Osa I

Pohjan luominen

1

Olioperusteisen ohjelmoinnin teoria

Tämän luvun päätavoite on käydä läpi olioperusteisen ohjelmoinnin (OOP:n) käsitteet ja antaa sinulle käsitys siitä, miten merkityksellinen oliperusteinen menetelmä ohjelmoinnissa on. Monen ohjelmointikielen, kuten C++:n ja Microsoft Visual Basicin, sanotaan "tukevan objekteja", mutta harvat ohjelmointikielet tosiasiassa tukevat kaikkia olioperusteiseen ohjelmointiin liittyviä käsitteitä. C# on yksi näistä kielistä: se on suunniteltu perusteista lähtien täysin olioperusteiseksi, komponenttipohjaiseksi kieleksi. Eli, jotta saisit tästä kirjasta mahdollisimman paljon irti, tarvitset tässä luvussa esitettävän tukevan pohjan olioperusteisesta ohjelmointitekniikasta.

Tiedän, että tämän tapaisen teoreettisen luvun jättävät usein väliin ne lukijat, jotka haluavat sukeltaa suoraan koodiin, mutta jos et pidä itseäsi "olioguruna", suosittelen tämän luvun lukemista. Ne teistä, jotka tuntevat jonkin verran olioperusteista ohjelmointia, saavat tästä luvusta hyvän kertauksen. Pidä myös mielessä, että seuraavat luvut tulevat käyttämään tässä esitettyjä termejä ja käsitteitä.

Kuten sanoin, monet ohjelmointikielet väittävät olevansa oliokieliä tai olioperusteisia kieliä, mutta harvat todella ovat. C++ ei ole sen vastaansanomattoman tosiasian takia, että sen juuret ovat syvällä upoksissa C-kieleen. Liian monet OOP:n periaatteista on uhrattu, jotta vanha C-koodin tuki säilyisi. Jopa Javalla, niin hyvä kuin se onkin, on joitakin rajoituksia olioperusteisena kielenä. Viittaan erityisesti siihen tosiasiaan, että Javassa sinulla on *perustyyppejä* (primitive types) ja *objektityyppejä* (object types), joita käsitellään ja jotka

käyttäytyvät hyvin erilaisesti. Tämän luvun päämäärä ei ole kuitenkaan eri ohjelmoinkielten vertailu olio-ominaisuuksien määrän ja puhtauden perusteella. Sen sijaan tässä luvussa esitetään objektiiviset ja kielistä riippumattomat OOP:n pääperiaatteet.

Ennen kuin jatkamme, haluan sanoa, että olioperusteinen ohjelmointi on paljon enemmän kuin markkinointitermi (vaikka siitä on joillekin tullut sellainen), uusi syntaksi tai uusi sovelluksen ohjelmointirajapinta (API). Olioperusteinen ohjelmointi on täydellinen joukko käsitteitä ja ajatuksia. Se on erilainen tapa ajatella tietokoneohjelman avulla ratkaistavaa ongelmaa ja ratkaista se luonnollisemmalla ja siksi myös tuotteliaammalla tavalla.

Ensimmäisessä työpaikassani käytin Pascal-kieltä tehdessäni lipunmyynti- ja matkasuunnitteluohjelman Holiday On Ice -kiertueelle. Kun siirryin muihin tehtäviin ja muihin sovelluksiin, tein ohjelmia PL/1:llä ja RPG III:lla (ja RPG/400:lla). Muutaman vuoden kuluttua aloin tekemään sovelluksia C-kielellä. Kussakin tilanteessa oli helppoa soveltaa edellisissä tehtävissä saatua kokemusta. Kunkin uuden ohjelmointikielen oppimiskäyrä oli lyhyempi riippumatta kielen monimutkaisuudesta. Tämä johtui siitä, että siihen asti, kun aloitin ohjelmoinnin C++:lla, olivat kaikki kielet olleet proseduraalisia kieliä ja siten eronneet toisistaan pääasiassa vain rakenteensa puolesta.

Jos kuitenkaan et ole aiemmin ohjelmoinut oliperusteisesti, ota huomioon: *aiempi proseduraalisen ohjelmointikielen tuntemus ei auta sinua nyt!* Olioperusteinen ohjelmointi on erilainen tapa ajatella miten ongelman ratkaisu suunnitellaan ja toteutetaan. Itse asiassa tutkimukset ovat osoittaneet, että sellaiset henkilöt, jotka eivät ole aiemmin ohjelmoineet, oppivat olioperusteisen ohjelmoinnin paljon nopeammin kuin jotain BASICin, COBOLin tai C:n tapaista proseduraalista kieltä osaavat. Tällaisten henkilöiden ei tarvitse "unohtaa" sellaisia proseduraalisen ohjelmoinnin huonoja tapoja, jotka hidastavat OOP-ymmärrystä. He aloittavat puhtaalta pöydältä. Jos olet ohjelmoinut vuosia proseduraalisella kielellä ja C# on ensimmäinen olioperusteinen kielesi, paras neuvoni sinulle on, että pidät mielesi avoimena ja sulattelet kirjassa antamani ajaukset ennen kuin nostat kätesi ylös ja sanot "Voin tekaista tämän hetkessä [lisää tähän oman proseduraalisen suosikkikielesi nimi]." Jokainen, joka on tullut proseduraalisesta maailmasta oliperusteiseen, on käynyt läpi tämän oppimiskäyrän ja se on ollut vaivan arvoista. Olioperusteisen ohjelmoinnin edut ovat huomattavat sekä tehokkaamman koodin kirjoittamisen kannalta että helposti muokattavan ja laajennettavan valmiin sovelluksen kannalta. Saattaa tosin olla, että edut eivät näy hetkessä. Kuitenkin lähes 20 vuoden ohjelmointiura (johon sisältyy viimeiset 8 vuotta oliperusteista ohjelmointia), on osoittanut minulle, että OOP-periaatteet, *kun niitä sovelletaan oikein*, täyttävät lupauksensa. Pitemmittä puheitta kääritään hihat ja ryhdytään tutkimaan, mistä tässä kaikessa oikeastaan on kysymys.

Kaikki ovat objekteja

Todellisessa olioperusteisessa kielessä kaikki ongelman piiriin kuuluvat käsitteet esitetään *objekteina*. (Käytän tässä kirjassa Coad/Yourdonin määritelmää "ongelman piirille". Määritys kuuluu suunnilleen näin: Ongelman piiri on se ongelma, jota yrität ratkaista täysin määriteltynä eli huomioiden sen monimutkaisuus, käsitteet, haasteet ja niin edelleen.) Kuten ehkä arvaatkin, objekti on keskeinen olioperusteisen ohjelmoinnin käsite. Useimmat meistä eivät kuljeksi ympäriinsä ajatellen sellaisin käsittein kuin tietue, datapaketti, funktiokutsu tai osoitin; sen sijaan me yleensä ajattelemme objektikäsittein. Katsotaanpa esimerkkiä.

Jos kirjoitat laskutussovellusta ja sinun pitää kirjata laskun yksittäiset rivit, niin kumpi seuraavista ajatusmalleista olisi ymmärrettävämpi asiakkaan kannalta katsottuna?

- **Perinteinen ajatusmalli** Minun pitää hakea laskun otsikkoa esittävä tietorakenne. Siihen kuuluu kaksisuuntainen laskurivin tietorakenteen sisältävä linkitetty lista, joihin sisältyy rivin summa. Siksi, jotta saisin laskun loppusumman selville, minun pitää määritellä vaikkapa *totalInvoiceAmount*-niminen muuttuja ja antaa sen alkuarvoksi 0, hakea osoitin laskun otsikkorakenteeseen, hakea laskurivien linkitetyn listan pää ja sen jälkeen käydä laskurivit läpi. Kunkin laskurin kohdalla haen sen jäsenmuuttujasta rivin summan ja lisään sen *totalInvoiceAmount*-muuttujaan.

- **Olioperusteinen ajatusmalli** Minulla on lasku-objekti. Lähetän viestin sille pyytäen kokonaisummaa. Minun ei tarvitse ajatella, miten tiedot objektin sisällä on tallennettu, kuten perinteisessä ajatusmallissa pitää tehdä. Käsittelen objektiä yksinkertaisen luonnollisesti, pyytäen siltä erilaisia asioita lähettämällä sille viestejä. (Sitä joukkoa viestejä, joita objekti voi käsitellä, kutsutaan kokonaisuutena objektin *rajapinnaksi*. Seuraavassa kappaleessa selitän, miksi oliperusteisessa ajatustavassa on parempi ajatella rajapintaa kuin toteutusta, kuten olen tässä tehnyt)

Oliperusteinen ajatustapa on selvästikin intuitiivisempi ja lähempänä sitä mallia, jota useimmat meistä käyttävät ongelmaa ratkaistessaan. Toisessa ratkaisussa laskuobjekti luultavasti käy läpi *kokoelman* laskuriviobjekteja, lähettäen kullekin niistä viestin ja pyytäen rivin summaa. Jos kuitenkin haluat tietää kokonaissumman, *sinua ei kiinnosta miten se saadaan selville*. Et välitä siitä, koska yksi oliperusteisen ohjelmonnin peruspilari on *kapselointi*

(encapsulation) eli objektin mahdollisuus piilottaa sisäiset tietonsa ja metodinsa ja esittää rajapinta, jonka avulla objektin tärkeitä osia voidaan ohjelmallisesti käsitellä. Se, miten objekti sisäisesti tehtävänsä tekee, ei ole tärkeää niin kauan, kuin se pystyy tehtävänsä tekemään. Sinulle on kerrottu objektin rajapinta ja käytät tuota rajapintaa saadaksesi objektin tekemään haluamasi asiat puolestasi. (Selitän myöhemmin tässä luvussa tarkemmin kapseloinnin ja rajapinnan käsitteet.) Ajatuksen ydin tässä on se, että ohjelmat, jotka on tehty olioperusteisesti ongelman piirissä olevia todellisia objekteja vastaavasti, ovat helpommin suunniteltavia ja toteutettavia, koska voimme ajatella niiden toimintaa luonnollisella tavalla.

Huomaa, että toinen ajatusmalli vaatii objektin, joka tekee työn puolestamme eli laskee rivien kokonaisumman. Objekti ei sisällä pelkästään tietoja kuten tietueet tekevät. Määritelmän mukaan objektit koostuvat tiedoista ja noita tietoja käsittelevistä metodeista. Tämä tarkoittaa, että kun työskennellään ongelman kimpussa, voimme tehdä enemmän kuin pelkästään miettiä tietorakenteen. Voimme myös miettiä, mitä metodeja kuhunkin objektiin tulee liittää niin, että siitä tulee tietojen suhteen täysin kapseloitu, sisältäen vain joukon toimintoja eli metodeja. Seuraava esimerkki ja tulevien kappaleiden esimerkit selventävät tätä ajatusmallia.

> Huomaa Tämän luvun koodiesimerkit esittelevät olioperusteisen ohjelmoinnin periaatteita. Vaikka monet koodipätkät onkin tehty C#:lla, pidä mielessä, että periaatteet sinänsä ovat yleisiä OOP:n periaatteita eivät minkään erityisen kielen ominaisuuksia. Vertailun vuoksi näytän tässä luvussa esimerkit myös toteutettuna C-kielellä, joka ei ole olioperusteinen kieli.

Oletetaan, että teet sovellusta, joka laskee uuden yhtiösi ainoa työnekijän Amyn palkan. C-kielellä tekisit jotain seuraavanlaista liittäen työntekijään määrätyn tietotyypin:

```
struct EMPLOYEE
{
    char szFirstName[25];
    char szLastName[25];

    int iAge;

    double dPayRate;

};
```

Näin lasket Amyn palkan käyttäen *EMPLOYEE*-tietuetta:

```
void main()
{
    double dTotalPay;

    struct EMPLOYEE* pEmp;
    pEmp = (struct EMPLOYEE*)malloc(sizeof(struct EMPLOYEE));

    if (pEmp)
    {
        pEmp->dPayRate = 100;

        strcpy(pEmp->szFirstName, “Amy”);
        strcpy(pEmp->szLastName, “Anderson”);
        pEmp->iAge = 28;

        dTotalPay = pEmp->dPayRate * 40;
        printf(“Total Payment for %s %s is %0.2f",
                pEmp->szFirstName, pEmp->szLastName, dTotalPay);
    }

    free(pEmp);
}
```

Tässä esimerkissä koodi perustuu tietueessa oleviin tietoihin ja ulkoiseen (tietueen kannalta) koodiin, joka käyttää tietuetta. Eli missä on ongelma? Suurin ongelma on abstraktiossa: *EMPLOYEE*-tietueen käyttäjän pitää tietää aivan liian paljon työntekijän tiedoista. Miksi? Sanotaan, että myöhemmin haluat muuttaa tapaa, jolla Amyn palkka lasketaan. Haluat esimerkiksi ottaa huomioon erilaiset vakuutusmaksut, jotta saat laskettua maksettavan nettopalkan. Et pelkästään joudu muuttamaan koodia, joka käsittelee *EMPLOYEE*-tietuetta vaan sinun pitää myös dokumentoida (yrityksesi muita koodaajia varten) se tosiasia, että on tapahtunut muutoksia sen käytössä.

Katsotaan nyt tämän esimerkin C#-versioita:

```
using System;

class Employee
{
    public Employee(string firstName, string lastName,
                    int age, double payRate)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.payRate = payRate;
    }
```

(jatkuu)

```
    protected string firstName;
    protected string lastName;
    protected int age;
    protected double payRate;

    public double CalculatePay(int hoursWorked)
    {
        // Calculate pay here.
        return (payRate * (double)hoursWorked);
    }
}

class EmployeeApp
{
    public static void Main()
    {
        Employee emp = new Employee (“Amy", “Anderson", 28, 100);
        Console.WriteLine(“\nAmy’s pay is $” + emp.CalculatePay(40));
    }
}
```

EmployeeApp-esimerkin C#-versiossa voi objektin käyttäjä yksinkertaisesti kutsua objektin *CalculatePay*-metodia, jolloin objekti laskee oman palkkansa. Tämän mallin etu on siinä, että käyttäjän ei enää tarvitse huolehtia siitä, miten palkan laskenta täsmällisesti suoritetaan. Jos joskus tulevaisuudessa päätät muokata palkan laskentatapaa, muutoksella ei ole vaikutusta olemassa olevaan koodiin. Abstraktion taso on yksi objektien käytön suurimmista eduista.

Yksi asiallinen kommentti tässä vaiheessa on se, että voit abstraktoida C-koodin luomalla funktion, joka käsittelee *EMPLOYEE*-tietuetta. Kuitenkin se tosiasia, että minun pitää luoda tämä funktio täysin erillään tietueesta, jota se käsittelee, on tarkkaan ottaen ongelma. Kun käytät C#:n tapaista olioperusteista kieltä, objektin tiedot ja metodit, jotka sitä käsittelevät (sen rajapinta) ovat aina yhdessä.

Pidä mielessä, että objektin metodit käsittelevät objektin muuttujia. Kuten näet edellä olevassa esimerkissä, kukin *Employee*-jäsenmuuttuja on määritelty käsittelymääreellä *protected* lukuunottamatta *CalculatePay*-metodia, joka on määritelty sanalla *public*. Käsittelymääreitä käytetään määrittämään taso, jolla periytetyt luokat ja asiakasohjelman koodi voivat käsitellä jäsenmuuttujia. *protected*-määre tarkoittaa, että periytetty luokka voi käsitellä sitä, mutta asiakasohjelma ei. *public*-määre tekee muuttujasta käsiteltävän sekä periytetyssä luokassa että asiakasohjemassa. Puhun näistä määreistä tarkemmin luvussa 5, “Luokat”, mutta oleellinen asia muistaa on se, että käsittelymääreiden avulla voit suojata luokkasi jäsenmuuttujia virheelliseltä käytöltä.

Objektit ja luokat

Termien luokka ja objekti välinen ero aiheuttaa yleensä väärinkäsityksia niille, jota eivät ole aiemmin tutustuneet olioperusteiseen ohjelmointiin. Selventääksemme näiden termien välistä eroa, tehdään EmloyeeApp-sovelluksesta totuudenmukaisempi olettamalla, että yhtiössämme on useampia työntekijöitä.

C-kielellä voisimme määritellä työntekijätaulukon, joka sisältäisi *EMPLOYEE*-tietueita ja jatkaa siitä. Koska emme tiedä, montako työntekijää yhtiöllämme joskus tulee olemaan, voimme luoda taulukon kiinteällä elementtien määrällä, esimerkiksi 10000:lla. Koska yhtiöllä kuitenkin tällä hetkellä on palkkalistoillaan vain Amy, sellainen taulukko olisi resurssien tuhlausta. Sen sijaan voisimme luoda *EMPLOYEE*-tietueista linkitetyn listan ja dynaamisesti varata tarpeen mukaan tilaa sovelluksessa.

Ajatukseni on se, että tässä teemme juuri sitä, mitä meidän ei pitäisi. Kulutamme ajatusenergiaa miettimällä ohjemointikieltä ja konetta (kun mietimme miten paljon muistia varataan ja koska), sen sijaan, että keskittyisimme varsinaiseen ongelmaan. Objekteja käyttämällä voimme keskittyä ongelmamme ratkaisevaan liiketoimintalogiikkaan laitteiston sijaan.

On monia tapoja määritellä luokka ja erottaa se objektista. Voit ajatella luokkaa yksinkertaisena tyyppinä (kuten *char*, *int* tai *long*), johon on liitetty metodeja. Objekti on tyypin tai luokan instanssi. Pidän kuitenkin itse eniten seuraavasta määritelmästä: luokka on objektin suunnitelma. Sinä ohjelmoijana teet tämän suunnitelman aivan kuten insinööri tekee talon suunnitelman. Kun suunnitelma on valmis, sinulla on vain yksi suunnitelma kustakin talotyypistä. Suunnitelman voi kuitenkin hankkia vaikka kuinka monta ihmistä ja rakentaa samanlaisen talon. Saman periaatteen mukaan luokka on suunnitelma määrätystä joukosta toimintoja ja tuon luokan perusteella luotu objekti sisältää nuo toiminnot.

Instantiointi

Olioperusteiselle ohjelmoinnille tyypillinen termi on *instantiointi* (instantiation, luominen). Se tarkoittaa instanssin luomista luokasta. Luotu instanssi on objekti. Seuraavassa esimerkissa teemme luokan eli määrittelyn objektille. Toisin sanoen, esimerkissä ei varata muistia, koska meillä on olemassa vasta objektin piirustus, ei vielä itse objektia.

```
class Employee
{
    public Employee(string firstName, string lastName,
                    int age, double payRate)
```

(jatkuu)

```
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.payRate = payRate;
    }

    protected string firstName;
    protected string lastName;
    protected int age;
    protected double payRate;

    public double CalculatePay(int hoursWorked)
    {
        // Palkka lasketaan tässä.
        return (payRate * (double)hoursWorked);
    }
}
```

Jotta saisimme instantioitua tämän luokan ja käytettyä sitä, meidän tulee määritellä sen instanssi seuraavanlaisella metodilla:

```
public static void Main()
{
    Employee emp = new Employee (“Amy", “Anderson", 28, 100);
}
```

Tässä esimerkissä *emp* määritellään olevan tyyppiä *Employee* ja se instantioidaan operaattorilla *new*. Muuttuja *emp* esittää *Employee*-luokan instanssia ja sitä käsitellään *Employee*-objektina. Instantioinnin jälkeen voimme käsitellä objektia sen julkisten jäsenten kautta. Voimme esimerkiksi kutsua *emp*-objektin *CalculatePay*-metodia. Emme voisi tehdä sitä, jos meillä ei olisi todellista objektia. (Tähän on yksi poikkeus ja se liittyy staattisiin jäsenmuuttujiin. Kerron staattisista jäsenmuuttujista sekä luvussa 5 että luvussa 6, “Metodit”)

Katsotaan seuraavaa C#-koodia:

```
public static void Main()
{
    Employee emp = new Employee();
    Employee emp2 = new Employee();
}
```

Tässä meillä on samasta *Employee*-luokasta kaksi instanssia (*emp* and *emp2)*. Vaikka ohjelmallisesti kummallakin objektilla on samat ominaisuudet, kumpikin instanssi sisältää omat tietonsa ja niitä voidaan käsitellä erikseen. Saman periaatteen mukaan voimme luoda näitä Employee-objekteja kokonaisen taulukon tai kokoelman. Luvussa 7, “Ominaisuudet, taulukot ja indeksoijat”, puhumme lisää taulukoista. Seikka, jonka haluan tässä esittää, on

se, että useimmat olioperusteiset kielet tukevat mahdollisuutta määritellä objektitaulukko. Siten voit helposti ryhmitellä objektit ja käydä ne läpi kutsumalla objektitaulukon metodeja tai käsittelemällä taulukkoa. Vertaa tätä työhön, joka sinun pitää tehdä linkitetyn listan kanssa, jossa sinun pitää itse liittää sen kukin jäsen edellä ja jäljessä tulevaan jäseneen.

Olioperusteisen ohjelmointikielen kolme perusominaisuutta

C++-kielen kehittäjän Bjarne Stroustrupin mukaan ohjelmointikieli voi kutsua itseään olioperusteiseksi, jos se tukee kolmea periaatetta: objekteja, luokkia ja periytymistä. Olioperusteiseksi kieleksi katsotaan kuitenkin nykyään yleisesti kielet, jotka tukevat seuraavia kolmea asiaa: *kapselointia*, *periytymistä* ja *monimuotoisuutta* (encapsulation, inheritance ja polymorphism). Syy tälläiseen ajatustavan muutoksen on se, että vuosien aikana olemme ymmärtäneet, että kapselointi ja monimuotoisuus ovat yhtä olennaisia osia olioperusteisissa järjestelmissä kuin luokka ja periytyminen.

Kapselointi

Kuten aiemmin mainitsin, *kapselointi*, jota joskus kutsutaan myös termillä informaation kätkentä, on objektin ominaisuus, jolla voit kätkeä sen sisäisen olemuksen ja tarjota rajapinnassa vain ne jäsenet, jotka haluat antaa asiakasohjelman käyttöön. Puhuin kuitenkin samassa yhteydessä myös abstraktiosta, joten tässä kappaleessa selvennän näiden kahden samantyyppisen asian eroa. Kapselointi tarjoaa muurin luokan ulkoisen rajapinnan (eli niiden julkisten jäsenten, jotka näkyvät luokan käyttäjälle) ja sen sisäisen toteutuksen yksityiskohtien välille. Kapseloinnin etu luokan ohjelmoijan kannalta on se, että hän voi tehdä näkyväksi luokan jäsenen, joka voi jäädä staattiseksi tai muuttumattomaksi samalla kun piilottaa dynaamisemman ja muuttuvan sisäisen toiminnan. Kuten näit aiemmin tässä luvussa, kapselointi toteutetaan C#:ssa liittämällä kuhunkin luokan jäseneen käsittelymääre (public, private tai protected).

Abstraktioiden suunnittelu

Abstraktio viittaa siihen, miten annettu ongelma esitetään ohjelmassa. Ohjelmointikielet sinällään tarjoavat abstraktion. Ajattele asiaa näin: Milloin jouduit viimeksi huolehtimaan prosessorin rekistereistä ja pinosta? Vaikka olisit alunperin opetellut ohjelmoimaan assemblerilla, lyön vetoa, että siitä on todella kauan, kuin jouduit ajattelemaan sellaisia alhaisen tason konekohtaisia ominaisuuksia. Syy on se, että useimmat ohjelmointikielet abstraktoivat eli piilottavat sinulta nuo yksityiskohdat, jotta voit keskittyä varsinaiseen ongelmaan.

Olioperusteiset ohjelmointikielet antavat sinun määritellä luokkia, joiden nimet ja rajapinnat muistuttavat todellisia ongelman käsitteitä siten, että objektien käyttäminen antaa ongelmiin "luonnollisemman" tuntuman. Tuloksena siitä, että poistetaan ne osat, jotka eivät suoraan liity ratkaistavaan ongelmaan, on se, että pystyt keskittymään todelliseen ongelmaan ja tuottavuus nousee. Itse asiassa, mukaillen Bruce Eckeliä väitettä, jonka hän esitti kirjassaan *Thinking in Java* (Prentice Hall Computer Books, 2000), "kyky ratkaista useimmat ongelmat on suoraan verrannollinen käytetyn abstraktion laatuun."

Se on kuitenkin vain yksi abstraktion taso. Mene askel pidemmälle, koska luokan suunnittelijana sinun pitää miettiä, miten voit parhaiten suunnitella sellaisen abstraktion luokkasi käyttäjälle, joka mahdollistaa käyttäjän keskittymisen oleelliseen eikä juuttuvan luokan toimintaan liittyviin yksityiskohtiin. Tässä vaiheessa hyvä kysymys on "Miten luokan rajapinta liittyy abstraktioon?" Luokan rajapinta on abstraktion toteutus.

Käytän ohjelmointikursseilta tuttua esimerkkiä auttaakseni kirkastamaan nämä kaksi käsitettä: juoma-automaatin sisäinen toiminta. Juoma-automaatin sisäinen toiminta on melko mutkikas. Toimiakseen kunnolla sen tulee hyväksyä kolikot ja rahakkeet, palauttaa vaihtorahat ja antaa valittu tuote. Juoma-automaatilla on kuitenkin rajattu joukko toimintoja, joita sen tulee tarjota käyttäjilleen. Tämä rajapinta on toteutettu kolikkoaukolla, valintapainikkeilla, vaihtorahojen pyyntövivulla, vaihtorahakaukalolla ja napilla, joka sylkee valitun tuotteen ulos. Kukin näistä osista esittää osaa koneen rajapinnassa. Juoma-automaatti on pysynyt suurin piirtein samanlaisena keksimisestään asti. Syy on se, että huolimatta sisäisestä muutoksesta tekniikan kehityksen myötä, ei sen perusrajapintaa ole tarvinnut juuri muuttaa. Oleellinen osa luokan rajapinnan suunnittelussa on ongelman kokonaisvaltainen ymmärtäminen. Tämän ymmärryksen myötä kykenet suunnittelemaan rajapinnan, joka antaa käyttäjälle ne tiedot ja toiminnot, jotka hän tarvitsee, mutta erottaa hänet luokan sisäisestä toiminnasta. Sinun pitää suunnitella rajapinta, ei pelkästään ratkaisemaan nykyiset ongelmat, vaan myös abstraktoimaan luokan sisäisen toiminnan niin, että yksityiset luokan jäsenet voivat käydä läpi rajattomasti muutoksia ilman, että se vaikuttaa olemassa olevaan koodiin.

Toinen yhtä tärkeä asia luokan abstraktion suunnittelussa on pitää kaiken aikaa mielessä asiakasohjelma. Kuvittele, että olet tekemässä yleistä tietokantamoottoria. Jos olet tietokantaguru, tunnet varmaan hyvin sellaiset termit kuin *cursor*, *commitment control* ja *tuples*. Kuitenkin useimmat ohjelmoijat, jotka eivät ole tehneet paljoa tietokantaohjelmointia, eivät välttämättä tunne näitä termejä. Käyttämällä termejä, jotka ovat luokkasi käyttäjille outoja, olet unohtanut koko abstraktion tarkoituksen, eli lisätä ohjelmoijan tuottavuutta esittämällä ongelma luonnollisilla termeillä.

Toinen esimerkki tilanteesta, jossa asiakasohjelma kannattaa pitää mielessä, on sen määrittäminen, mitkä luokan jäsenistä tulee olla yleisesti käytettäviä. Tässäkin auttaa itse ongelman ymmärtäminen. Tietokantamoottoriesimerkissämme et luultavasti halua asiakasohjelmien käsittelevän suoraan jäseniä, jotka edustavat sisäisiä tietopuskureita. Näiden puskurien ohjelmointi voi helposti tulevaisuudessa muuttua. Koska puskurit ovat myös oleellisen tärkeitä moottorisi kokonaistoiminnan kannalta, haluat ilman muuta varmistaa, että niitä käsitellään vain metodiesi kautta. Tällä tavalla voit varmistaa, että niitä käsitellään huolellisesti.

> Huomaa Saatat kuvitella, että olioperusteiset järjestelmät on suunniteltu pääasiassa luokkien helpon luonnin näkökulmasta. Vaikka tällainen periaate toisikin lyhyellä tähtäimellä hyviä tuloksia, kestäviä etuja saavutetaan vasta, kun ymmärretään, että tärkeintä on luokan asiakasohjelmien ohjelmoinnin helpottaminen. Pidä aina luokkia suunnitellesasi mielessäsi ohjelmoija, joka instantioi tai periyttää niitä.

Hyvän abstraktion edut

Luokkien abstraktion suunnittelu niin, että siitä on mahdollisimman paljon hyötyä niitä käyttäville ohjelmoijille, on ensiarvoisen tärkeää uudelleenkäytön kannalta. Jos pystyt tekemään vakaan ja staattisen rajapinnan, joka säilyy toteutuksen muutoksissakin, niin sovelluksesi tarvitsee aikojen kuluessa vähän muutoksia. Ajatellaanpa esimerkkinä edellä ollutta palkanlaskenta-sovelluksen koodia. *Employee*-objekstin ja palkanlaskennan toiminnallisuuden kannalta oleellisia metodeja on vain muutama, esimerkiksi *CalculatePay*, *GetAddress* ja *GetEmployeeType*. Jos tunnet palkanlaskennan hyvin, voit helposti määritellä ne metodit, joita luokan käyttäjät tulevat tarvitsemaan. Jos yhdistät luokan suunnittelussa yleisen tietämyksesi asiasta harkintaan ja hyvään suunnitteluun, voit olla suhteellisen varma, että suurin osa tämän luokan rajapinnoista tulee säilymään muuttumattomana huolimatta tulevista muutoksista luokan toteutuksessa. Loppujen lopuksi käyttäjän näkökulmasta se on vain *Employee*-luokka. Käyttäjän näkökulmasta oikeastaan mitään ei tule muuttumaan versiosta versioon.

Käyttäjän ja toteutuksen yksityiskohtien yhdistäminen on se, joka tekee koko järjestelmästä helpommin ymmärrettävän ja siksi helpomman ylläpitää. Vertaa tätä C:n tapaiseen proseduraaliseen kieleen, jossa jokaisen moduulin tulee täsmällisesti nimetä ja käsitellä annetun tietueen jäseniä. Tällöin joka kerta, kun tietueen jäsen muuttuu, jokainen koodirivi, joka viittaa tuohon tietueeseen, pitää myös muuttaa.

Periytyminen

Periytyminen viittaa ohjelmoijan mahdollisuuteen määritellä, että tietyllä luokalla on *kind-of*-yhteys toiseen luokkaan. Periytymisen avulla voit luoda (tai periyttää) uuden luokan, joka perustuu olemassa olevaan luokkaan. Voit sen jälkeen määritellä luokan haluamallasi tavalla ja luoda periytetystä luokasta uusia objekteja. Tämä kyky on välttämätön, kun luodaan luokkien hierarkkioita. Lukuunottamatta abstraktiota, periytyminen on merkittävin osa järjestelmän kokonaissuunnittelua. *Periytetty luokka* (derived class) on uusi luokka ja *kantaluokka* (base class) on se, josta uusi luokka periytyy. Uusi periytetty luokka perii kaikki kantaluokan jäsenet ja voit siten hyödyntää aiemmin tehtyä työtä.

> Huomaa C#:ssa kerrotaan mitkä kantaluokan jäsenet peritään jäsenen määrittelyssä annettavalla käsittelymääreellä. Tähän asiaan palataan luvussa 5. Tässä vaiheessa voit olettaa, että periytyvä luokka perii kaikki kantaluokkansa jäsenet.

Esimerkkinä siitä, milloin ja miten käytät periytymistä katsotaan uudelleen EmployeeApp-esimerkkiä. Meillä tulee melko varmasti olemaan eri tyyppisiä työntekijöitä, kuten kuukausipalkkaisia, urakoitsijoita ja tuntipalkkaisia. Vaikka kaikilla näillä *Employee*-objekteilla on samanlainen rajapinta, ne kuitenkin toimivat monessa suhteessa eri tavalla. Esimerkiksi *CalculatePay*-metodi toimii eri tavalla kuukausipalkkaisella kuin urakoitsijalla. Haluat kuitenkin tarjota käyttäjälle aina saman *CalculatePay*-rajapinnan riippumatta työntekijän tyypistä.

Jos et ole aiemmin tutustunut olio-ohjelmointiin, saatat miettiä "Enhän edes tarvitse objekteja tässä. Miksi en yksinkertaisesti voi ottaa *EMPLOYEE*-tietuetta ja employee-tyyppistä jäsentä ja kirjoittaa seuraavanlaisen funktion?"

```
Double CalculatePay(EMPLOYEE* pEmployee, int iHoursWorked)
{
    // Tutkitaan pEmployee-osoitinta.

    if (pEmployee->type == SALARIED)
    {
        // Lasketaan kuukausipalkka.
    }
    else if (pEmployee->type == CONTRACTOR)
    {
        // Lasketaan urakoitsijan palkka.
    }
    else if (pEmployee-> == HOURLY)
```

```
    {
        // Lasketaan tuntipalkka.
    }
    else
    {
        // Muu tapaus.
    }

    // Palautetaan jonkin valintarakenteen
    // laskema arvo.
}
```

Tässä koodissa on muutama ongelma. Ensinnäkin, *CalculatePay*-funktion onnistuminen on tiiviisti yhteydessä *EMPLOYEE*-tietueeseen. Kuten aiemmin mainitsin, tämän tyyppinen yhteys on ongelma, koska jokainen *EMPLOYEE*-tietueeseen tehty muutos rikkoo koodin. Olioperusteisena ohjelmoijana viimeinen asia, jolla haluat luokkasi käyttäjää rasittaa on se, että hänen pitäisi tietää luokkasi sisäinen rakenne. Se olisi kuin juoma-automaatti, jonka valmistaja vaatisia sinua ymmärtämään automaatin sisäisen toiminnan ennen kuin antaisi ostamasi pullon.

Toiseksi, tämä koodi ei tue uudelleenkäyttöä. Heti kun alat ymmärtää, miten periytyminen tukee uudelleenkäyttöä, huomaat, että luokat ja objektit ovat hyvä asia. Tässä tapauksessa määrittelisit kaikki jäsenet kantaluokassa, joka toimisi samoin riippumatta työntekijän tyypistä. Jokainen periytetty luokka voisi sitten periä tämän toiminnallisuuden ja muuttaa sitä tarpeen mukaan. Esimerkkimme näyttäisi C#:lla tällaiselta:

```
class Employee
{
    public Employee(string firstName, string lastName,
                    int age, double payRate)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.payRate = payRate;
    }

    protected string firstName;
    protected string lastName;
    protected int age;
    protected double payRate;

    public double CalculatePay(int hoursWorked)
```

(jatkuu)

```
    {
        // Laskenta tehdään tässä.
        return (payRate * (double)hoursWorked);
    }
}

class SalariedEmployee : Employee
{
    public string SocialSecurityNumber;

    public void CalculatePay (int hoursWorked)
    {
        // Kuukauspalkan laskenta.
    }
}

class ContractEmployee : Employee
{
    public string FederalTaxId;

    public void CalculatePay (int hoursWorked)
    {
        // Urakoitsijan palkan laskenta.
    }
}
```

Edellä olevassa esimerkissä on kolme huomionarvoista seikkaa:

- Kantaluokka *Employee* sisältää jäsenen nimeltä *EmployeeId*, joka periytyy sekä *SalariedEmployee*- että *ContractEmployee*-luokkaan. Kaksi periytyvää luokkaa eivät tee mitään saadakseen tämän jäsenen, ne perivät sen automaattisesti *Employee*-luokasta.
- Molemmat periytyvät luokat toteuttavat oman versionsa metodista *CalculatePay*. Huomaa kuitenkin, että molemmat perivät rajapinnan ja vaikka ne muuttavat sisäistä koodiaan itselleen sopivaksi, käyttäjän koodi pysyy samana.
- Molemmat periytyvät luokat lisäävät jäsenen kantaluokasta perimiensä jäsenten lisäksi. *SalariedEmployee*-luokka määrittelee merkkijonojäsenen *SocialSecurityNumber* ja *ContractEmployee*-luokkan sisältyy jäsenen *FederalTaxId* määrittely.

Olet nähnyt tästä pienestä esimerkistä, että periytymisen ansiosta voit käyttää koodia uudelleen periyttämällä toiminnallisuuden kantaluokasta. Ja enemmänkin, sillä voit laajentaa luokkaa lisäämällä siihen omia muuttujia ja metodeja.

Asianmukaisen periytymisen määrittely

Osoittaakseni asianmukaisen peritytymisen tärkeyden, käytän Marshall Clinen ja Greg Lomowin kirjassaan *C++ FAQs* (Addison-Wesley, 1998) esittämää termiä: *korvattavuus* (substitutability). Se tarkoittaa, että periytyvän luokan ilmoitettu käyttäytyminen on korvattavissa kantaluokassa. Mieti hetken tätä sääntöä, se on tärkein yksittäinen sääntö, mitä tulee toimivan luokkahierarkian rakentamiseen.

Toinen sääntö, joka kannattaa pitää mielessä, kun suunnittelee luokkahierarkkiaa, kuuluu näin: *periytyvän luokan ei pidä vaatia enempää eikä luvata vähempää kuin sen kantaluokka missään periytyvässä rajapinnassa.* Tästä säännöstä poikkeaminen rikkoo olemassa olevan koodin. Luokan rajapinta on sitova sopimus luokan itsensä ja luokkaa käyttävän koodin välillä. Kun ohjelmoijalla on viittaus periytyvään luokkaan, hän voi aina luottaa, että sitä voi käsitellä kuin kantaluokkaa. Tätä kutsutaan englanninkielisellä termillä *upcasting*. Jos asiakasohjelmalla esimerkissämme on viittaus *ContractEmployee*-objektiin, sillä on samalla viittaus sen kantaluokkaan *Employee*-objektiin. Niinpä määrityksen mukaan *ContractEmployee*-objektin tulee aina kyetä toimimaan kuin kantaluokkansa objektin. Huomioi, että tämä sääntö koskee vain kantaluokkaa. Periytetty luokka voi lisätä rajoittavia metodeja ja luvata niin vähän kuin haluaa. Siksi tätä sääntöä sovelletaan vain perittyihin jäseniin, koska olemassa olevalla koodlla on "sopimus" vain niihin.

Monimuotoisuus

Paras ja lyhin määritelmä, jonka olen kuullut monimuotoisuudesta, kuuluu näin: monimuotoisuus on toiminnallisuus, jossa vanha koodi kutsuu uutta koodia. Tämä on ilman muuta suurin etu olioperusteisesta ohjelmoinnista, koska se mahdollistaa järjestelmän laajentamisen ja parantamisen olemassa olevaa koodia muokkaamatta tai rikkomatta.

Oletetaan, että kirjoitat metodia, jonka pitää käydä läpi *Employee*-objektien kokoelma ja kutsua jokaisen *CalculatePay*-metodia. Tämä toimii hienosti, kun yhtiölläsi on vain yhden tyyppisiä työntekijöitä, koska voit silloin lisätä kokoelmaan tarkkaan määrätyn objektityypin. Mutta mitä tapahtuu, kun palkkaat uuden tyyppisen työntekijän? Jos sinulla esimerkiksi on luokka nimeltä *Employee,* joka toteuttaa kuukausipalkkaisen työntekijän toiminnallisuuden, mutta palkkaatkin tuntipalkkaisia työntekijöitä, joiden palkka pitää laskea eri tavalla? Proseduraalisessa ohjelmoinnissa muokkaisit laskentafunktiota käsittelemään uuden työntekijätyypin, jonka seurauksena vanha koodi ei ymmärtäisi, miten uutta koodia tulee käsitellä. Olioperusteinen ratkaisu käsittelee tällaiset erot monimuotoisuuden avulla.

Määrittelet kantaluokan nimeltä *Employee*. Sen jälkeen määrittelet periytyvän luokan kullekin työntekijätyypille (kuten olemme tehneet aiemmin). Kullakin periytetyllä työntekijäluokalla on sitten oma toteutuksena *CalculatePay*-metodista. Nyt tapahtuu ihmeitä. Monimuotoisuuden ansiosta, kun sinulla on ylöspäin tyyppimuunnettu (upcasted) -osoitin objektiin ja kutsut objektin metodia, kielen ajonaikainen ympäristö varmistaa, että kutsutaan metodin oikeaa versiota. Tässä koodi, joka selventää tätä asiaa:

```
using System;

class Employee
{
    public Employee(string firstName, string lastName,
                    int age, double payRate)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.payRate = payRate;
    }

    protected string firstName;
    protected string lastName;
    protected int age;
    protected double payRate;

    public virtual double CalculatePay(int hoursWorked)
    {
        Console.WriteLine("Employee.CalculatePay");
        return 42; // satunnaisluku
    }
}

class SalariedEmployee : Employee
{
    public SalariedEmployee(string firstName, string lastName,
                            int age, double payRate)
    : base(firstName, lastName, age, payRate)
    {}

    public override double CalculatePay(int hoursWorked)
    {
        Console.WriteLine("SalariedEmployee.CalculatePay");
        return 42; // satunnaisluku
    }
}

class ContractorEmployee : Employee
```

```
{
    public ContractorEmployee(string firstName, string lastName,
                              int age, double payRate)
    : base(firstName, lastName, age, payRate)
    {}

    public override double CalculatePay(int hoursWorked)
    {
        Console.WriteLine(“ContractorEmployee.CalculatePay”);
        return 42; // bogus value
    }
}

class HourlyEmployee : Employee
{
    public HourlyEmployee(string firstName, string lastName,
                          int age, double payRate)
    : base(firstName, lastName, age, payRate)
    {}

    public override double CalculatePay(int hoursWorked)
    {
        Console.WriteLine(“HourlyEmployee.CalculatePay”);
        return 42; // bogus value
    }
}

class PolyApp
{
    protected Employee[] employees;

    protected void LoadEmployees()
    {
        Console.WriteLine(“Loading employees...”);

        // Todellisessa sovelluksessa saisimme
        // tämän tietokannasta.
        employees = new Employee[3];

        employees[0] = new SalariedEmployee (“Amy", “Anderson", 28, 100);
        employees[1] = new ContractorEmployee (“John", “Maffei", 35, 110);
        employees[2] = new HourlyEmployee (“Lani", “Ota", 2000, 5);

        Console.WriteLine(“\n”);
    }

    protected void CalculatePay()
```

(jatkuu)

```
    {
        foreach(Employee emp in employees)
        {
            emp.CalculatePay(40);
        }
    }

    public static void Main()
    {
        PolyApp app = new PolyApp();

        app.LoadEmployees();
        app.CalculatePay();
    }
}
```

Ohjelman kääntäminen ja ajaminen aiheuttaa seuraavat tulokset:

```
c:\>PolyApp

Loading employees...

SalariedEmployee.CalculatePay
ContractorEmployee.CalculatePay
HourlyEmployee.CalculatePay
```

Huomaa, että monimuotoisuus tarjoaa vähintään kaksi etua. Ensinnäkin se antaa mahdollisuuden ryhmitellä objektit, joilla on yhteinen kantaluokka ja käsitellä niitä samalla tavalla. Vaikka minulla yllä olevassa esimerkissä oli teknisessä mielessä kolme erilaista objektityyppiä, *SalariedEmployee*, *ContractorEmployee* ja *HourlyEmployee*, pystyin käsittelemään niitä kaikkia *Employee*-objektina, koska ne kaikki periytyivät *Employee*-kantaluokasta. Siten pystyin käsittelemään niitä taulukossa, joka oli määritelty *Employee* -objektien taulukoksi. Kun kutsuin yhtä noiden objektien metodeista, monimuotoisuuden ansiosta ajonakainen ympäristö varmisti, että kutsuttiin oikean periytetyn luokan metodia.

Toinen etu on se, jonka mainitsin kappaleen alussa: vanha koodi voi käyttää uutta koodia. Huomaa, että *PolyApp.CalculatePay*-metodi käy läpi luokan *Employee*-objektien jäsentaulukon. Koska tämä metodi käsittelee objekteja *Employee*-objekteina ja monimuotoisuuden ajonaikainen toteutus varmistaa, että oikean periytetyn luokan metodia kutsutaan, voin lisätä muita periytettyjä työntekijätyyppejä järjestelmään, lisätä ne *Employee* objektitaulukkoon ja kaikki koodini toimii edelleen ilman, että olen joutunut muuttamaan alkuperäistä koodia lainkaan!

Yhteenveto

Tämä luku on vienyt sinut pikaiselle matkalle niiden termien ja käsitteiden pariin, jotka sisältyvät termin olioperusteinen ohjelmointi alle. Aiheen täydellinen käsittely vaatisi useita kappaleita ja veisi siksi sivuun tämän kirjan varsinaisesta aiheesta. Olioperusteisen ohjelmoinnin perustuntemus on kuitenkin tarpeellista, jotta saisit kaiken irti C#-kielestä.

Käsittelimme tässä luvussa vain muutamia aiheita. Olioperusteisen järjestelmän ymmärtämisen perusedellytys on erottaa luokat, objektit ja rajapinnat ja tietää, miten nämä liittyvät tehokkaaseen ratkaisuun. Hyvä olioperusteinen ratkaisu riippuu myös kolmen olioperusteisen ohjelmoinnin perusasian toteutuksesta. Nämä kolme ovat: kapselointi, periytyminen ja monimuotoisuus. Tässä luvussa esitetyt periaatteet luovat pohjan seuraavalle luvulle, joka esittelee Microsoft .NET Frameworkin ja Common Runtime Libraryn.