluokan operaatiot. Tärkeä havainto on, että aliluokan ei tarvitse tietää yliluokan toteutusta, vaan sille riittää yliluokan rajapinnan esittely. Tähän mennessä esittämästäni periytymishierarkiasta puuttuvat vielä monimuotoisuus ja myöhäinen sidonta, joiden toteutuksen Fortran 90 -kielellä esittelen seuraavassa luvussa.

Kuten ”CHenkilo”-luokan kohdalla, voitaisiin myös ”COpiskelija”-luokan merkkijonotaulukko korvata dynaamisia merkkitaulukoita sisältävällä taulukolla. En katsonut sitä tässä esimerkissä selkeyden vuoksi tarpeelliseksi. Mikäli ”CHenkilo”-luokan ilmentymien luonti haluttaisiin estää, olisi siitä muodostettava abstrakti luokka, kuten esimerkiksi C++-kielessä, mutta tämä ei ole mahdollista Fortran 90:ssä (Decyk et al. 1997b).

”CHenkilo”-luokasta voidaan periä toinenkin luokka samalla tavoin kuin ”COpiskelija”-luokka. Looginen pari opiskelijalle on tietysti opettaja, josta tehdään luokka ”COpettaja”. ”COpettaja”-luokka perii ”CHenkilo”-luokan ja lisää attribuutteihinsa yhden kokonaislukumuuttujan, joka kuvaa opettajan palkkaa (kuva 5.23).

```
    ! Luokan attribuutit
    TYPE Opettaja
        TYPE (Henkilo) :: henkilo   ! Perityt ominaisuudet
        INTEGER :: palkka           ! Opettajan palkka
    END TYPE
```

Kuva 5.23 “COpettaja”-luokan attribuutit.

Luokan “COpettaja” ”Rakentaja”-operaatiota muokataan lisäämällä uusi parametri, opettajan palkka. Luokalle “COpettaja” lisätään myös operaatio “PaivitaPalkka”, joka päivittää opettajan palkan. Vielä ”COpettaja”-luokan ”Tulosta”-operaatio muokataan tulostamaan ”CHenkilo”-luokan tulostuksen lisäksi ”COpettaja”-luokan ”palkka”-muuttujan arvo. ”COpettaja”-luokan koko toteutus on liitteessä viisi ja liitteessä kuusi on vastaava C++-kielinen toteutus. Kuvassa 5.24 on testiohjelma, joka käyttää sekä ”COpettaja”- että ”COpiskelija”-luokkaa. Lisäksi kuvassa on ohjelman tulostus. Käytän ”COpettaja”- ja ”COpiskelija”-luokkia vielä seuraavassa luvussa (5.7).

```
! Testiohjelma
PROGRAM main
    USE COpiskelija
    USE COpettaja

    TYPE (Opiskelija) :: OpiskelijaA
    TYPE (Opettaja) :: OpettajaA

    CALL Rakentaja(OpiskelijaA, 25, "Simo")
    CALL Rakentaja(OpettajaA, 42, "Jukka", 2000)

    CALL LisaaKurssi(OpiskelijaA, "Biologia")
    CALL LisaaKurssi(OpiskelijaA, "Maantiede")
    CALL LisaaKurssi(OpiskelijaA, "Kemia")

    CALL Tulosta(OpiskelijaA, .TRUE.)
    WRITE (*,*)
    CALL Tulosta(OpettajaA, .FALSE.)

    CALL Purkaja(OpiskelijaA)
    CALL Purkaja(OpettajaA)
END PROGRAM


--- Ohjelman tuottama tuloste ---
 Henkilön nimi: Simo
 Ika on:           25
 Simo on opiskelija
 Simo on            3 kurssilla
 Kurssit:
 Biologia
 Maantiede
 Kemia

 Henkilön nimi: Jukka
 Jukka on opettaja
 Palkka:         2000
```

Kuva 5.24 Esimerkkiohjelma “COpettaja”- ja “COpiskelija”-luokkien käytöstä.

5.7 Monimuotoisuus ja myöhäinen sidonta Fortran 90:ssä

Monimuotoisuus ja myöhäinen sidonta ovat olennaisia olio-ohjelmoinnin käsitteitä. Luvussa 5.3 käsittelin periytymistä ja kuvasin kuinka samanniminen operaatio toimii eri tavoin riippuen parametrina annetusta oliosta. Monimuotoisuuden avulla voidaan yhdellä viitteellä viitata kaikkiin samassa periytymishierarkiassa oleviin olioihin. Kutsuttaessa operaatiota monimuotoisella viitteellä ohjelman suorituksen aikana, ei viitteen ominaisuuksia välttämättä tunneta. Tällöin sovelletaan myöhäistä sidontaa, ja oikea operaatio ratkaistaan vasta ohjelman suorituksen aikana.

Fortran 90:ssä monimuotoisuuden ja myöhäisen sidonnan toteuttaminen edellyttää kahta asiaa (Decyk et al. 1998). Ensinnäkin tarvitaan monimuotoinen osoitin-olio, joka pystyy osoittamaan mihin tahansa periytymishierarkian olioon. Toiseksi tarvitaan operaation valintamenetelmä, joka valitsee oikean operaation osoitin-olion osoittamalle oliolle. C++-kielessä monimuotoisuus ja myöhäinen sidonta ovat automaattisesti käytössä, mutta Fortran 90:ssä joudutaan erikseen toteuttamaan monimuotoinen luokka, joka vastaa monimuotoisuudesta ja myöhäisestä sidonnasta (Decyk et al. 1997b).

Kuvassa 5.25 ovat monimuotoisen ”CPoly”-luokan attribuutit. Monimuotoisen luokan attribuutit koostuvat pelkästään osoittimista periytymishierarkiassa oleviin luokkiin (Decyk et al. 1998). Kuvan 5.25 tapauksessa osoittimet osoittavat luvun 5.3 luokkiin ”CHenkilo”, ”COpiskelija” ja ”COpettaja”.

```
    TYPE Osoitin
        TYPE (Henkilo), POINTER :: henkilo       ! Osoitin henkilo-olioon
        TYPE (Opiskelija), POINTER :: opiskelija ! Osoitin opiskelijaolioon
        TYPE (Opettaja), POINTER :: opettaja     ! Osoitin opettajaolioon
    END TYPE
```

Kuva 5.25 Monimuotoisen ”CPoly”-luokan attribuutit.

Ohjelman suorituksen aikana ”CPoly”-olio voidaan asettaa osoittamaan yhteen ”CHenkilo”-, ”COpiskelija”- tai ”COpettaja”-luokan osoittimista (kuva 5.25). Osoitinta asetettaessa ”CPoly”-olio nollaa muut osoittimet, jolloin varmistetaan, että ”CPoly”-olio osoittaa vain yhteen olioon kerrallaan. Kuvassa 5.26 on esimerkki siitä kuinka tämä voidaan toteuttaa.

```
TYPE (Opiskelija), TARGET :: OpiskelijaA
TYPE (Osoitin) :: os

CALL Rakentaja(OpiskelijaA, 23, 'Kalle')

! Aseta monimuotoinen olio osoittamaan opiskelijaolioon
os%opiskelija=>OpiskelijaA

! Aseta muut vaihtoehdot tyhjiksi
NULLIFY(os%henkilo)
NULLIFY(os%opettaja)
```

Kuva 5.26 Esimerkki osoittimien asettamisesta.

Fortran 90 -kielessä käytetään osoitinsijoitukseen merkintää “=>” ja osoittimen kohteella täytyy olla “TARGET” ominaisuus. Fortran 90:n standardialiohjelma ”NULLIFY” alustaa osoittimen nollattuun tilaan, jossa se ei osoita mihinkään kohteeseen (Haataja et al., 1998). Kuvassa 5.26 esitellyt sijoitusoperaatiot voidaan kätevästi kapseloida kuvan 5.27 operaatioon. Kuvan 5.27 operaatiolla ”COpiskelija”-olio sijoitetaan monimuotoiseen olioon kuvan 5.28 osoittamalla tavalla.

```
FUNCTION Sijoita_Opiskelija(os_opiskelija) RESULT(os)
    TYPE (Osoitin) :: os
    TYPE (Opiskelija), TARGET, INTENT(IN) :: os_opiskelija

    os%opiskelija=>os_opiskelija
    NULLIFY(os%henkilo)
    NULLIFY(os%opettaja)
END FUNCTION
```

Kuva 5.27 Operaatio, joka sijoittaa ”COpiskelija”-olion monimuotoiseen olioon.

```
CALL Rakentaja(OpiskelijaA, 23, 'Kalle')
os=Sijoita_Opiskelija(OpiskelijaA)
```

Kuva 5.28 Esimerkki ”COpiskelija”-olion sijoittamisesta monimuotoiseen olioon.

Kuvan 5.27 operaatiosta voidaan samalla mallilla luoda vastaavat operaatiot ”CHenkilo” ja ”COpettaja”-luokille. Lopuksi kaikkien olioiden sijoitusoperaatiot voidaan kuormittaa yhden ”Poly”-operaation alle käyttämällä ”INTERFACE”-lohkoa. Tämän jälkeen voidaan mikä tahansa ”CPoly”-luokalle kelpaava olio sijoittaa ”CPoly”-luokkaan yhden ainoan operaation kautta.

Olion sijoituksen lisäksi ”CPoly”-luokassa tarvitaan operaatioiden valintamekanismi. Tällöin vertaillaan ”CPoly”-luokan osoittimia ja tarkistetaan mikä osoittimista on aktiivinen (Decyk et al. 1998). Kutsuttava operaatio saa parametrinaan aktiivisen osoittimen, jolloin operaatio kohdistuu oikealle oliolle. Kuvassa 5.29 on esimerkki ”Tulosta”-operaatiosta, joka löytyy luokista ”CHenkilo”-, ”COpiskelija”- ja ”COpettaja”-luokasta. Näiden luokkien ”Tulosta”-operaation parametrit ovat samoja, joten yksi operaatio riittää kaikille kolmelle luokalle.

```
SUBROUTINE CPoly_Tulosta(this, ika)
    TYPE (Osoitin) :: this
    LOGICAL, OPTIONAL, INTENT (IN) :: ika

    ! Tarkista aktiivinen osoitin ja kutsu
    ! ”Tulosta”-operaatiota aktiivisella osoittimella.
    IF(ASSOCIATED(this%henkilo)) THEN
        CALL Tulosta(this%henkilo, ika)
    ELSE IF(ASSOCIATED(this%opiskelija)) THEN
        CALL Tulosta(this%opiskelija, ika)
    ELSE IF(ASSOCIATED(this%opettaja)) THEN
        CALL Tulosta(this%opettaja, ika)
    END IF
END SUBROUTINE
```

Kuva 5.29 Operaation valinta aktiivisen osoittimen perusteella.

Edellä esiteltyjen ominaisuuksien avulla voidaan toteuttaa ”CPoly”-luokka, joka vastaa monimuotoisuudesta ja myöhäisestä sidonnasta Fortran 90:ssä. Kuvassa 5.30 on yksinkertainen esimerkki ”CPoly”-luokan toteutuksesta.

”CPoly”-luokan avulla voidaan monimuotoiseen olioon sijoittaa olio ja kutsua muun muassa ”Tulosta”-operaatiota monimuotoiselle oliolle. Liitteessä seitsemän on ”CPoly”-luokan koko toteutus, josta löytyvät ”CHenkilo”-, COpiskelija”- ja ”COpettaja”-luokan operaatiot ”Rakentaja”- ja ”Purkaja”-operaatiota lukuun ottamatta. Näin monimuotoisella viitteellä voidaan siis kutsua minkä tahansa samassa periytymishierarkiassa olevan olion operaatioita.

```
MODULE CPoly
    USE COpiskelija
    USE COpettaja

    PRIVATE :: Sijoita_Henkilo, Sijoita_Opiskelija, Sijoita_Opettaja, CPoly_Tulosta

    TYPE Osoitin
        TYPE (Henkilo), POINTER :: henkilo       ! Osoitin henkilo-olioon
        TYPE (Opiskelija), POINTER :: opiskelija ! Osoitin opiskelijaolioon
        TYPE (Opettaja), POINTER :: opettaja     ! Osoitin opettajaolioon
    END TYPE

    INTERFACE Poly
        MODULE PROCEDURE Sijoita_Henkilo, Sijoita_Opiskelija, Sijoita_Opettaja
    END INTERFACE

    INTERFACE Tulosta
        MODULE PROCEDURE CPoly_Tulosta
    END INTERFACE

    CONTAINS

    FUNCTION Sijoita_Henkilo(os_henkilo) RESULT(os)
        TYPE (Osoitin) :: os
        TYPE (Henkilo), TARGET, INTENT(IN) :: os_henkilo

        os%henkilo=>os_henkilo
        NULLIFY(os%henkilo)
        NULLIFY(os%opettaja)
    END FUNCTION

    FUNCTION Sijoita_Opiskelija(os_opiskelija) RESULT(os)
        TYPE (Osoitin) :: os
        TYPE (Opiskelija), TARGET, INTENT(IN) :: os_opiskelija

        NULLIFY(os%henkilo)
        os%opiskelija=>os_opiskelija
        NULLIFY(os%opettaja)
    END FUNCTION

    FUNCTION Sijoita_Opettaja(os_opettaja) RESULT(os)
        TYPE (Osoitin) :: os
        TYPE (Opettaja), TARGET, INTENT(IN) :: os_opettaja

        NULLIFY(os%henkilo)
        NULLIFY(os%opiskelija)
        os%opettaja=>os_opettaja
    END FUNCTION

    SUBROUTINE CPoly_Tulosta(this, ika)
        TYPE (Osoitin) :: this
        LOGICAL, OPTIONAL, INTENT (IN) :: ika

        IF(ASSOCIATED(this%henkilo)) THEN
            CALL Tulosta(this%henkilo, ika)
        ELSE IF(ASSOCIATED(this%opiskelija)) THEN
            CALL Tulosta(this%opiskelija, ika)
        ELSE IF(ASSOCIATED(this%opettaja)) THEN
            CALL Tulosta(this%opettaja, ika)
        END IF
    END SUBROUTINE
END MODULE
```

Kuva 5.30 Yksinkertaisen “CPoly”-luokan toteutus.

Kuvassa 5.31 on esimerkkisovellus, joka hyödyntää sekä ”CPoly”-luokkaa että ”COpiskelija”- ja ”COpettaja”-aliluokkia. Sovellus luo kaksi oliota: ”OpiskelijaA” ja ”OpettajaA” ja sijoittaa ne vuorotellen monimuotoiseen olioon. Lisäksi sovellus kutsuu ”Tulosta”-operaatiota

monimuotoiselle ”Cpoly”-oliolle, joka suorittaa myöhäisen sidonnan ja kutsuu oikeata operaatiota monimuotoisen olion edustamalle oliolle.

```
PROGRAM main
    USE COpiskelija
    USE COpettaja
    USE CPoly

    TYPE (Opiskelija), TARGET :: OpiskelijaA
    TYPE (Opettaja), TARGET :: OpettajaA
    TYPE (Osoitin) :: os

    CALL Rakentaja(OpiskelijaA, 23, 'Kalle')
    CALL Rakentaja(OpettajaA, 45, 'Pekka', 2500)

    os=Poly(OpiskelijaA)
    CALL Tulosta(os)
    WRITE (*,*)
    os=Poly(OpettajaA)
    CALL Tulosta(os)
END PROGRAM


--- Ohjelman tuottama tuloste ---
 Henkilön nimi: Kalle
 Kalle on opiskelija
 Kalle ei opiskele

 Henkilön nimi: Pekka
 Pekka on opettaja
 Palkka:         2500
```

Kuva 5.31 Esimerkkisovellus, joka hyödyntää monimuotoista luokkaa.

Monimuotoinen luokka tietää sisältämiensä luokkien tyypit ja rajapinnat, mutta ei niiden toteutuksia. Tämän takia monimuotoisen luokan luominen on mekaanista ja isojen periytymishierarkioiden kohdalla hyvin työlästä. Decykin, Nortonin ja Szymanskin (1997b) mukaan monimuotoisen luokan luominen voitaisiin periaatteessa automatisoida erillisellä sovelluksella.

5.8 Monimuotoisen luokan automaattinen luominen

Monimuotoisen luokan automaattinen luominen erillisen sovelluksen avulla on toteutettu esimerkiksi Metsäntutkimuslaitoksen uuden sukupolven metsätietojärjestelmän kehityshankkeessa (Berger, 2002). Liitteessä kahdeksan on automaattisesti generoidun monimuotoisen luokan Fortran 90 -kielinen toteutus, joka toteuttaa monimuotoisuuden luokille ”CHenkilo”, ”COpiskelija” ja ”COpettaja”.

Sovellus on toteutettu C++-kielellä ja sen laajuus on noin 1500 koodiriviä. Fortran 90 -luokat sisältävät tiedostot sekä sovellus laitetaan samaan hakemistoon, jonka jälkeen suoritetaan sovellus. Sovellus etsii hakemistosta automaattisesti Fortran 90 -luokat eikä käyttäjän tarvitse erikseen määritellä mitään. Automaattisesti luotu monimuotoinen luokka on sovelluksen suorituksen jälkeen yhdessä tiedostossa. Sovelluksessa on pyritty käyttäjän kannalta mahdollisimman pieneen työmäärään.

Sovelluksessa on tehty yleinen Fortran 90 -kielellä toteutettua luokkaa kuvaava tietorakenne, johon tallennetaan Fortran 90 -luokan nimi, attribuutit ja operaatiot sekä operaatioiden palautusarvot ja parametrit. Käytännön sovelluksessa Fortran 90 -luokan toteutuksessa on noudatettava tiettyjä sääntöjä. Tärkeimpiä sääntöjä on kolme. Ensinnäkin jokainen luokan operaatio saa ensimmäisenä parametrinaan ”this”-nimisen muuttujan, joka on luokan omat attribuutit sisältävä tietue. Toiseksi pelkän operaation palautusarvo ei riitä erottelemaan ylikuormitettuja operaatioita. Kolmanneksi parametrien tyypit on esiteltävä samassa järjestyksessä kuin ne on esitelty operaation esittelyssä.

Monimuotoinen luokka luodaan keräämällä kohdeluokat ja kunkin kohdeluokan operaatiot. Aluksi monimuotoiseen luokkaan liitetään osoitinattribuutit ja niitä käsittelevät operaatiot, kuten osoittimen aktivointi. Monimuotoisen luokan attribuutit ovat suojattuja, joten luokassa tarvitaan erillisiä operaatioita, jotka palauttavat attribuuttien eli osoittimien arvoja. Lisäksi luokassa on erillinen operaatio attribuuttien alustamiseksi.

Tämän jälkeen sovellus luo monimuotoisen luokan rajapinnan ”INTERFACE”-lohkojen avulla. Rajapinta koostuu monimuotoiseen luokkaan kuuluvien luokkien operaatioista, lukuun ottamatta luokkien rakentajia ja purkajia. Samannimisen operaation esiintyessä erilaisin parametrein (esimerkiksi liitteessä kahdeksan operaatio ”VANHENE”), esitellään ”INTERFACE”-lohkon sisällä kaksi tai useampia aliohjelmia, jotka erotetaan toisistaan numerosta nolla alkavalla kasvavalla numeroinnilla nimen lopussa. Samannimisen operaation esiintyessä parametreiltaan aina samanlaisena (esimerkiksi liitteessä kahdeksan operaatio ”TULOSTA”), riittää ”INTERFACE”-lohkossa yhden aliohjelman esittely operaation nimen alla, joten ylikuormitusta ei tarvita. On hyvä huomata, että kuitenkin myös parametreiltaan samanlaisen operaation aliohjelman nimen perässä on numero nolla.

Rajapinnan luomisen jälkeen sovellus luo samalla kaavalla kohdeluokkien operaatioiden valintamekanismin. Sovellus käy läpi kaikkien monimuotoiseen luokkaan kuuluvien luokkien operaatiot ja tarkastaa jokaisen operaation kohdalla millä muulla luokalla on vastaavan niminen ja parametreiltaan samanlainen operaatio. Operaatiolle, joka esiintyy vain yksittäisellä luokalla, laaditaan aliohjelma tarkistamaan onko luokan osoitin aktiivinen. Mikäli osoitin on aktiivinen, kutsutaan luokan operaatiota (esimerkiksi liitteessä kahdeksan operaatio ”LISAA-KURSSI”). Operaatiolle, joka esiintyy samanlaisena useammalla luokalla, täytyy sovelluksen tekemässä aliohjelmassa tarkistaa kaikkien niiden luokkien osoittimet, joihin operaatio voi kohdistua (esimerkiksi liitteessä kahdeksan operaatio ”VANHENE”).

Aliohjelmien nimeämisessä käytetään samaa nimeämiskäytäntöä kuin rajapinnassakin. Näin rajapinta viittaa oikeaan aliohjelmaan. Aliohjelmien ensimmäinen parametri on aina ”this”-parametri. Muut sitä mahdollisesti seuraavat parametrit nimetään alkaen kirjaimesta ”A” ja jatkaen siitä aakkosjärjestyksessä.

6. Yhteenveto

Tutkielmassani olen tarkastellut olio-ohjelmointia yleisesti ja sen toteuttamista proseduraalisilla kielillä ja hybridikielellä. Tarkasteluni pääpaino on ollut Fortran 90 -kielen soveltuvuus olio-ohjelmointiin.

Tarkasteluni mukaan luvun kolme proseduraalisten kielten käyttäminen olio-ohjelmointiin on periaatteessa mahdollista, mutta esimerkiksi Fortran 77 ei ole tarpeeksi kehittynyt olio-ohjelmoinnin toteuttamiseen. Fortran 77 -kielen puutteita on korjattu Fortran 90 -kielessä. Pascal-kielellä pystytään Jackyn ja Kaletin (1987) mallilla esittämään useita olio-ohjelmoinnin piirteitä. Parhaiten valitsemistani kielistä olio-ohjelmointiin soveltuu C-kieli, koska siinä on vähiten teknisiä rajoitteita. C-kielen monipuolisuudesta ja joustavuudesta kertoo sekin, että C++-kieli perustuu C-kieleen. C++-kielen ensimmäiset versiot toimivatkin niin, että C++-kielinen koodi tulkittiin C-kieliseksi ja käännettiin C-kääntäjällä (Stroustrup, 1994). Kaikkia luvun kolme kieliä yhdistää kuitenkin se, että olio-ohjelmoinnin toteuttaminen niillä ei yleensä noudata kyseisten kielten normaaleja ohjelmointityylejä. Loppujen lopuksi riippuu näkökulmasta kannattaako näillä kielillä kuitenkaan harrastaa kunnianhimoisempaa olio-ohjelmointia.

Fortran 90 -kieltä käyttäen on mahdollista ohjelmoida olioläheisesti. Olion, luokan, luokan suojauksen, operaation ylikuormituksen ja operaation oletusarvon käsitteiden toteutus onnistuu Fortran 90 -kielellä ongelmitta. Periytymisen toteuttaminen on jo hieman työläämpää. Monimuotoisuuden ja myöhäisen sidonnan toteuttaminen vaatii jo huomattavasti enemmän työtä, koska nämä ominaisuudet täytyy ensin erikseen rakentaa Fortran 90 -kielessä.

Verrattaessa Fortran 90 -kielen olio-ohjelmointiominaisuuksia C++-kielen vastaaviin ominaisuuksiin huomaa, että Fortran 90:ssä täytyy tehdä useita asioita monimutkaisemmin. Se onko olio-ohjelmoinnin keskeisten rakenteiden käyttäminen Fortran 90:ssä järkevää, riippuu hyvin paljon sovelluksesta. Kaikkia olio-ohjelmoinnin keskeisiä rakenteita ei kuitenkaan tarvitse käyttää, vaan esimerkiksi tietojen organisointi luokkiin auttaa jo hallitsemaan isoja sovelluksia.

Tutkielman tärkein tavoite oli osoittaa, että olio-ohjelmointi ei välttämättä edellytä tiettyä olio-ohjelmointikieltä, vaan kyseessä on myös ohjelmointitekniikka, jota voidaan hyödyntää enemmän tai vähemmän muissakin ohjelmointikielissä.

Lähdeluettelo

Ben-Ari, M. (1996) *Understanding programming languages*. John Wiley & Sons Ltd, Great Britain.

Berger, F. (2002) *Simulointiin ja optimointiin perustuvan integroidun metsätalouden suunnittelumenetelmän ohjelmointikielen olioiden toteutus,* Käsikirjoitus.

Brookshear, J.G. (1994) *Computer science: an overview*. The Benjamin/Cummings Publishing Company, USA.

Decyk, V.K., Norton, C.D. and Szymanski, B.K. (1997a) Expressing Object-Oriented Concepts in Fortran90. *ACM Fortran Forum* **16**(1), 13–18.

Decyk, V.K., Norton, C.D. and Szymanski, B.K. (1997b) How to Express C++ Concepts in Fortran 90. *Scientific Programming* **6**(4), 363–390.

Decyk, V.K., Norton, C.D. and Szymanski, B.K. (1998) How to Support Inheritance and Run-Time Polymorphism in Fortran 90. *Computer Physics Communications* **115**, 9–17.

Haataja, J., Rahola, J. ja Ruokolainen J. (1998) *Fortran 90/95*. Yliopistopaino, Helsinki.

Jacky, J.P. and Kalet, I.J. (1987) An object-oriented programming discipline for standard Pascal. *Communications of the ACM* **30**(9), 772–776.

Klump, R. (2001) Understanding Object-Oriented Programming Concepts. *Power Engineering Society Summer Meeting* **2**, 1070–1074.

Korson, T. and McGregor J. (1990) Understanding object-oriented: A unifying paradigm. *Communications of the ACM **33**(9),* 40–60.

Korpela, J., Larmela, T. ja Salmela, J. (1985) *FORTRAN 77*. Teknillinen korkeakoulu tietojenkäsittelyopin laboratorio, OtaDATA ry, Espoo.

Koskimies, K. (2000) *Oliokirja*. Gummerus Kirjapaino Oy, Jyväskylä.

Lippman, S.B. (1989) *A C++ Primer.* Addison-Wesley Publishing Company, USA.

Meyer, B. (1988) *Object-oriented software construction.* Prentice Hall Publishing Company, Cambridge.

Stroustrup, B. (1997) *The C++ programming language.* Addison-Wesley Publishing Company, USA.

Stroustrup, B. (1994) *The Design and Evolution of C++.* Addison-Wesley Publishing Company, USA.

Wegner, P. (1987) Dimensions of Object-Based language design. *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87*), 168–182.

Liite 1. ”CHenkilo”-luokan lopullinen toteutus Fortran 90:llä.

```
MODULE CHenkilo

    ! Yksityinen operaatio
    PRIVATE :: Strcpy


    ! Luokan attribuutit
    TYPE Henkilo
        PRIVATE
        INTEGER :: ika                                ! Henkilön ikä
        CHARACTER, DIMENSION (:), POINTER :: nimi   ! Henkilön nimi
    END TYPE


    INTERFACE Rakentaja
        MODULE PROCEDURE CHenkilo_Rakentaja
    END INTERFACE

    INTERFACE Purkaja
        MODULE PROCEDURE CHenkilo_Purkaja
    END INTERFACE

    ! Kuormitettu operaatio
    INTERFACE Vanhene
        MODULE PROCEDURE CHenkilo_Vanhene
        MODULE PROCEDURE CHenkilo_Vanhene_Int
    END INTERFACE

    ! Operaatio mallintaa oletusarvon käyttöä
    INTERFACE Tulosta
        MODULE PROCEDURE CHenkilo_Tulosta
    END INTERFACE

    ! Yksityinen operaatio
    INTERFACE Strcpy
        MODULE PROCEDURE CHenkilo_Strcpy
    END INTERFACE

    INTERFACE Ika
        MODULE PROCEDURE CHenkilo_Ika
    END INTERFACE

    INTERFACE Nimi
        MODULE PROCEDURE CHenkilo_Nimi
    END INTERFACE


    CONTAINS


    ! Luokan rakentaja
    SUBROUTINE CHenkilo_Rakentaja(this, ika, nimi)
        TYPE (Henkilo), INTENT (OUT) :: this
        INTEGER, INTENT (IN) :: ika
        CHARACTER (LEN=*), INTENT (IN) :: nimi

        ! Muistin varaaminen
        ALLOCATE(this%nimi(LEN(nimi)))

        this%ika=ika
        CALL Strcpy(this%nimi, nimi)
    END SUBROUTINE


    ! Luokan purkaja
    SUBROUTINE CHenkilo_Purkaja(this)
        TYPE (Henkilo), INTENT (OUT) :: this

        ! Muistin vapauttaminen
        DEALLOCATE(this%nimi)
    END SUBROUTINE
```

```
    ! Lisää ika-attribuuttiin 1
    SUBROUTINE CHenkilo_Vanhene(this)
        TYPE (Henkilo), INTENT (OUT) :: this

        this%ika=this%ika+1
    END SUBROUTINE


    ! Lisää ika-attribuuttiin arvo
    SUBROUTINE CHenkilo_Vanhene_Int(this, arvo)
        TYPE (Henkilo), INTENT (OUT) :: this
        INTEGER, INTENT (IN) :: arvo

        this%ika=this%ika+arvo
    END SUBROUTINE


    ! Tulostaa attribuutit
    SUBROUTINE CHenkilo_Tulosta(this, tulosta_ika)
        TYPE (Henkilo), INTENT (IN) :: this
        LOGICAL, OPTIONAL, INTENT (IN) :: tulosta_ika

        WRITE (*,*) 'Henkilön nimi: ', this%nimi

        ! Tarkista onko parametri annettu
        IF(PRESENT(tulosta_ika)) THEN
            ! Tarkista parametrin arvo
            IF(tulosta_ika==.TRUE.) THEN
                WRITE (*,*) 'Ikä on: ', this%ika
            END IF
        END IF
    END SUBROUTINE


    ! Merkkijonon kopiointi merkkitaulukkoon
    SUBROUTINE CHenkilo_Strcpy(mtaulu, mjono)
        CHARACTER, DIMENSION (:), POINTER :: mtaulu
        CHARACTER (LEN=*), INTENT (IN) :: mjono
        INTEGER :: i

        DO i=1, MAX(LEN(mjono), SIZE(mtaulu))
            mtaulu(i)=mjono(i:i)
        END DO
    END SUBROUTINE


    ! Palauttaa ika-attribuutin
    INTEGER FUNCTION CHenkilo_Ika(this)
        TYPE (Henkilo), INTENT (IN) :: this

        CHenkilo_Ika=this%ika
    END FUNCTION


    ! Palauttaa nimi-attribuutin
    FUNCTION CHenkilo_Nimi(this) RESULT(mjono)
        TYPE (Henkilo), INTENT (IN) :: this
        CHARACTER, DIMENSION (:), POINTER :: mjono

        mjono=>this%nimi
    END FUNCTION


END MODULE
```

Liite 2. ”CHenkilo”-luokan lopullinen toteutus C++-kielellä.

```
#include <iostream>

using namespace std;

// Luokka CHenkilo
class CHenkilo
{
public:
    // Rakentaja ja Purkaja
    CHenkilo(const int nIka, const char *strNimi);
    ~CHenkilo();

    // Vanhene-operaatio on kuormitettu
    void Vanhene(void) { m_nIka++; }
    void Vanhene(int nArvo) { m_nIka+=nArvo; }

    // bIka-parametrilla on oletusarvo false
    virtual void Tulosta(bool bIka=false);

    int Ika(void) { return m_nIka; }        // Palauttaa henkilön iän
    char * Nimi(void) { return m_strNimi; } // Palauttaa henkilön nimen

protected:
    int m_nIka;         // Henkilön ikä
    char *m_strNimi;    // Henkilön nimi
};


// Luokan rakentaja
CHenkilo::CHenkilo(const int nIka, const char *strNimi)
{
    // Asettaa attribuutit ikä ja nimi paramerrien mukaan
    m_nIka=nIka;
    m_strNimi=new char[strlen(strNimi)+1];
    strcpy(m_strNimi, strNimi);
}


// Luokan purkaja
CHenkilo::~CHenkilo(void)
{
    delete m_strNimi;
}


// Tulostaa attribuutit
void CHenkilo::Tulosta(bool bIka)
{
    cout << "Henkilön nimi: " << m_strNimi << "\n";

    if(bIka)
        cout << "Ikä on: " << m_nIka << "\n";
}
```

Liite 3. ”COpiskelija”-luokan lopullinen toteutus Fortran 90:lla.

```
MODULE COpiskelija

    USE CHenkilo


    ! Luokan attribuutit
    TYPE Opiskelija
        TYPE (Henkilo) :: henkilo                    ! Perityt ominaisuudet
        INTEGER :: kursseja                          ! Monellako kursilla
        CHARACTER (LEN=16), DIMENSION(10) :: kurssit ! Kurssien nimet
    END TYPE


    INTERFACE Rakentaja
        MODULE PROCEDURE COpiskelija_Rakentaja
    END INTERFACE

    INTERFACE Purkaja
        MODULE PROCEDURE COpiskelija_Purkaja
    END INTERFACE

    ! Kuormitettu operaatio
    INTERFACE Vanhene
        MODULE PROCEDURE COpiskelija_Vanhene
        MODULE PROCEDURE COpiskelija_Vanhene_Int
    END INTERFACE

    ! Operaatio mallintaa oletusarvon käyttöä
    INTERFACE Tulosta
        MODULE PROCEDURE COpiskelija_Tulosta
    END INTERFACE

    INTERFACE Ika
        MODULE PROCEDURE COpiskelija_Ika
    END INTERFACE

    INTERFACE Nimi
        MODULE PROCEDURE COpiskelija_Nimi
    END INTERFACE

    INTERFACE LisaaKurssi
        MODULE PROCEDURE COpiskelija_LisaaKurssi
    END INTERFACE


    CONTAINS


    ! Luokan rakentaja
    SUBROUTINE COpiskelija_Rakentaja(this, ika, nimi)
        TYPE (Opiskelija), INTENT (OUT) :: this
        INTEGER, INTENT (IN) :: ika
        CHARACTER (LEN=*), INTENT (IN) :: nimi

        ! Yliluokan attribuuttien alustus
        CALL Rakentaja(this%henkilo, ika, nimi)

        this%kursseja=0
    END SUBROUTINE


    ! Luokan purkaja
    SUBROUTINE COpiskelija_Purkaja(this)
        TYPE (Opiskelija), INTENT (OUT) :: this

        ! Yliluokan tuhoaminen
        CALL Purkaja(this%henkilo)
    END SUBROUTINE
```

```
    ! Lisää ika-attribuuttiin 1
    SUBROUTINE COpiskelija_Vanhene(this)
        TYPE (Opiskelija), INTENT (OUT) :: this

        CALL Vanhene(this%henkilo)
    END SUBROUTINE


    ! Lisää ika-attribuuttiin arvo
    SUBROUTINE COpiskelija_Vanhene_Int(this, arvo)
        TYPE (Opiskelija), INTENT (OUT) :: this
        INTEGER, INTENT (IN) :: arvo

        CALL Vanhene(this%henkilo, arvo)
    END SUBROUTINE


    ! Tulostaa attribuutit
    SUBROUTINE COpiskelija_Tulosta(this, tulosta_ika)
        TYPE (Opiskelija), INTENT (IN) :: this
        LOGICAL, OPTIONAL, INTENT (IN) :: tulosta_ika

        CALL Tulosta(this%henkilo, tulosta_ika)

        WRITE (*,*) Nimi(this%henkilo), ' on opiskelija'

        IF(this%kursseja<=0) THEN
            WRITE (*,*) Nimi(this%henkilo), ' ei opiskele'
        ELSE
            WRITE (*,*) Nimi(this%henkilo), ' on ', this%kursseja, ' kurssilla'
            WRITE (*,*) 'Kurssit:'

            DO i=1, this%kursseja
                WRITE (*,*) this%kurssit(i)
            END DO
        END IF
    END SUBROUTINE


    ! Palauttaa ika-attribuutin
    FUNCTION COpiskelija_Ika(this)
        TYPE (Opiskelija), INTENT (IN) :: this

        COpiskelija_Ika=Ika(this%henkilo)
    END FUNCTION


    ! Palauttaa nimi-attribuutin merkkijonona
    FUNCTION COpiskelija_Nimi(this) RESULT(mjono)
        TYPE (Opiskelija), INTENT (IN) :: this
        CHARACTER, DIMENSION (:), POINTER :: mjono

        mjono=>Nimi(this%Henkilo)
    END FUNCTION


    ! Kurssin lisääminen opiskelijalle
    SUBROUTINE COpiskelija_LisaaKurssi(this, kurssi)
        TYPE (Opiskelija), INTENT (INOUT) :: this
        CHARACTER (LEN=*), INTENT (IN) :: kurssi

        this%kursseja=this%kursseja+1
        this%kurssit(this%kursseja)=kurssi
    END SUBROUTINE


END MODULE
```

Liite 4. ”COpiskelija”-luokan lopullinen toteutus C++-kielellä.

```
class COpiskelija : public CHenkilo
{
public:
    COpiskelija(const int nIka, const char *strNimi);
    ~COpiskelija();

    void Tulosta(bool bIka=true);
    void LisaaKurssi(const char *strKurssi);

private:
    int m_nKursseja;
    char *kurssit[10];
};


// Luokan rakentaja
COpiskelija::COpiskelija(const int nIka, const char *strNimi) : CHenkilo(nIka, strNimi)
{
    m_nKursseja=0;
}


// Luokan purkaja
COpiskelija::~COpiskelija()
{
    for(int i=0;i<m_nKursseja;i++)
        delete kurssit[i];
}


// Tulostaa attribuutit
void COpiskelija::Tulosta(bool bIka)
{
    CHenkilo::Tulosta(bIka);

    cout << m_strNimi << " on opiskelija\n";

    if(m_nKursseja<=0)
        cout << m_strNimi << " ei opiskele\n";
    else
    {
        cout << m_strNimi << " on " << m_nKursseja << " kurssilla\n";
        cout << "Kurssit:\n";

        for(int i=0;i<m_nKursseja;i++)
            cout << kurssit[i] << "\n";

        cout << "\n";
    }
}


// Kurssin lisääminen opiskelijalle
void COpiskelija::LisaaKurssi(const char *strKurssi)
{
    kurssit[m_nKursseja]=new char[strlen(strKurssi)+1];
    strcpy(kurssit[m_nKursseja], strKurssi);
    m_nKursseja++;
}
```

Liite 5. ”COpettaja”-luokan lopullinen toteutus Fortran 90:lla.

```
MODULE COpettaja

    USE CHenkilo


    ! Luokan attribuutit
    TYPE Opettaja
        TYPE (Henkilo) :: henkilo   ! Perityt ominaisuudet
        INTEGER :: palkka           ! Opettajan palkka
    END TYPE


    INTERFACE Rakentaja
        MODULE PROCEDURE COpettaja_Rakentaja
    END INTERFACE

    INTERFACE Purkaja
        MODULE PROCEDURE COpettaja_Purkaja
    END INTERFACE

    ! Kuormitettu operaatio
    INTERFACE Vanhene
        MODULE PROCEDURE COpettaja_Vanhene
        MODULE PROCEDURE COpettaja_Vanhene_Int
    END INTERFACE

    ! Operaatio mallintaa oletusarvon käyttöä
    INTERFACE Tulosta
        MODULE PROCEDURE COpettaja_Tulosta
    END INTERFACE

    INTERFACE Ika
        MODULE PROCEDURE COpettaja_Ika
    END INTERFACE

    INTERFACE Nimi
        MODULE PROCEDURE COpettaja_Nimi
    END INTERFACE

    INTERFACE PaivitaPalkka
        MODULE PROCEDURE COpettaja_PaivitaPalkka
    END INTERFACE


    CONTAINS


    ! Luokan rakentaja
    SUBROUTINE COpettaja_Rakentaja(this, ika, nimi, palkka)
        TYPE (Opettaja), INTENT (OUT) :: this
        INTEGER, INTENT (IN) :: ika, palkka
        CHARACTER (LEN=*), INTENT (IN) :: nimi

        ! Yliluokan attribuuttien alustus
        CALL Rakentaja(this%henkilo, ika, nimi)

        this%palkka=palkka
    END SUBROUTINE


    ! Luokan purkaja
    SUBROUTINE COpettaja_Purkaja(this)
        TYPE (Opettaja), INTENT (OUT) :: this

        ! Yliluokan tuhoaminen
        CALL Purkaja(this%henkilo)
    END SUBROUTINE
```

```
    ! Lisää ika-attribuuttiin 1
    SUBROUTINE COpettaja_Vanhene(this)
        TYPE (Opettaja), INTENT (OUT) :: this

        CALL Vanhene(this%henkilo)
    END SUBROUTINE


    ! Lisää ika-attribuuttiin arvo
    SUBROUTINE COpettaja_Vanhene_Int(this, arvo)
        TYPE (Opettaja), INTENT (OUT) :: this
        INTEGER, INTENT (IN) :: arvo

        CALL Vanhene(this%henkilo, arvo)
    END SUBROUTINE


    ! Tulostaa attribuutit
    SUBROUTINE COpettaja_Tulosta(this, tulosta_ika)
        TYPE (Opettaja), INTENT (IN) :: this
        LOGICAL, OPTIONAL, INTENT (IN) :: tulosta_ika

        CALL Tulosta(this%henkilo, tulosta_ika)

        WRITE (*,*) Nimi(this%henkilo), ' on opettaja'
        WRITE (*,*) 'Palkka: ', this%palkka
    END SUBROUTINE


    ! Palauttaa ika-attribuutin
    FUNCTION COpettaja_Ika(this)
        TYPE (Opettaja), INTENT (IN) :: this

        COpettaja_Ika=Ika(this%henkilo)
    END FUNCTION


    ! Palauttaa nimi-attribuutin merkkijonona
    FUNCTION COpettaja_Nimi(this) RESULT(mjono)
        TYPE (Opettaja), INTENT (IN) :: this
        CHARACTER, DIMENSION (:), POINTER :: mjono

        mjono=>Nimi(this%Henkilo)
    END FUNCTION


    ! Päivittää opettajan palkan
    SUBROUTINE COpettaja_PaivitaPalkka(this, palkka)
        TYPE (Opettaja), INTENT (OUT) :: this
        INTEGER, INTENT (IN) :: palkka

        this%palkka=palkka
    END SUBROUTINE


END MODULE
```

Liite 6. ”COpettaja”-luokan lopullinen toteutus C++-kielellä.

```
class COpettaja : public CHenkilo
{
public:
    COpettaja(const int nIka, const char *strNimi);

    void Tulosta(bool bIka=true);
    void PaivitaPalkka(int palkka);

private:
    int m_nPalkka;
};


// Luokan rakentaja
COpettaja::COpettaja(const int nIka, const char *strNimi) : CHenkilo(nIka, strNimi)
{
    m_nPalkka=0;
}


// Luokan purkaja
void COpettaja::Tulosta(bool bIka)
{
    CHenkilo::Tulosta(bIka);

    cout << m_strNimi << " on opettaja\n";
    cout << "Palkka: " << m_nPalkka << "\n";
    cout << "\n";
}


// Päivittää opettajan palkan
void COpettaja::PaivitaPalkka(int palkka)
{
    m_nPalkka=palkka;
}
```

Liite 7. ”CPoly”-luokan lopullinen toteutus Fortran 90:lla.

```
MODULE CPoly

    USE COpiskelija
    USE COpettaja


    TYPE Osoitin
        PRIVATE
        TYPE (Henkilo), POINTER :: henkilo       ! Osoitin henkilö-olioon
        TYPE (Opiskelija), POINTER :: opiskelija ! Osoitin opiskelijaolioon
        TYPE (Opettaja), POINTER :: opettaja     ! Osoitin opettajaolioon
    END TYPE


    INTERFACE Poly
        MODULE PROCEDURE Sijoita_Henkilo, Sijoita_Opiskelija, Sijoita_Opettaja
    END INTERFACE

    INTERFACE Vanhene
        MODULE PROCEDURE CPoly_Vanhene
        MODULE PROCEDURE CPoly_Vanhene_Int
    END INTERFACE

    INTERFACE Tulosta
        MODULE PROCEDURE CPoly_Tulosta
    END INTERFACE

    INTERFACE Ika
        MODULE PROCEDURE CPoly_Ika
    END INTERFACE

    INTERFACE Nimi
        MODULE PROCEDURE CPoly_Nimi
    END INTERFACE

    INTERFACE LisaaKurssi
        MODULE PROCEDURE CPoly_LisaaKurssi
    END INTERFACE

    INTERFACE PaivitaPalkka
        MODULE PROCEDURE CPoly_PaivitaPalkka
    END INTERFACE


    CONTAINS


    FUNCTION Sijoita_Henkilo(os_henkilo) RESULT(os)
        TYPE (Osoitin) :: os
        TYPE (Henkilo), TARGET, INTENT(IN) :: os_henkilo

        os%henkilo=>os_henkilo
        NULLIFY(os%henkilo)
        NULLIFY(os%opettaja)
    END FUNCTION


    FUNCTION Sijoita_Opiskelija(os_opiskelija) RESULT(os)
        TYPE (Osoitin) :: os
        TYPE (Opiskelija), TARGET, INTENT(IN) :: os_opiskelija

        NULLIFY(os%henkilo)
        os%opiskelija=>os_opiskelija
        NULLIFY(os%opettaja)
    END FUNCTION
```

```
FUNCTION Sijoita_Opettaja(os_opettaja) RESULT(os)
    TYPE (Osoitin) :: os
    TYPE (Opettaja), TARGET, INTENT(IN) :: os_opettaja

    NULLIFY(os%henkilo)
    NULLIFY(os%opiskelija)
    os%opettaja=>os_opettaja
END FUNCTION


SUBROUTINE CPoly_Vanhene(this)
    TYPE (Osoitin) :: this

    IF(ASSOCIATED(this%henkilo)) THEN
        CALL Vanhene(this%henkilo)
    ELSE IF(ASSOCIATED(this%opiskelija)) THEN
        CALL Vanhene(this%opiskelija)
    ELSE IF(ASSOCIATED(this%opettaja)) THEN
        CALL Vanhene(this%opettaja)
    END IF
END SUBROUTINE


SUBROUTINE CPoly_Vanhene_Int(this, arvo)
    TYPE (Osoitin) :: this
    INTEGER, OPTIONAL, INTENT (IN) :: arvo

    IF(ASSOCIATED(this%henkilo)) THEN
        CALL Vanhene(this%henkilo, arvo)
    ELSE IF(ASSOCIATED(this%opiskelija)) THEN
        CALL Vanhene(this%opiskelija, arvo)
    ELSE IF(ASSOCIATED(this%opettaja)) THEN
        CALL Vanhene(this%opettaja, arvo)
    END IF
END SUBROUTINE


SUBROUTINE CPoly_Tulosta(this, tulosta_ika)
    TYPE (Osoitin) :: this
    LOGICAL, OPTIONAL, INTENT (IN) :: tulosta_ika

    IF(ASSOCIATED(this%henkilo)) THEN
        CALL Tulosta(this%henkilo, tulosta_ika)
    ELSE IF(ASSOCIATED(this%opiskelija)) THEN
        CALL Tulosta(this%opiskelija, tulosta_ika)
    ELSE IF(ASSOCIATED(this%opettaja)) THEN
        CALL Tulosta(this%opettaja, tulosta_ika)
    END IF
END SUBROUTINE


FUNCTION CPoly_Ika(this)
    TYPE (Osoitin) :: this

    IF(ASSOCIATED(this%henkilo)) THEN
        CPoly_Ika=Ika(this%henkilo)
    ELSE IF(ASSOCIATED(this%opiskelija)) THEN
        CPoly_Ika=Ika(this%opiskelija)
    ELSE IF(ASSOCIATED(this%opettaja)) THEN
        CPoly_Ika=Ika(this%opettaja)
    END IF
END FUNCTION


FUNCTION CPoly_Nimi(this) RESULT(mjono)
    TYPE (Osoitin) :: this
    CHARACTER, DIMENSION (:), POINTER :: mjono

    IF(ASSOCIATED(this%henkilo)) THEN
        mjono=Nimi(this%henkilo)
    ELSE IF(ASSOCIATED(this%opiskelija)) THEN
        mjono=Nimi(this%opiskelija)
    ELSE IF(ASSOCIATED(this%opettaja)) THEN
        mjono=Nimi(this%opettaja)
    END IF
END FUNCTION
```

```
    SUBROUTINE CPoly_LisaaKurssi(this, kurssi)
        TYPE (Osoitin) :: this
        CHARACTER (LEN=*), INTENT (IN) :: kurssi

        IF(ASSOCIATED(this%opiskelija)) THEN
            CALL LisaaKurssi(this%opiskelija, kurssi)
        END IF
    END SUBROUTINE


    SUBROUTINE CPoly_PaivitaPalkka(this, palkka)
        TYPE (Osoitin) :: this
        INTEGER, INTENT (IN) :: palkka

        IF(ASSOCIATED(this%opettaja)) THEN
            CALL PaivitaPalkka(this%opettaja, palkka)
        END IF
    END SUBROUTINE


END MODULE
```

Liite 8. ”CPoly”-luokka automaattisesti luotuna Fortran 90:lle.

```
! Generated CPoly class.
MODULE CPoly
  USE CHENKILO
  USE COPETTAJA
  USE COPISKELIJA

  IMPLICIT NONE

  PUBLIC :: Poly
  PUBLIC :: ObjectP

  TYPE ObjectP
    PRIVATE
    TYPE (HENKILO), POINTER :: pHENKILO
    TYPE (OPETTAJA), POINTER :: pOPETTAJA
    TYPE (OPISKELIJA), POINTER :: pOPISKELIJA
  END TYPE ObjectP

  INTERFACE Poly
    MODULE PROCEDURE ConvertObjectHENKILO
    MODULE PROCEDURE ConvertObjectOPETTAJA
    MODULE PROCEDURE ConvertObjectOPISKELIJA
  END INTERFACE

  INTERFACE VANHENE
    MODULE PROCEDURE VANHENE_0
    MODULE PROCEDURE VANHENE_1
  END INTERFACE

  INTERFACE TULOSTA
    MODULE PROCEDURE TULOSTA_0
  END INTERFACE

  INTERFACE IKA
    MODULE PROCEDURE IKA_0
  END INTERFACE

  INTERFACE NIMI
    MODULE PROCEDURE NIMI_0
  END INTERFACE

  INTERFACE PAIVITAPALKKA
    MODULE PROCEDURE PAIVITAPALKKA_0
  END INTERFACE

  INTERFACE LISAAKURSSI
    MODULE PROCEDURE LISAAKURSSI_0
  END INTERFACE

  CONTAINS

  FUNCTION ConvertObjectHENKILO(obj) RESULT(ptr)
    TYPE (HENKILO), TARGET, INTENT(IN) :: obj
    TYPE (ObjectP) :: ptr

    ptr%pHENKILO=>obj
    NULLIFY(ptr%pOPETTAJA)
    NULLIFY(ptr%pOPISKELIJA)
  END FUNCTION

  FUNCTION ConvertObjectOPETTAJA(obj) RESULT(ptr)
    TYPE (OPETTAJA), TARGET, INTENT(IN) :: obj
    TYPE (ObjectP) :: ptr

    NULLIFY(ptr%pHENKILO)
    ptr%pOPETTAJA=>obj
    NULLIFY(ptr%pOPISKELIJA)
  END FUNCTION

  FUNCTION ConvertObjectOPISKELIJA(obj) RESULT(ptr)
    TYPE (OPISKELIJA), TARGET, INTENT(IN) :: obj
    TYPE (ObjectP) :: ptr

    NULLIFY(ptr%pHENKILO)
```

```
  NULLIFY(ptr%pOPETTAJA)
  ptr%pOPISKELIJA=>obj
END FUNCTION

SUBROUTINE NullifyAll(obj)
  TYPE (ObjectP) :: obj

  NULLIFY(obj%pHENKILO)
  NULLIFY(obj%pOPETTAJA)
  NULLIFY(obj%pOPISKELIJA)
END SUBROUTINE

FUNCTION GetHENKILOPtr(obj) RESULT(ptr)
  TYPE (ObjectP) :: obj
  TYPE (HENKILO), POINTER :: ptr

  IF(ASSOCIATED(obj%pHENKILO)) THEN
    ptr=>obj%pHENKILO
  ELSE
    NULLIFY(ptr)
  END IF
END FUNCTION

FUNCTION GetOPETTAJAPtr(obj) RESULT(ptr)
  TYPE (ObjectP) :: obj
  TYPE (OPETTAJA), POINTER :: ptr

  IF(ASSOCIATED(obj%pOPETTAJA)) THEN
    ptr=>obj%pOPETTAJA
  ELSE
    NULLIFY(ptr)
  END IF
END FUNCTION

FUNCTION GetOPISKELIJAPtr(obj) RESULT(ptr)
  TYPE (ObjectP) :: obj
  TYPE (OPISKELIJA), POINTER :: ptr

  IF(ASSOCIATED(obj%pOPISKELIJA)) THEN
    ptr=>obj%pOPISKELIJA
  ELSE
    NULLIFY(ptr)
  END IF
END FUNCTION

SUBROUTINE VANHENE_0(this)
  TYPE (ObjectP) :: this

  IF(ASSOCIATED(this%pHENKILO)) THEN
    CALL VANHENE(this%pHENKILO)
  ELSE IF(ASSOCIATED(this%pOPETTAJA)) THEN
    CALL VANHENE(this%pOPETTAJA)
  ELSE IF(ASSOCIATED(this%pOPISKELIJA)) THEN
    CALL VANHENE(this%pOPISKELIJA)
  END IF
END SUBROUTINE

SUBROUTINE VANHENE_1(this, A)
  TYPE (ObjectP) :: this
  INTEGER, INTENT (IN) :: A

  IF(ASSOCIATED(this%pHENKILO)) THEN
    CALL VANHENE(this%pHENKILO, A)
  ELSE IF(ASSOCIATED(this%pOPETTAJA)) THEN
    CALL VANHENE(this%pOPETTAJA, A)
  ELSE IF(ASSOCIATED(this%pOPISKELIJA)) THEN
    CALL VANHENE(this%pOPISKELIJA, A)
  END IF
END SUBROUTINE

SUBROUTINE TULOSTA_0(this, A)
  TYPE (ObjectP) :: this
  LOGICAL, OPTIONAL, INTENT (IN) :: A

  IF(ASSOCIATED(this%pHENKILO)) THEN
    CALL TULOSTA(this%pHENKILO, A)
  ELSE IF(ASSOCIATED(this%pOPETTAJA)) THEN
```

```
      CALL TULOSTA(this%pOPETTAJA, A)
    ELSE IF(ASSOCIATED(this%pOPISKELIJA)) THEN
      CALL TULOSTA(this%pOPISKELIJA, A)
    END IF
  END SUBROUTINE

  FUNCTION IKA_0(this) RESULT(ret)
    INTEGER :: ret
    TYPE (ObjectP) :: this

    IF(ASSOCIATED(this%pHENKILO)) THEN
      ret=IKA(this%pHENKILO)
    ELSE IF(ASSOCIATED(this%pOPETTAJA)) THEN
      ret=IKA(this%pOPETTAJA)
    ELSE IF(ASSOCIATED(this%pOPISKELIJA)) THEN
      ret=IKA(this%pOPISKELIJA)
    END IF
  END FUNCTION

  FUNCTION NIMI_0(this) RESULT(ret)
    CHARACTER, DIMENSION (:), POINTER :: ret
    TYPE (ObjectP) :: this

    IF(ASSOCIATED(this%pHENKILO)) THEN
      ret=NIMI(this%pHENKILO)
    ELSE IF(ASSOCIATED(this%pOPETTAJA)) THEN
      ret=NIMI(this%pOPETTAJA)
    ELSE IF(ASSOCIATED(this%pOPISKELIJA)) THEN
      ret=NIMI(this%pOPISKELIJA)
    END IF
  END FUNCTION

  SUBROUTINE PAIVITAPALKKA_0(this, A)
    TYPE (ObjectP) :: this
    INTEGER, INTENT (IN) :: A

    IF(ASSOCIATED(this%pOPETTAJA)) THEN
      CALL PAIVITAPALKKA(this%pOPETTAJA, A)
    END IF
  END SUBROUTINE

  SUBROUTINE LISAAKURSSI_0(this, A)
    TYPE (ObjectP) :: this
    CHARACTER (LEN=*), INTENT (IN) :: A

    IF(ASSOCIATED(this%pOPISKELIJA)) THEN
      CALL LISAAKURSSI(this%pOPISKELIJA, A)
    END IF
  END SUBROUTINE

END MODULE CPoly
```