Tampereen ammattikorkeakoulu

Funktionaalinen ohjelmointi

Ville-Veikko Nieminen

Opinnäytetyö
Marraskuu 2020

Tietojenkäsittely
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

NIEMINEN, VILLE-VEIKKO:
Funktionaalinen Ohjelmointi

Opinnäytetyö 30 sivua
Marraskuu 2020

Opinnäytetyössä käsitellään funktionaalista ohjelmointia yhtenä ohjelmointiparadigmana. Työnhaussa termi nousee esiin jatkuvasti, joten opinnäytetyössä haluttiin avata käsitettä laajemmin. Tämän opinnäytetyön tarkoituksena on kertoa lukijalle taustaa yleisesti käytetyistä ohjelmointiparadigmoista, keskittyen erityisesti funktionaaliseen ohjelmointiin. Opinnäytetyössä aihetta lähestytään mahdollisimman selkeästi. Tavoitteena on täyttää lukijan tarpeet ymmärtää, mitä funktionaalinen ohjelmointi on. Opinnäytetyössä ei käytetä empiirisiä aineistoja, vain aiheeseen liittyvää kirjallisuutta. Työssä vertaillaan eri ohjelmointiparadigmojen eroavaisuuksia.

Opinnäytetyössä pyritään käsittelemään funktionaalista ohjelmointia siten, että lukijan on helppoa rakentaa käsitys eri ohjelmistoparadigmojen eroista. Työssä selvitetään myös funktionaalisen ohjelmoinnin yleisiä periaatteita, jotta lukija saa yleiskuvan siitä, mitä funktionaalinen ohjelmointi on.

Funktionaalisen ohjelmoinnin rinnalle opinnäytetyöhön valittiin käsiteltäviksi proseduraalinen ohjelmointi sekä olio-ohjelmointi. Funktionaalisen ohjelmointiparadigman tapaan nämä ovat laajasti käytettyjä. Proseduraalinen ohjelmointi on monelle ensimmäinen tapa tutustua ohjelmointiin. Olio-ohjelmointi vastaavasti on yksi yleisimmistä ohjelmointiparadigmoista ja on saavuttanut laajan suosion myös opetuksessa.

Tulevaisuudessa aihetta voitaisiin käsitellä laajemmin esimerkiksi tutkimalla eri ohjelmointiparadigmojen käyttöä tietyissä toteutuksissa tai tarkastelemalla syvemmin funktionaalisen ohjelmoinnin käsitettä ja mahdollisuuksia. Myös aiheen lähestyminen ja esimerkkien esittäminen puhtaasti funktionaalisia ohjelmointikieliä käyttäen (mm. Haskell) voisi olla hyvä tapa syventyä aiheeseen tulevaisuudessa syvemmin.

Asiasanat: funktionaalinen ohjelmointi, ohjelmointiparadigma

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Option of Software Development

NIEMINEN, VILLE-VEIKKO:
Functional Programming

Bachelor's thesis 30 pages
November 2020

The objective of this Bachelor’s thesis was to find the real definition of functional programming as a one of the programming paradigms. The term has repeatedly come up in job notices. The purpose of the thesis is to provide some knowledge about commonly used programming paradigms and their differences, but mainly keep focus on functional programming.

The aim of this study is to handle the subject in such a way that it is easy for the reader to build a basic knowledge about the differences between different paradigms. This thesis offers understanding of what functional programming is and when to use it.

In the future, the topic could be approached more broadly by studying the real use of different programming paradigms in certain implementations. It is possible to look much more deeply at the concept and possibilities of the functional programming, and one possible way to approach it could be to use only pure functional programming languages and not a hybrid language like was done in this thesis.

Key words: functional programming, programming paradigm

SISÄLLYS

LYHENTEET JA TERMIT

ohjelmointiparadigma	Tyyli tai tapa tavoittaa haluttu lopputulos ohjelmoinnissa.
Lambda	Anonyymi, eli nimetön funktio. Java-ohjelmointikielen osalta kuten metodi, jolla ei ole nimeä ja jota voidaan käyttää parametrina. Java 8-versiossa tullut ominaisuus.
metodiviite	Yksinkertaistettu tapa toteuttaa Lamba-lauseke. Java 8-versiossa tullut ominaisuus.
rekursio	Rekursiivinen funktio kutsuu itseään jatkuvasti uudestaan, kunnes sille asetettu ehto täyttyy.
puhdas funktio	Puhtaassa funktiossa ei ole sivuvaikutuksia, eli se ei muuta ohjelman tilaa.
evaluointimalli	Malli, jolla ohjelma arvioi lausekkeen pätevyyden.
luokkakaavio	Kaavio, jonka avulla kuvataan olio-ohjelmointikielien luokkia ja niiden suhteet toisiinsa.

.

1 JOHDANTO

Funktionaalinen ohjelmointi on termi, joka nousee esiin melko usein niin työnhakuilmoituksissa kuin ohjelmoinnista kertovissa artikkeleissa ja kirjoituksissakin. Moni ohjelmistoalasta ja ohjelmoinnista kiinnostunut varmasti uskoo ymmärtävänsä termin sisällön ja tarkoituksen, ainakin pääpiirteittäin.

Funktionaalinen ohjelmointi on yksi paljon käytetyistä paradigmoista, ja sen historia yltää melko kauas, jopa 1930-luvulle. Ensimmäinen virallinen funktionaalinen ohjelmointikieli LISP, syntyi kuitenkin vasta 1950-luvulla. (Räty 2020.) Vaikka funktionaalista ohjelmointia käytetään paljon ja termi nousee usein esiin, se ei ole kuitenkaan yhtä tunnettu kuin esimerkiksi olio-ohjelmointi. Olio-ohjelmointi saa esimerkiksi koulutuksessa ja opetuksessa huomattavasti suuremman roolin, kun taas funktionaalinen ohjelmointi jää usein syrjään.

Työn tarkoitus on käydä läpi tarkemmin funktionaalista ohjelmointia, sen tarkoitusta, hyötyjä ja ongelmia. Tärkeimpänä kysymyksenä on, mitä ongelmia se on toteutettu ratkaisemaan. Samalla työssä kerrotaan myös hieman muista ohjelmointiparadigmoista ja niiden eroista. Työssä käytetyt selventävät ja selittävät esimerkit on toteutettu ja kirjoitettu mahdollisuuksien mukaan käyttäen Java-ohjelmointi kieltä, puhtaasti siitä syystä, että se on laajalti käytetty ja sekä niin kutsutusti moniparadigmainen. Vaikka Java-ohjelmointikieli mielletään monesti puhtaaksi olio-ohjelmointikieleksi, voidaan sen avulla toteuttaa myös muita ohjelmointiparadigmoja. Opinnäytetyö ei keskity ohjelmointikieliin, vaan paradigmoihin yleisesti.

2 OHJELMISTOTUOTANTO

Java, metodi, funktionaalinen ohjelmointi, bugi, versionhallinta, syntaksi. Mikäli tunnistat yhden tai useamman termin aidon merkityksen, sinulla on mahdollisesti jonkinlainen ymmärrys mistä ohjelmistotuotannossa pääpiirteittäin on kyse. Useimmiten, kun keskustelu ajautuu ohjelmistotuotantoon sellaisen henkilön kanssa, jolla ei ole juurikaan kosketuspintaa aiheeseen, hänellä ei ole aidosti minkäänlaista käsitystä siitä, mitä ohjelmistotaloissa tehdään ja mitä tietojenkäsittelijän tavallinen työpäivä voi pitää sisällään.

Sanomalehdistä voi lukea kuinka kaksi autotallissa työskentelevää ohjelmoijaa on rakentanut vaikuttavan ohjelman, joka ylittää suurien tiimien ponnistelujen tuotokset. Ohjelmoijat ovat valmiita uskomaan nämä tarinat ja uskovat voivansa rakentaa minkälaisen ohjelman tahansa. (Brooks 1995, 4.) Monesti kuvitelmat isojen ohjelmistojen toteuttamisesta saattavat olla hyvin epärealistisia. Pienemmänkin laajemmin käyttöön tulevan ohjelmiston toteutus niin, että se on ylläpidettävä ja luotettava, vaatii huomattavasti enemmän kuin vain muutaman taitavan ohjelmoijan.

Mikäli kuvitelmat eivät vastaa realistisia odotuksia, tiimin tietotaidosta tai koosta riippumatta, jossakin kohtaa tekemistä luultavasti juututaan paikoilleen tai kohdataan ongelmia, jotka ovat myöhäistä taklata. Kun puhutaan ammattimaisesta ja uraauurtavasta kehityksestä, se pitää sisällään myös tavoitteissa, aikataulussa ja budjetissa pysymisen. Ohjelmiston tulee olla ylläpidettävä niin, että se palvelee muutaman vuoden jälkeenkin ja sitä voidaan päivittää ja uudistaa tarpeiden mukaan.

Miksi ohjelmointi on hauskaa, ja mitä sen harjoittaja voi odottaa siitä ansaitsevansa (Brooks 1995, 7). Moni voi varmasti samaistua tunteeseen, jonka saavuttaa, kun saa itse toteuttaa jotakin toimivaa ja näkyvää. Etenkin, jos se on jotakin sellaista, joka ratkaisee ongelmia, täyttää tarpeen tai tuottaa muuten jollekin aitoa hyötyä ja iloa. Erilaiset aivopähkinät tai ratkaistavat pulmat ja palapelit ovat aina kiinnostaneet ihmisiä juuri onnistumisentunteen vuoksi, jonka kokee, kun löytää ratkaisun. Ohjelmistokehittäjän työskentelyssä näitä onnistumisentunteita

voi kokea jatkuvasti. Muutoinkin jatkuva oppiminen ja itsensä kehittäminen on hyvin vahvasti läsnä. Nämä ovat aitoja tunteita, ja myös Brooks on osan näistä maininnut esittämäänsä kysymykseen vastauksena.

Mikään ei ikinä ole täysin vaivatonta, vaan myös ohjelmistotuotannon haasteet saattavat olla todella hermoja raastavia sekä turhautumista aiheuttavia. Kun kohdataan jokin tarpeeksi suuri haaste tai tarve jollekin uudelle, aletaan kiihkeästi etsimään parempaa keinoa tai ratkaisua. Suurin osa erilaisista menetelmistä ja tekniikoista on syntynyt oikeaan tarpeeseen, näin myös funktionaalisen ohjelmoinnin osalta.

Itse ohjelmointikieliä on kymmeniä erilaisia, erilaisiin tarkoituksiin. Silti yksikään niistä ei ole yleispätevä, kaikkeen sopiva. Nykypäivien ohjelmointikielet ovat vaatineet tavattoman paljon työtä kehittyäkseen tasolle, jolla ne ovat. Varhaisimmat kielet ja ohjelmointitavat ovat olleet sellaisia, että ei niitä voi juurikaan verrata nykypäivän menetelmiin. Nykypäivänä ohjelmoija voi tehdä hienoja ja haastavia ohjelmistoja ilman, että juurikaan edes ymmärtää syvempää logiikkaa, jonka esimerkiksi ohjelmointikielen kääntäjät tekevät kääntäessään kirjoitetun koodin sellaiseksi, että tietokone sitä ymmärtää.

Miksi on tärkeää ymmärtää, mitä tekniikoita, kieliä ja ohjelmistoja hyödynnetään missäkin projektissa? On mahdollista, että ajatellaan olevan helpompaa toteuttaa projekti tai työ samoilla välineillä ja tavoilla, kuten aikaisempikin hyvin sujunut toteutus. Tämä ei aina pidä paikkaansa. Kuten todettu, mikään ohjelmointikieli tai tekniikka ei ole yleispätevä. Tämä pätee varmasti kaikkeen tekemiseen. Huonot työkalut tekevät työstä vähemmän tehokasta ja vaikuttavat suuresti työn tekemisen mielekkyyteen. Juuri tästä syystä on tärkeää ylläpitää taitoja ja käsitystä erilaisista työkaluista, vaikka niitä ei juuri tässä hetkessä kaipaisikaan. Mikäli eteen tulee esimerkiksi tilanne, jossa työskentelee jonkin vanhemman, tai vastaavasti uudemman toteutuksen parissa mihin yleensä on tottunut, voi törmätä tilanteeseen, että ei yksinkertaisesti voi käyttää omasta työkalupakista löytyviä välineitä.

3 MITÄ ON FUNKTIONAALINEN OHJELMOINTI?

3.1 Yleisesti

Ohjelmistotuotannossa, kuten IT-alalla yleisestikin, tapahtuu jatkuvasti kaikkea uutta. Tekniikat ja ohjelmistot kehittyvät ja alalla työskentelevien täytyy sisäistää paljon uutta kiihtyvällä tahdilla. Tästä huolimatta vielä vanhat, ison vaikutuksen tehneet tekniikat ja menetelmät ovat levinneet niin laajalle, että niitä ei sovi unohtaa, päinvastoin.

Funktionaalinen ohjelmointi on ohjelmointiparadigma, jossa vältetään tilanmuutoksia ja datan muuttamista funktioiden suorittamisen aikana (Peltomäki 2019, 7). Käytännössä tämä tarkoittaa sitä, että funktioiden lopputulos ei riipu ulkoisista tekijöistä, vaan sille annetuista argumenteista. Samat funktiolle annetut argumentit siis johtavat aina samaan lopputulemaan. Tiivistettynä voisi todeta, että funktionaalisessa ohjelmoinnissa pyritään välttämään kaikenlaisia sivuvaikutuksia. Sivuvaikutukseksi kutsutaan tilannetta, jossa funktion suorituksen aikana muutetaan jotakin funktion ulkopuolista tilaa pysyvästi. ”Sivuvaikutusten välttäminen ei tarkoita sitä, ettei käytetä tietorakenteita, vaikka funktionaalisessa ohjelmoinnissa pyritään käyttämään muuntamattomia tietorakenteita (unmodifiable collections), joiden tila ei muutu suoritusten aikana.” (Peltomäki 2019, 8.)

Ohjelmointikieli voi olla funktionaalinen, vaikka siinä esiintyisi sivuvaikutuksia, mutta on olemassa myös niin sanottuja puhtaita funktionaalisia ohjelmointikieliä, kuten Haskell. Puhtaasti funktionaalisissa ohjelmissa ei tilaa muuteta lainkaan funktioiden toimesta. Täysin puhtaasti funktionaalisessa koodissa jo syötteen ja tulosteen toteuttaminen on hankalaa, siksi usein sivuvaikutusten salliminen on järkevää käytännöllisten syiden takia.

Ohjelmointikielen voidaan ajatella olevan funktionaalinen, jos sillä lähtökohtaisesti ohjelmoidaan kuten puhtaasti funktionaalisilla kielillä. Suurin osa funktionaalisista ohjelmointikielistä sallii tilamuuttujat ja sivuvaikutukset.

3.2 Tausta

Funktionaalisen ohjelmoinnin juuret ulottuvat hyvin pitkälle, jopa 1930-luvulla kehitettyyn Lambda-laskentaan. Lambda-laskenta tai sen ymmärtäminen ei itsessään ole välttämättömyys sen käyttämiseen. (Peltomäki 2019, 7.)

Lisp-ohjelmointikieli (1958) toi ohjelmointiajatteluun funktionaalisen paradigman, jossa käytetään matemaattisia funktioita. Lisp ei ollut ensimmäinen näin tekevä kieli, mutta sitä kuitenkin pidetään kielenä, joka teki siitä tunnetun. (Räty 2020.) Lisp on yksi vanhimmista ohjelmointikielistä ja pitää pintansa edelleen suosituiden ohjelmointikielten parissa.

Saavutettuaan suosiota funktionaalinen ohjelmointi koki kovia 1970-luvulla, kun ohjelmat muuttuivat yksinkertaisesta matematiikasta enemmänkin vaihe vaiheelta suoritettaviksi ohjeiksi, joita tietokone noudattaa. Kuitenkin jälleen muutama vuosikymmen myöhemmin erinäisten trendien nostettuaan päätään funktionaalinen ohjelmointi nousi esiin ja saavutti jatkuvasti suurempaa suosiota. (Elliot 2017.)

Melko harva funktionaalisista ohjelmointikielistä on raivannut tiensä suosituimpien ja käytetyimpien ohjelmointikielten joukkoon. Funktionaalinen paradigma itsessään on yleistynyt, mutta eritoten sellaisissa kielissä, jotka ovat niin kutsuttuja hybiridikieliä. Hybridikielet toteuttavat useamman ohjelmointiparadigman. Useimmiten hybridikielet toteuttavat olio-ohjelmoinnin sekä funktionaalisen paradigman. Tällaisia kieliä ovat esimerkiksi Kotlin, Swift ja C#.

Kuten Tungin kirjoittamasta artikkelista käy ilmi, on Java yksi suosituimmista ohjelmointikielistä (Tung 2020). Javaan funktionaalinen ohjelmointi tuli olio-ohjelmoinnin rinnalle Java SE 8 -versiossa vuonna 2014 (Peltomäki 2019, 11), ja nämä ominaisuudet eivät ainakaan haitanneet Javan suosiota. Funktionaalinen ohjelmointi on nykypäivänä yhä useamman ohjelmoijan hallittava jollakin tasolla jo siitä syystä, että se saa jalansijaa maailman suosituimpien ohjelmointikielien toteutuksessa. Java lukeutuukin Java SE 8 -version myötä hybridikieleksi muiden aikaisemmin mainittujen ja monien muiden ohjelmointikielten lisäksi. Java

ohjelmointikielellä toteutetuissa ohjelmistoissa käytetäänkin useimmiten näitä molempia paradigmoja, funktionaalista ja oliopohjaista, rinnakkain.

3.3 Piirteet

Yleisimpiä ja selkeimpiä funktionaalisen ohjelmoinnin piirteitä ovat globaalien muuttujien välttäminen ja puuttumattomuus. Muuttujat, joita yleisesti funktionaalisessa ohjelmoinnissa käytetään, ovat muuttumattomia, jotta sivuvaikutuksilta vältyttäisiin tehokkaammin. Tärkeintä funktionaalisessa ohjelmoinnissa on jo aikaisemmin esiin tullut sivuvaikutusten välttäminen.

Funktionaalisessa ohjelmoinnissa esiintyy hieman tavallista useammin tilanteita, joissa funktiot palauttavat ja ottavat parametreinään funktioita. Funktioiden saamat argumentit ja palauttamat arvot ovat funktionaalisessa ohjelmoinnissa lähtökohtaisesti tyypitetty. Mikäli näitä ei tyypitettäisi, ei funktioita ole määritelty kunnolla.

Funktionaalisen ohjelmoinnin kautta koodi on usein uudelleenkäytettävää, koska funktioiden palauttamat arvot tai toiminta ei ole riippuvaisia ulkopuolisista tekijöistä. Koska ulkoiset tekijät eivät lopputulokseen vaikuta, voidaan uudelleenkäytettävyyteen luottaa.

3.3.1 Rekursio

Funktionaalisessa ohjelmoinnissa paljon hyödynnetty ja käytetty rekursio, tilanne, jossa jokin funktio tai aliohjelma kutsuu itse itseään uusin parametrein tai ehdoin, on paljon muistia syövä ominaisuus, mutta siitä huolimatta paljon käytetty ominaisuus. Joissakin tapauksissa funktio saattaa kutsua itseään lukuisia kertoja ja lopettaakin vasta tietyn ehdon täytyttyä. Esimerkki yksinkertaisesta rekursiosta voisi olla kertoman laskeminen. Metodi *kertoma()* kutsuu itseään arvolla, joka on sille annettu kokonaisluku vähennettynä yhdellä, kunnes ehto (*arvo <=1*) täytyy eikä ohjelmaa enää jatketa. Ehdon täyttyessä metodi palauttaa sille alun perin annetun argumentin kertoman:

```
public static int kertoma(int arvo) {
  if(arvo <= 1) {
    return 1;
  }
  return arvo * kertoma(arvo-1);
}
```

KOODIESIMERKKI 1. Kertoman laskeminen rekursiota hyödyntäen

3.3.2 Puhtaat funktiot

Funktionaalisessa ohjelmoinnissa puhtaat funktiot ovat erittäin suuressa roolissa. Kuten jo aikaisemmin on tullut esiin, puhtaat funktiot ovat aliohjelmia, jotka eivät muuta ohjelman tilaa. Tällaiset funktiot palauttavat aina saman lopputuloksen samoilla argumenteilla.

Seuraavassa esimerkissä on hyvin yksinkertaistettu ohjelma, jossa metodi *kertolasku* palauttaa liukulukuna sille annetun argumentin kolmella kerrottuna. Esimerkin metodi on epäpuhdas, koska se riippuu ohjelman yleisesti muuttujasta *kerroin*. Esimerkki epäpuhtaasta metodista:

```
public class Main {

  public static double kerroin = 1.6;

  public static void main(String[] args) {
    double km = 5;
    double miles = muutaKilometritMaileiksi(km);
  }

  public static double muutaKilometritMaileiksi(double km) {
    return km * kerroin;
  }
}
```

KOODIESIMERKKI 2. Epäpuhdas funktio muutaKilometritMaileiksi()

Ylemmän esimerkin (koodiesimerkki 2) metodi *muutaKilometritMaileiksi()* on epäpuhdas, koska sen palauttama arvo riippuu metodin ulkopuolisista muuttujasta, tilamuuttujasta. Samasta toteutuksesta saadaan puhdas siirtämällä muuttuja *kerroin* metodin sisäiseksi muuttujaksi, jolloin metodin palauttama arvo ei voi muuttua, mikäli argumentti säilyy samana:

```
public static double muutaKilometritMaileiksi(double km) {
  double kerroin = 1.6;
  return km * kerroin;
}
```

KOODIESIMERKKI 3. Puhdas funktio muutaKilometritMaileiksi()

3.3.3 Funktiot ensimmäisen luokan kansalaisia

Funktionaalisessa ohjelmoinnissa yleisesti funktiot ovat niin kutsuttuja ”ensimmäisen luokan kansalaisia”. Tämä tarkoittaa, että funktioita voidaan käyttää kuten muitakin arvoja tai muuttujia. Niitä voi antaa toisille funktioille argumentteina tai niitä voi palauttaa funktion lopputuloksena. Tämä on tärkeä ominaisuus, joka mahdollistaa funktioiden monipuolisemman käsittelyn. Tämä ominaisuus on hyvin ominainen piirre funktionaalisessa ohjelmoinnissa ja sitä pystytään hyödyntämään hieman eri lailla riippuen käytettävästä ohjelmointikielestä. Java-ohjelmointikielen osalta metodien antaminen argumentteina ei ole aivan yhtä suoraviivaista kuin muiden kielten osalta. Kuitenkin Java 8-versiossa tulleiden Lamba-lausekkeiden ja metodiviitteiden myötä pääsemme aika lähelle sitä.

Seuraavassa esimerkissä on esitetty yksi tilanne, jossa metodi annetaan parametrinä metodille. Esimerkissä on toteutettu listan suodatus hyödyntämällä *Predicate* rajapintaa, joka saadaan normaalisti Java:n kirjastoista. Esimerkissä se on esitetty selventämään tilannetta. Esimerkin tilanne, jossa annetaan metodi argumenttina, mahdollistaa useiden erilaisten *boolean* tyyppisten metodien toteuttamisen ja siinä määritellyiden sääntöjen mukaan listan suodattamisen ilman suurta koodin toistamista. Esimerkin auki selostaminen menisi herkästi enemmänkin Java-ohjelmointikielen ominaisuuksien avaamiseen, mutta tiivistettynä *suodataLista()* -metodi ottaa vastaan listan, joka sisältää kokonaislukuja sekä *boolean* arvon palauttavan metodin. Koodin toistaminen vähenee, mikäli toteutukseen lisätään muita erilaisia tapoja suodattaa listaa. Esimerkki on toteutettu metodiviitettä hyödyntäen:

```
public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> lista = new ArrayList<Integer>();
    lista.add(5);
    lista.add(9);
    lista.add(8);
    lista.add(1);
    ArrayList<Integer> suodatettuLista
      = suodataLista(lista, Main::onJaollinenKahdella);
  }

  public interface Predicate<T> {
    public boolean testi(T t);
  }

  public static ArrayList<Integer> suodataLista(
    ArrayList<Integer> lista, Predicate<Integer> p) {

    ArrayList<Integer> suodatettuLista = new ArrayList<>();
      for(int luku : lista) {
        if(p.test(luku)) {
          suodatettuLista.add(luku);
        }
      }
      return suodatettuLista;
    }

    public static boolean onJaollinenKahdella(int luku) {
      return luku % 2 == 0;
    }
}
```

KOODIESIMERKKI 4. Metodin antaminen argumenttina

Metodin *suodataLista()* kutsun voisi toteuttaa samassa ohjelmassa myös käyttämällä metodiviitteen sijaan Lamba-lauseketta:

```
public static void main(String[] args) {
  ArrayList<Integer> lista = new ArrayList<Integer>();
  lista.add(5);
  lista.add(9);
  lista.add(8);
  lista.add(1);
  Predicate<Integer> suodatin = (luku) ->
    onJaollinenKahdella(luku);

  ArrayList<Integer> suodatettuLista =
    suodataLista(lista, suodatin);
}
```

KOODIESIMERKKI 5. Ohjelman käyttäminen Lambda-lausekkeen avulla

3.3.4 Evaluointimalli

Evaluointimalleja on kaksi toisistaan poikkeavaa, tiukka (*strict*) ja laiska (*lazy*). Tiukassa evaluointimallissa lausekkeet eli funktiot evaluoidaan aina riippumatta siitä, missä vaiheessa ohjelmaa sitä kutsutaan (Heinonen 1996). Tämä johtaa siihen, että mikäli funktiossa on jokin kääntämisen estämä virhe, tulee se esiin jo ohjelman kääntövaiheessa eikä vasta sitä ajaessa. Tiukan evaluointimallin hyöty on virheiden esiin nouseminen aikaisessa vaiheessa, mutta pakottaa samalla ohjelmoijaa määrittelemään funktiot ohjelmointivaiheessa tarkasti.

Laiska evaluointimalli evaluoi funktiot vasta, kun se on tarpeellista. Tällöin mahdolliset virheet funktioissa nousevat esiin ajonaikaisesti. Laiska evaluointimalli nopeuttaa ohjelmaa, koska sen ei tarvitse tehdä työtä turhaan, mikäli joitakin funktioita ei kutsuta ohjelman ajossa. Laiska evaluointimalli mahdollistaa myös äärettömien tietorakenteiden muodostamisen (Heinonen 1996). Funktionaalisessa ohjelmoinnissa ja sen toteuttavissa ohjelmointikielissä hyödynnetään laajalti laiskaa mallia ja se on yksi sen olennaisista piirteistä.

4 MIHIN FUNKTIONAALISTA OHJELMOINTIA KÄYTETÄÄN

Toiset näkevät, että funktionaalinen ohjelmointi on paras paradigma käytettäväksi ongelmaan kuin ongelmaan. Ohjelmointiparadigmojen paremmuutta ei voi suoraan verrata keskenään, vaan aina on otettava huomioon konteksti sekä projekti, jota ollaan toteuttamassa.

Tiivistettynä funktionaalinen ohjelmointi eroaa muista ohjelmointiparadigmoista siten, että monissa muissa ongelma pyritään ratkaisemaan kirjoittamalla vaiheet tarkasti askel askeleelta. Funktionaalisessa ohjelmoinnissa pyritään hyödyntämään matemaattisia funktioita ilman, että tarvitsee miettiä sitä, miten kone nämä vaiheet suorittaisi (Mueller 2019, 4). Funktionaalisessa ohjelmoinnissa käytetyssä matemaattisessa lähestymistavassa voi siis keskittyä ennemmin ongelman kuvaukseen, kuin itse ratkaisuun. Lähestymistavassa voidaan toteuttaa kuvauksen mukaisia vaiheita, eikä niinkään miettiä ongelmanratkaisua vaihe vaiheelta.

4.1 Hyödyt

Funktionaalisen ohjelmoinnin pyrkimys välttää tilanmuutoksia helpottaa olennaisesti koodin ymmärtämistä ja mahdollista virheen etsintää. Mikäli ohjelmassa on sivuvaikutuksia, on mahdotonta selvittää tai tietää, mitä siinä tapahtuu ja mitkä ovat sen aikomukset ilman, että pääse käsiksi lähdekoodeihin ja selvittää ohjelman aikomukset sitä lukemalla.

On hyvin vaikeaa seurata esimerkiksi globaalin tilan muuttumista ja elinkaarta, kun sitä muutetaan erinäisissä funktioissa. Funktionaalisessa ohjelmoinnissa tämän voi selvittää pienellä vaivalla funktioiden tarkastelulla ja mahdollisten dokumentaatioiden lukemisella.

4.2 Ongelmat

Kaikkiin käyttötarkoituksiin funktionaalista ohjelmointia ei voi hyödyntää. Tällaisia ovat mm. I/O (input/output) toteutukset, joissa otetaan esimerkiksi käyttäjän syötteitä ja hyödynnetään niitä ohjelmassa. I/O ohjelmointi lähestulkoon makaa sivuvaikutusten varassa.

Funktionaaliseen ohjelmointiin ja sen piirteisiin kuuluva rekursio voi joskus osoittautua haasteeksi sen toteuttajalle. Jos ohjelmoija ei ole tullut tutuksi rekursion kanssa, saattaa sen toteuttaminen olla haastavaa ilman, että se turhaan syö resursseja tai, että sen toiminta on varmasti virheetöntä. On tärkeää määritellä tällaiset toiminnot huolella, jotta esimerkiksi lopetusehto on oikeanlainen, eikä rekursio voi jatkua loputtomiin.

Usein funktionaalisen ohjelmoinnin piirteet ja ominaisuudet kääntyvät ongelmiksi, mikäli niiden käyttäminen ei ole tuttua tai määrittelyä ei ole toteutettu kunnolla. Kuten aikaisemmin on todettu, sivuvaikutusten jonkin asteinen salliminen helpottaa huomattavasti funktionaalista ohjelman toteutusta.

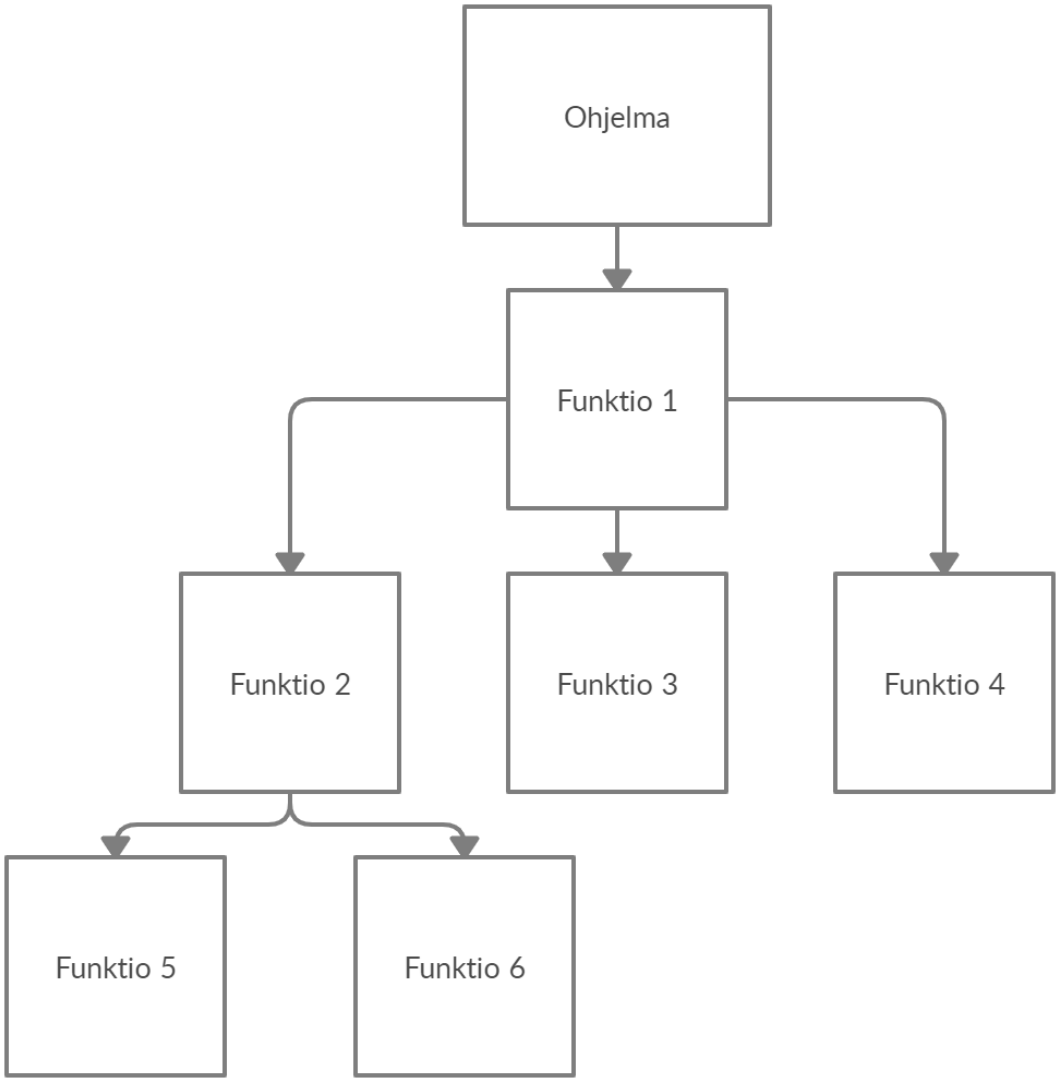
5 OHJELMOINTIPARADIGMAT

5.1 Proseduraalinen ohjelmointi

Erilaiset ohjelmointiparadigmat edustavat erilaisia tapoja lähestyä ongelmaa ja sen ratkaisemista. Proseduraalisessa ohjelmoinnissa pyritään saavuttamaan ohjelman ymmärrettävyyttä ja luettavuutta.

Proseduraalisessa ohjelmoinnissa pääasiallisena ideana on jakaa ohjelman toiminallisuutta niin kutsuttuihin aliohjelmiin, proseduureihin. Haluttu lopputulos tavoitetaan kutsumalla tarvittavia proseduureja vaihe vaiheelta. Tällainen proseduuri on täysin itsenäinen osa, jota voidaan kutsua itse pääohjelmasta tai muista aliohjelmista. Nämä aliohjelmat mahdollistavat uudelleenkäytettävyyden näille ohjelmille. Aliohjelmille voidaan antaa käsiteltävä data joko parametreinä, tai ne voivat olla globaaleja muuttujia itse funktion ulkopuolella. Toki relevantimpaa on antaa funktion tarvitsema data parametrina, jotta ne säilyvät helpommin testattavina ja uudelleenkäytettävämpinä.

Proseduraalisessa ohjelmoinnissa voidaan jakaa ohjelman osia jatkuvasti erilaisiin aliohjelmiin (kuvio 1), kunnes nämä ovat hyvin pieniä ja ysinkertaisia. Lähtökohtaisesti proseduraalista ohjelmointia voidaan pitää melko helposti ymmärrettävänä ja ylläpidettävänä. Kun toteutus on hajotettu pieniin osuuksiin, on mahdollisen vian etsiminen ja erityisesti korjaaminen huomattavasti helpompaa.



KUVIO 1. Proseduraalisen ohjelman hajottaminen pienempiin aliohjelmiin

Muutamin yksinkertaisin esimerkein voimme havainnollistaa proseduraalisessa ohjelmoinnissa paljon käytettyä toiminnallisuuksien pilkkomista. Seuraavat esimerkit on toteutettu Java-ohjelmointikielellä. Esimerkeissä on yksinkertainen etunimen ja sukunimen tulostus:

```
public class Main {

  public static void main(String []args) {
    String etunimi = "Essi";
    String sukunimi = "Esimerkki";

    System.out.println(etunimi +" "+ sukunimi);
  }
}
```

KOODIESIMERKKI 6. Yksinkertainen tulostus

Yllä olevassa esimerkissä (koodiesimerkki 6) on mahdollisesti yksinkertaisin tapa tulostaa kaksi muuttujaa. Seuraavaksi kärjistetty esimerkki, miten kyseisen yksinkertaisen toteutuksen voisi pilkkoa proseduureihin:

```
public class Main {

  public static void main(String []args) {
    String etunimi = "Essi";
    String sukunimi = "Esimerkki";
    tulosta(etunimi, sukunimi);
  }

  public static void tulosta(String etunimi,
    String sukunimi) {

    tulosta(etunimi);
    tulosta(" ");
    tulosta(sukunimi);
    tulosta("\n");
  }

  public static void tulosta(String value) {
      System.out.print(value);
  }
}
```

KOODIESIMERKKI 7. Yksinkertainen proseduraalinen ohjelma

Tässä yksinkertaisessa proseduraalisessa ohjelmassa (koodiesimerkki 7) on siis pilkottu tulostamista useampaan proseduuriin. Proseduurit käyttävät ainoastaan saamiaan parametrejä, sekä viimeisessä proseduurissa hyödynnetään globaalia metodia *System.out.println().*

5.1.1 Käyttökohteita

Proseduraalinen ohjelmointi on erittäin pätevä vaihtoehto pienempiin ohjelmistoihin, joissa toimenpiteet ovat hyvin suoria ja ehdottomia. Proseduraalisella ohjelmoinnilla pystytään vähentämään myös koodin toistoa tehokkaasti ohjelmissa, joissa käytetään samanlaisia toimintoja useissa eri paikoissa.

Tämä saattaa olla monille aloitteleville ohjelmoijille ohjelmointiparadigmana ensimmäinen opittava, sillä se on helposti ymmärrettävä ja sisäistettävä. Proseduraalinen on ohjelmointiparadigmana melko suoraviivaista, koska siinä kerrotaan koneelle, miten suorittaa toiminnot loogisesti järjestyksessä. Suoraviivaisuus tekee siitä mainion paradigman moniin pienempiin toteutuksiin.

5.1.2 Oleellisimmat erot ja piirteet

Proseduraalisen ohjelmoinnin haasteet tulevat esiin, kun puhutaan suuremmista ja laajemmista ohjelmistoista. Tällaisissa pieniksi pilkotut suoritukset johtavat siihen, että proseduureja on huomattavan suuri määrä. Mikäli nämä useat proseduurit käyttävät samaa globaalina olevaa dataa, on myös vaikeaa jäljittää datan käyttöä ja ylläpitää ohjelmakoodia.

Koska proseduraalisessa ohjelmoinnissa nämä proseduurit ovat useasti koko ohjelmistolle näkyviä, jotta niitä voidaan hyödyntää laajasti, saattaa se aiheuttaa ylimääräistä huolenaihetta tietoturvan puolesta. Proseduraalista ohjelmointia harkittaessa tulee tämän ja monen muun seikan lisäksi ottaa huomioon sen vaatimat resurssit.

5.2 Olio-ohjelmointi

Olio-ohjelmointi on eittämättä yksi maailman käytetyimmistä ja suosituimmista ohjelmointiparadigmoista. Olio-ohjelmoinnissa käytetään hyödyksi erilaisia olioita, jotka ovat tietorakenteita ja voivat sisältää niille yhteisiä muuttujia sekä metodeja. Monesti oliot ovat tosielämästä tuttuja ilmiöitä, kuten esimerkiksi *Ihminen* tai *Nisäkäs*. Eri oliot voivat myös periytyä jostakin toisesta oliosta, esimerkin mukaisesti ihminen voisi periytyä nisäkkäästä. Tällöin *Ihmisellä* on käytettävissä kaikki *nisäkkäille* yleisesti yhteiset muuttujat sekä metodit. Lisäksi *ihmisellä* voi olla vielä omia tietoja ja toiminnallisuuksia. *Nisäkkäällä* voi esimerkiksi olla muuttujina *paino* ja *ikä*, sekä toiminnallisuuksina *syö* ja *synnytä*. Tällöin *Ihmisestä* luodulla oliolla on pääsy näihin, mutta sillä voi olla vielä omat toiminnalli-

suudet kuten, *puhu* tai *työskentele*. Usein olio-ohjelmoinnissa ohjelmoija tarvitsee vain tietää mihin käyttöön olio on luotu ja miten sitä kuuluu käyttää, olennaista ei ole ollenkaan tietää tarkasti, minkälaista ohjelmakoodia olio pitää sisällään.

Olio-ohjelmoinnin yksi suurimmista hyödyistä on uudelleenkäytettävyys. Yhtä ja samaa oliota voidaan tilanteen mukaan hyödyntää todella monenlaisissa eri projekteissa. Ohessa Java-ohjelmointikielellä yksinkertainen esimerkki olioiden käytöstä:

```
abstract class Nisakas {
  private int ika;
  private double paino;

  public Nisakas(int ika, double paino) {
    this.ika = ika;
    this.paino = paino;
  }

  public void syo() {
    System.out.println("Nisäkäs syö");
  }

  public void synnyta() {
    System.out.println("Nisäkäs synnyttää");
  }
}
```

KOODIESIMERKKI 8. Nisakas luokka

Abstrakti luokka *Nisakas* on luokka, josta ei suoraan voi luoda omaa oliota, mutta josta voidaan periyttää jokin toinen olio. Voimme ajatella, että tosielämässäkään ei ole olemassa lajia nimeltä nisäkäs, vaan nisäkäs on käsite, joka sisältää useita alalajeja. Seuraavassa luokka *Ihminen:*

```
class Ihminen extends Nisakas {
  private String etunimi;
  private String sukunimi;

  public Ihminen(int ika, double paino) {
    super(ika, paino);
    this.etunimi = etunimi;
    this.sukunimi = sukunimi;
  }

  public Ihminen(int ika, double paino,
    String etunimi, String sukunimi) {

    super(ika, paino);
    this.etunimi = etunimi;
    this.sukunimi = sukunimi;
  }

  public void asetaNimi(String etunimi, String sukunimi) {
    this.etunimi = etunimi;
    this.sukunimi = sukunimi;
  }

  public void puhu() {
    System.out.println("Ihminen puhuu");
  }

  public void tyoskentele() {
    System.out.println("Ihminen työskentelee");
  }

  public void toString() {
    System.out.println(etunimi + ” ” + sukunimi);
  }
}
```

KOODIESIMERKKI 9. Ihminen luokka

Luokka *Ihminen* siis periytyy *Nisäkkäästä,* jolloin sillä on olemassa kaikki samat muuttujat ja toiminnot kuin yläluokallaan. Sen lisäksi kyseisessä esimerkissä (koodiesimerkki 9) *Ihmisellä* on olemassa sellaisia muuttujia ja toimintoja, joita ei välttämättä ole kaikilla *Nisäkkäästä* periytyvillä olioilla. Tässä tapauksessa sellaisia ovat muuttuja *nimi*, sekä toiminnot *puhu* ja *tyoskentele*. Seuraavaksi esimerkki, miten voimme luoda ja käyttää oliota käytännössä:

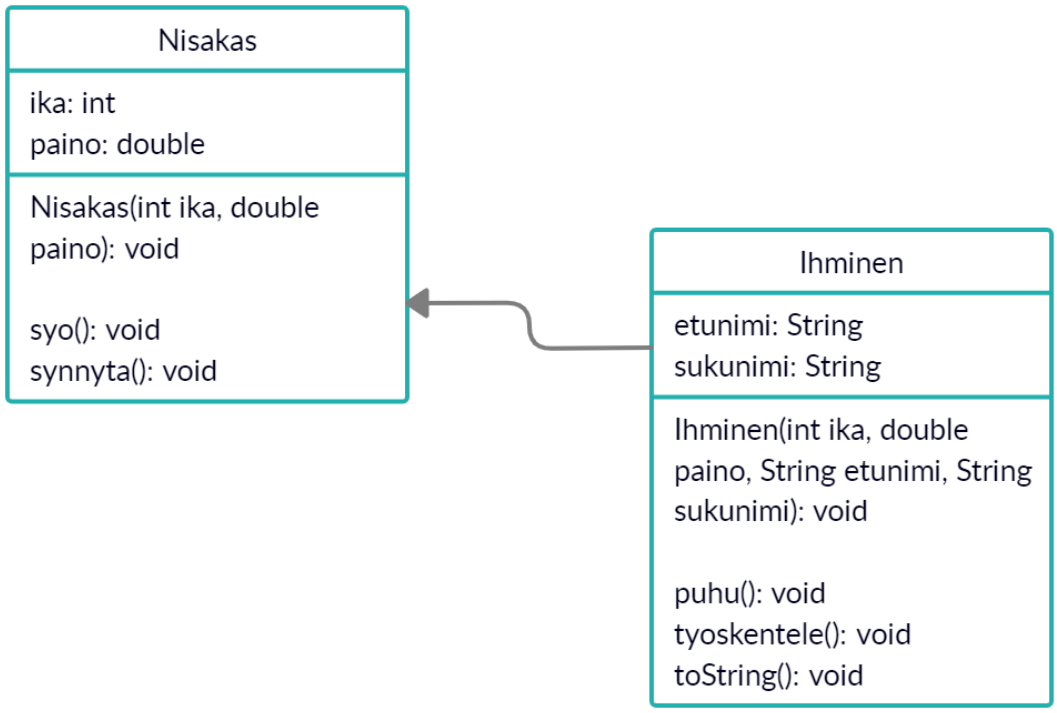
```
class Main {
  public static void main(String args[]) {

    Ihminen kaisa = new Ihminen(42, 75.6);
    kaisa.asetaNimi("Kaisa", ”Kaikkonen”);

    kaisa.tyoskentele();
    kaisa.synnyta();
  }
}
```

KOODIESIMERKKI 10. Main luokka

Voimme siis luoda olion luokasta *Ihminen*, jolloin voimme kutsua kyseistä oliota käyttäen toimintoja, jotka ovat *Nisakas* luokan tai *Ihminen* luokan sisässä. Esimerkin (koodiesimerkki 10) ohjelma tulostaa siis ”Ihminen työskentelee” sekä ”Nisäkäs synnyttää”.

Käytetyt esimerkit ovat hyvin karkeita, jotka vain pintapuolisesti esittää olioiden käyttämistä. Kyseinenkään tilanne ei ole tosielämässä aukoton, sillä kaikki *Ihminen* -luokan oliot eivät välttämättä ole kykeneväisiä synnyttämään, mikäli otetaan huomioon myös sukupuoli.

Olio-ohjelmointia hyödyntäen voimme käyttää Ihminen-luokan oliota kuvaamaan ihmistä ja sen toString()-metodia tulostamaan kyseiselle oliolle asetetut nimet. Kyseistä esimerkkiä voimme verrata aikaisempaan proseduraalisen ohjelmoinnin esimerkkiin ja päätellä, kuinka suuri apu olio-ohjelmoinnista on, mikäli ohjelmassa käytetään paljon samantyyppistä dataa eri arvoilla.
Mikäli ohjelmassa perimishierarkia kasvaa, voidaan tilannetta havainnollistaa luomalla luokkakaavio (kuvio 2).



KUVIO 2. Luokkakaavio, josta käy ilmi luokkarakenne ja perimishierarkia

5.2.1 Käyttökohteita

Olio-ohjelmoinnin suurin hyöty, kuten mainittu, on uudelleenkäytettävyys. Uudelleenkäytettävyys taas mahdollistaa huomattavan säästön itse kehitykseen panostettavasta ajasta ja vaivasta sekä testauksesta. Perimishierarkia voi toki aiheuttaa epäselvyyttäkin. Kun perimieshierarkia ylettyy kauas, on työlästä selvittää ilman oikeanlaisia dokumentaatioita, mitä jokin metodi pitää sisällään, ja mitä se oikeasti tekee.

Olio-ohjelmointia hyödynnetään usein ohjelmissa, joissa hyödynnetään useassa paikassa samantyyppistä koodia ja toimintoja. Olio-ohjelmoinnilla voidaan välttää koodin osalta duplikaatteja, jolloin myös ylläpidettävyys paranee huomattavasti. Mahdolliset muutokset voi tehdä vain yhteen luokkaan, jolloin muutos vaikuttaa kaikkiin siitä luotuihin olioihin. Kun luokasta luodaan olio, eli instanssi kyseisestä luokasta, allokoi se itselleen osan muistista ja voi sisältää juuri kyseiselle oliolle ominaiset arvot (Sarcar 2019, 2).

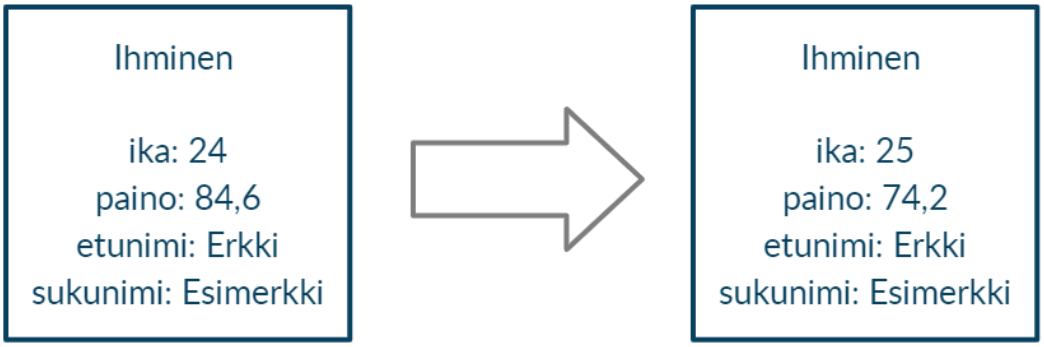
5.2.2 Oleellisimmat erot ja piirteet

Periytymisen lisäksi olennainen piirre olio-ohjelmoinnissa on kapselointi, tietojen ja toimintojen sisällyttäminen yhteen kokonaisuuteen, kuten luokkaan (Sarcar 2019, 1). Kapseloinnin ja hyvän dokumentaation vuoksi tällaisen luokan käyttäjän ei tarvitse tarkkaan tietää, miten esimerkiksi jotkin toiminnot on toteutettu vaan ne voidaan piilottaa. Kapseloinnin avulla voidaan parantaa ohjelmiston turvallisuutta, piilottamalla nämä toiminnot nähtäviltä. Kun toiminnot ja tiedot on koottu yhteen luokkaan, voidaan sitä käyttää mahdollisuuksien mukaan helposti muualla, jolloin myös uudelleenkäytettävyys paranee.

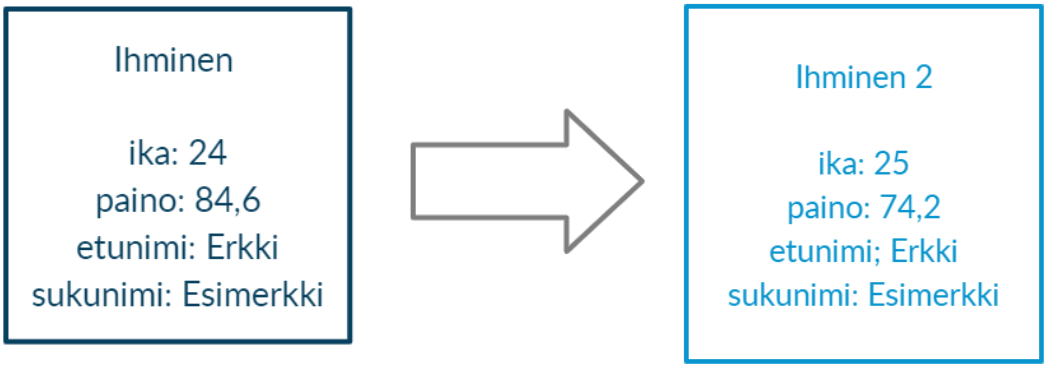
Polymorfismi on edellä mainittujen lisäksi olennainen piirre olio-ohjelmoinnissa. Polymorfismi tarkoittaa olioiden monimuotoisuutta. Olio voi edustaa kaikkia perintöhierarkiasta ja toteutuneista rajapinnasta löytyviä eri olioita. Tämä tarkoittaa käytännössä sitä, että periytetyn olion voi antaa esimerkiksi parametrinä metodille, joka ottaa vastaan jonkin yläluokan olioista. Esimerkiksi *Ihminen* luokan olion voi antaa parametriksi metodille, joka ottaa vastaan *Nisakas* luokan olion.

Olio-ohjelmoinnissa on piirteitä, jotka voivat aiheuttaa ongelmia. Mikäli jokin luokka sisältää jonkin virheen ohjelmakoodissa tai rakenteessa muuten, periytyy tämä virhe kaikille olioille, jotka tästä periytyvät. Siksi onkin tärkeää, että luokat on huolellisesti testattu. On myös etu, että kun olioluokka on testattu asiallisesti, ei sitä tarvitse enää uudelleenkäytön kohdalla testata.

Toisin kuin funktionaalisessa ohjelmoinnissa olio-ohjelmoinnissa ei usein pyritä ollenkaan välttämään tilan muuttamista metodien suorituksen aikana. Olio-ohjelmoinnissa se päinvastoin on hyvin tavallista ja ominaista. Hyvä esimerkki tästä on arvon asettamiset, kuten aikaisemmissa esimerkeissä ja muut *void*-tyyppiset metodit, joilla ei ole paluuarvoa ollenkaan. Arvon asettamisen lisäksi jo valmiiksi asetettuja arvoja voidaan myös muuttaa saman olion sisällä (kuvio 3). Jos yhdistäisimme toteutukseen funktionaalisen ajattelutavan ja välttäisimme tilan muuttumista, loisimme uuden olion, jolle arvot asetettaisiin (kuvio 4).



KUVIO 3. Olio-ohjelmoinnissa olioiden tietoja voidaan muuttaa suoraan luotuun olioon



KUVIO 4. Funktionaalisessa ohjelmoinnissa tietojen muutos toteutettaisiin luomalla uusi olio

6 POHDINTA

Opinnäytetyö luotiin tarjoamaan tietoa ja ymmärrystä erilaisten ohjelmointiparadigmojen osalta, keskittyen funktionaalisen ohjelmoinnin paradigmaan. Opinnäytetyössä käsiteltävät käsitteet nousevat usein esiin työnhakuilmoituksissa ja ohjelmistoalaan liittyvissä kirjoituksissa. Opinnäytetyö tarjoaa ymmärryksen erilaisten ohjelmistoparadigmojen olennaisimmista eroista sekä funktionaalisen ohjelmoinnin merkityksestä. Työllä ei ollut tilaajaa, eikä sitä ole toteutettu yhteistyössä minkään toimeksiantajan kanssa.

Opinnäytetyön tavoite on tutkia ja tuoda esiin perusajatukset suosituimpien ohjelmistoparadigmojen perusperiaatteista, niiden hyödyistä ja niihin liittyvistä ongelmista. Tavoite on tuoda havainnot esiin jäsennellysti ja helposti ymmärrettävässä muodossa.

Opinnäytetyö ja sen tarjoama hyöty on suunnattu sellaisille lukijoille, jotka tahtovat täyttää tarvetta ymmärtää alalla paljon käytettyä perusterminologiaa. Työ ei tarjoa syvällistä tai ohjeistavaa informaatiota funktionaalisen ohjelmoinnin toteuttamiseen.

Opinnäytetyössä käytettyihin lähteisiin kuuluu ohjelmistoalan kirjallisuus, artikkelit, seminaarit sekä e-kirjat. Lähteinä käytetään niin kotimaisia kuin englanninkielisiä aineistoja. Osa käytetyistä aineistoista käsittelee yleisesti ohjelmistoalaa, osa keskittyen eri ohjelmistoparadigmoihin ja niiden käyttöön. Jotta opinnäytetyö on luotettava, on samaa aihetta käsitteleviä lähteitä useampia. Luotettavuutta lisäämään on työssä käytetty lähdeviittauksia reilusti.

Työn suurimmaksi haasteeksi osoittautui olennaisten poimintojen löytäminen ilman, että työ käsittelee aihetta liian syvällisesti. Esimerkeissä käytetty ohjelmointikieli (Java) asettaa myös omat haasteensa, koska se ei ole puhtaasti funktionaalinen ohjelmointikieli.

Opinnäytetyön tekeminen opetti ohjelmoinnin historiasta ja erilaisten ohjelmistoparadigmojen synnystä. Myös lähteenä käytetty kirjallisuus opetti ohjelmistoalasta ja sen kokemista muutoksista yleisesti. Olio-ohjelmointi oli ennestään

tuttu paradigma, mutta funktionaalisesta ja proseduraalisesta ohjelmoinnista opittiin työn myötä paljon uutta.

Opinnäytetyössä olisi voitu käsitellä ohjelmointiparadigmoja hieman syvällisemmin sekä niihin liittyvät esimerkit olisi voitu avata laajemmin. Esimerkeissä käytetty ohjelmointikieli (Java) on laajasti käytetty, mutta esimerkkien avaaminen sanallisesti olisi tarjonnut paremman käsityksen sen esittelemästä tilanteesta, jolloin Java-ohjelmointikielen osaamistaso ei asettaisi esteitä niiden ymmärtämiselle.

Opinnäytetyön jatkokehitysideana voitaisiin aihetta käsitellä syvällisemmin tutkien erilaisten ohjelmistoparadigmojen käyttöä oikeissa toteutuksissa. Näin työ tarjoaisi lukijalle laajemman kuvan siitä, miten eri ohjelmistoparadigmat tosielämän toteutuksissa eroavat toisistaan. Jatkokehityksenä voitaisiin myös keskittyä vielä tarkemmin funktionaaliseen ohjelmointiin ja lähestyä aihetta puhtaiden funktionaalisten ohjelmointikielien avulla. Näin erilaiset funktionaaliselle ohjelmoinnille tyypilliset piirteet ja toiminnot olisi helpompi tuoda esiin.

LÄHTEET

Brooks, Frederick P. 1995. The Mythical Man-Month. Anniversary Edition. Boston: Addison-Wesley Pub.

Elliot, E. 2017. The Rise and Fall and Rise of Functional Programming (Composing Software). Luettu 3.11.2020. https://medium.com/javascript-scene/the-rise-and-fall-and-rise-of-functional-programming-composable-software-c2d91b424c8c

Heinonen, M. 1996. Funktionaalinen ohjelmointi. Luettu 3.11.2020. http://www.mit.jyu.fi/opiskelu/seminaarit/ohjelmistotekniikka/funktion/#2.2

Mueller, P. 2019. Functional Programming For Dummies. 1st edition. Newark: Wiley. https://learning.oreilly.com/library/view/functional-programming-for/9781119527503/c01.xhtml

Peltomäki, J. 2019. Funktionaalinen ohjelmointi Javalla. Helsinki: BoD – Books nn Demand.

Räty, P. 2020. Ohjelmointikielten historia – parasta kieltä ei ole vielä keksitty. Luettu 28.09.2020. Vaatii käyttöoikeuden. https://www.mikrobitti.fi/uutiset/ohjelmointikielten-historia-parasta-kielta-ei-ole-viela-keksitty/35712163-2373-493a-b870-959bbc62c78f

Sarcar, V. 2019. Interactive Object-Oriented Programming in Java: Learn and test Your Programming Skills. 2nd edition. Berkeley: CA: Apress L.P. https://learning.oreilly.com/library/view/interactive-object-oriented-programming/9781484254042/html/Cover.xhtml

Tung, L. 2020. Top programming languages: Python still rules but old Cobol gets a pandemic bump. Luettu 10.10.2020. https://www.zdnet.com/article/top-programming-languages-python-rules-still-but-old-cobol-gets-a-pandemic-bump/