Karelia-ammattikorkeakoulu
Tradenomi (AMK), Tietojenkäsittely

Funktionaalisen ohjelmoinnin hyödyt ja haitat

Ohjelmistoparadigman vahvuudet ja heikkoudet

Toni Cantarella

Opinnäytetyö, huhtikuu 2024

www.karelia.fi

Karelia AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Huhtikuu 2024
Tietojenkäsittelyn koulutus

Tikkarinne 9
80200 JOENSUU
+358 13 260 600

Tekijä(t)
Toni Cantarella

Nimeke
Funktionaalisen ohjelmoinnin hyödyt ja haitat

Toimeksiantaja
Toimeksiantajayhteisön nimi

Tiivistelmä

Ohjelmointia voidaan toteuttaa monella eri tavalla. Eri ongelmanratkaisumenetelmille on luotu toisistaan eriäviä ohjelmointiparadigmoja. Eriävät tavat mallintaa sovelluskehitystä tuovat mukanaan vastakkaisia mieltymyksiä ja näkökantoja siihen, millä tavalla mitäkin ratkaisua tulisi käyttää missäkin tilanteessa.

Opinnäytetyön tavoite on tutkia artikkeleista löytyviä väittämiä funktionaalisen ohjelmoinnin puolesta ja vastaan ja testata niiden paikkaansa pitävyyttä. Väittämiä testataan verraten funktionaalista ja olio-ohjelmointia keskenään. Kummankin paradigman käsitteitä testataan toteuttamalla joko tyypillinen ohjelmistoon liittyvä ongelmanratkaisu tai opetusmateriaalista löytyvä ohjelmointitehtävä. Testattavat mittarit keskittyvät tuotetun koodin luettavuuteen ja käytettyyn aikaan jokaisen ohjelmistotehtävän kohdalla.

Opinnäytetyön tulokset viittaavat siihen, että kummallakaan paradigmalla ei ole merkittäviä hyötyjä tai haittoja luettavuuden tai ohjelmointiin käytetyn ajan suhteen. Kuten kaikki mielipiteet ja näkemykset, kannanotot ohjelmointiparadigmoista ovat enimmäkseen subjektiivisia. Kuitenkin verrattuna muihin paradigmoihin, funktionaalinen ohjelmointi on vahvimmillaan laskelmointiin liittyvissä tehtävissä, kuten matematiikassa ja data-analytiikassa.

Kieli	Sivuja 33
suomi	Liitteet 5
	Liitesivumäärä 17

Asiasanat
ohjelmointi, funktionaalisuus, paradigma

Karelia
UNIVERSITY OF APPLIED SCIENCES

THESIS
April 2024
Information Technology

Tikkarinne 9
80200 JOENSUU
FINLAND
+ 358 13 260 600

Author (s)
Toni Cantarella

Title
The Pros and Cons of Functional Programming

Commissioned by
Commissioner

Abstract

Programming can be implemented in many ways. Different problem-solving methods have brought upon unique kinds of programming paradigms. These distinct implementations to model software development inevitably create opposing schools of thought in terms of what paradigms should be used to solve each use case.

The aim of this thesis is to investigate the validity of the arguments found in opinionated articles opposing and supporting functional programming. To parse the validity of these views, programming is implemented in functional and object-oriented programming. Each paradigms features are tested in either common programming tasks or problem-solving tasks found in teaching material. Focus is on the readability of the produced code and time spent on each task.

The results of this thesis suggest that there are no meaningful pros or cons in terms of code readability or time spent in programming with either paradigm. As with all opinionated views, the opinions on programming paradigms are mostly subjective. However, in comparison to other paradigms, functional programming strives in tasks involving calculations, such as mathematics and statistical analysis.

Language
Finnish

Pages 33
Appendices 5
Pages of Appendices 17

Keywords
programming, functionality, paradigm

Sisältö

1 Johdanto 5
2 Tietoperusta 6
2.1 Mitä on funktionaalinen ohjelmointi 6
2.1.1 Muuttumattomuus 6
2.1.2 Puhdas funktio 7
2.1.3 Funktioiden kompositio 7
2.1.4 Rekursio 8
2.1.5 Funktionaalisen ohjelmoinnin käsitteet käytännössä 9
2.2 Mihin funktionaalinen ohjelmointi soveltuu 10
2.3 Funktionaalista ohjelmointia koskevat näkemykset 11
2.3.1 Näkemykset puolesta 11
2.3.2 Näkemykset vastaan 12
3 Tavoite ja tehtävä 13
3.1 Innovaation mahdollistaminen 14
3.1.1 Paradigma laatikon ulkopuolella 14
3.1.2 Hyöty opiskelussa 15
4 Menetelmät 15
4.1 Ohjelmointitehtävät 15
4.2 Vertailtavat käsitteet 15
4.2.1 Oliot/luokat ja funktiot 16
4.2.2 Iteraatio ja rekursio 16
4.2.3 Periytyvyys ja funktioiden kompositio 16
4.2.4 Muuttujat ja muuttumattomuus 17
4.2.5 Tila ja tilattomuus 17
4.3 Vertailutavat ja mittaus 17
4.3.1 Käytetty aika 18
4.3.2 Luettavuus 18
5 Luotettavuus ja eettisyys 20
5.1 Subjektiiviset mittarit 20
6 Ohjelmistotestaus 20
6.1 Oliot/luokat ja funktiot 20
6.1.1 Oliopohjainen ratkaisu 20
6.1.2 Funktionaalinen ratkaisu 21
6.2 Iteraatio ja rekursio 22
6.2.1 Oliopohjainen ratkaisu 22
6.2.2 Funktionaalinen ratkaisu 23
6.3 Periytyvyys ja funktioiden kompositio 24
6.3.1 Oliopohjainen ratkaisu 25
6.3.2 Funktionaalinen ratkaisu 26
6.4 Muuttujat ja muuttumattomuus 27
6.4.1 Oliopohjainen ratkaisu 27
6.4.2 Funktionaalinen ratkaisu 28
6.5 Tila ja tilattomuus 29
6.5.1 Oliopohjainen ratkaisu 29
6.5.2 Funktionaalinen ratkaisu 31
7 Tulokset 34
7.1 Oliot/Luokat ja funktiot 34

7.1.1 Aika 34
7.1.2 Kognitiivinen kompleksisuus 34
7.1.3 Syklomaattinen kompleksisuus 34
7.2 Iteraatio ja rekursio 35
7.2.1 Aika 35
7.2.2 Kognitiivinen kompleksisuus 35
7.2.3 Syklomaattinen kompleksisuus 36
7.3 Periytyvyys ja funktioiden kompositio 37
7.3.1 Aika 37
7.3.2 Kognitiivinen kompleksisuus 37
7.3.3 Syklomaattinen kompleksisuus 38
7.4 Muuttujat ja muuttumattomuus 38
7.4.1 Aika 38
7.4.2 Kognitiivinen kompleksisuus 39
7.4.3 Syklomaattinen kompleksisuus 39
7.5 Tila ja tilattomuus 40
7.5.1 Aika 40
7.5.2 Kognitiivinen kompleksisuus 40
7.5.3 Syklomaattinen kompleksisuus 41
7.6 Taulukko 42
8 Pohdinta 43
8.1 Väitteiden ja näkemysten pätevyys 43
8.1.1 Hyödyt 43
8.1.2 Haitat 43
8.2 Testauksen havainnot 44
8.3 Katse tulevaisuuteen 44
8.3.1 Ohjelmistokehitys 45
8.3.2 Opiskelu 45
8.3.3 Paradigmat ja niiden rajat 45
Lähteet 47

Liitteet

1 Johdanto

Ohjelmointia, kuten mitä tahansa kehitystä on mahdollista toteuttaa erilaisin menetelmin (Liyan 2023). Jotta kehityksen kaareen voitaisiin lisätä tehokkuutta, nopeutta ja yhtenäisyyttä, on luotava rajat sille, miten ohjelmoinnin keskeisintä osaa, eli ongelmanratkaisua pyritään toteuttamaan. Tällaisten työmenetelmien mallintamiseen on ohjelmistokehityksessä pitkään käytetty erinäisiä ohjelmointiparadigmoja, jotka ovat käytännössä tapoja ajatella ja toteuttaa ohjelmointia, niiden jäsentelyä sekä rakennetta. Paradigmat nojautuvat periaatteisiin, käsitteistöihin ja ajattelumalleihin, joihin ohjelmoijan tulee asettua ennen toteuttamista.

Ohjelmointiparadigmoja on monia, joista keskeisinä opinnäytetyössä toimivat funktionaalinen- ja olio-ohjelmointi. Näiden lisäksi on kuitenkin olemassa esimerkiksi imperatiivinen, deklaratiivinen, proseduraalinen ja looginen ohjelmointi, mutta kyseiset paradigmat ovat rajattu tämän opinnäytetyön ulkopuolelle. Eri ohjelmointiparadigmoja on usein pyritty yhdistelemään toistensa kanssa, eikä olekaan olemassa rajausta sille, miten eri paradigmojen käyttöä tulisi erotella tai rajoittaa (Liyan 2023).

Vaikka paradigmoja on useita, saman ohjelmointiin liittyvään ongelman ratkaisuun voidaan yleensä luoda toteutus millä tahansa ohjelmointiparadigmalla. On kuitenkin tärkeää huomioida, että kaikki paradigmat eivät sovellu yhtä hyvin kaikkiin ongelmanratkaisutilanteisiin (Liyan 2023) ja tästä seuraakin se, että ohjelmoijien kesken esiintyy paljon erimielisyyttä siitä, minkä paradigman kanssa ohjelmointia tulisi tehdä. Opinnäytetyö käsittelee funktionaalisen ohjelmoinnin arvoa ohjelmistokehityksen maailmassa, keskustelua eri näkökulmista funktionaalisesta ohjelmoinnista ja sen sopivuudesta eri tehtäviin, sekä vertailla funktionaalista ohjelmointia käytännön esimerkkien kautta olio-ohjelmoinnin paradigmaan. Lopuksi kerätään yhteenveto ja tarkastellaan, miten funktionaalinen ohjelmointi sopii sovelluskehityksessä olio-ohjelmoinnin rinnalle, vai tarjoaako se jotain, mitä olio-ohjelmoinnilta jää puuttumaan.

2 Tietoperusta

2.1 Mitä on funktionaalinen ohjelmointi

Funktionaalinen ohjelmointi on ohjelmointiparadigma, jonka lähestymistapa ohjelmointiin koostuu yksittäisten funktioiden soveltamisesta ja kokoamisesta uusiksi monimutkaisemmiksi funktioiksi. Deklaratiivisena paradigmana, se pyrkii kuvailemaan ohjelmiston logiikkaa ja rakennetta, toisin kuin imperatiivisessa lähestymistavassa, jossa kuvaillaan ohjelmiston tilaa ja sen suuntaa (Thulie 2024).

Kuten muutkin ohjelmointiparadigmat, sisältää myös funktionaalinen ohjelmointi omia käsitteitään ja tapoja mallintaa ratkaisuja, joiden kautta se rajoittaa ohjelmoijaa paradigman ehtoihin.

2.1.1 Muuttumattomuus

Perinteisemmässä ohjelmointitavassa on tyypillistä huomata, kuinka muuttujien arvojen muokkaaminen ja vaihtuminen ohjelmiston suorituksen aikana on osa ohjelmiston rakennetta. Funktionaalisen ohjelmoinnin kohdalla on tavoiteltu niin sanottua muuttumattomuutta, jossa muuttujien arvoa ei muuteta muuttujan luomisvaiheen jälkeen. Aivan kuten matemaattisissa funktioissa, ohjelmisto toimii täten ennustettavasti tuottaen samalla syötteellä aina saman lopputuloksen.

Jos siis funktiolle annetaan syöte n, on funktion palautusarvo aina sama huolimatta siitä, kuinka useaan otteeseen funktiota kutsutaan syötteellä n. Muissa paradigmoissa on mahdollista, että sama funktio tai metodi tuottaa eri palautusarvon samalla syötteellä riippuen siitä, kuinka usein funktiota on kutsuttu aikaisemmin. Tämä johtuu esimerkiksi siitä, että funktion tai metodin sisällä on määritetty jokin muuttuja, jonka arvo on eri riippuen siitä, milloin sitä kutsutaan tai palautusarvo voi vaihtua ohjelman tilan muuttumisen seurauksena suorituksen

aikana, joka vaikuttaa myös funktion sisäisiin ominaisuuksiin, sekä täten palautusarvoon.

2.1.2 Puhdas funktio

Puhdas funktio tuottaa aina saman palautusarvon huolimatta siitä, miten useaan otteeseen funktiota kutsutaan, mikäli syöte on sama jokaisella kutsukerralla (Thulie 2024). Puhtaan funktion määritelmään kuuluukin siis edellä mainittu muuttumattomuus, kuitenkin sen lisäksi täyttääkseen puhtaan funktion määritelmän, täytyy funktion olla tuottamatta sivuvaikutuksia (Thulie 2024).

Sivuvaikutukset ohjelmistossa ovat funktioiden ja muun ohjelmiston osien vuorovaikutusten tuloksena seuranneita muutoksia (Ruiz 2022). Esimerkkejä sivuvaikutuksista ovat mm. ohjelmiston tilan muuttuminen, tiedoston sisällön muokkaaminen, vaikutukset laitteisiin ja tiedon siirtäminen. Puhdas funktio ei vaikuta siis itsensä ulkopuolella mihinkään, vaan palauttaa ennustettavan arvon annetun syötteen vastineeksi, vaikuttamatta ympäristöönsä.

2.1.3 Funktioiden kompositio

Funktio on ohjelmiston osa, joka suorittaa toimintoja ennalta määritellyllä tavalla. Funktiolle voidaan antaa syötteitä ja vastaanottaa sen suorituksen jälkeen palautusarvo. Funktiolle annettuja syötteitä kutsutaan parametreiksi, joita voivat olla esimerkiksi erilaiset arvot, kuten numeeriset tai tekstisyötteet. Funktionaalisessa ohjelmoinnissa funktion syötteeksi voidaan antaa toisia funktioita, joihin perustuen pääfunktio suorittaa omia toimintojaan. Tätä kutsutaan funktioiden kompositioksi.

Kompositio on tapa yhdistellä funktioita toisiinsa, antaen yhden funktion palautusarvon syötteeksi pääfunktion seuraavalle askeleelle (Anand Akhil 2023). Verrattuna olio-ohjelmointiin, jossa ohjelma kertoo, miten ohjelman eri askeleet

etenevät ja muuttavat ohjelman tilaa, funktionaalisen ohjelmoinnin komposition päämäärä on kuvailla, mitä eri askeleet tekevät. Funktioiden yhdisteleminen auttaa koodia olemaan uudelleenkäytettävää, jolloin yhden funktion toiminnallisuutta voidaan käyttää toisen funktion sisällä sen sijaan, että sen tarjoama toiminnallisuus kirjoitettaisiin uuteen funktioon toistamiseen.

2.1.4 Rekursio

Iterointi on tapa toistaa määriteltyä toimintoa, kunnes lopetusehto täyttyy. Ohjelmoinnissa iterointia voidaan suorittaa niin kutsutuilla silmukoilla. Rekursio on tapa toteuttaa iterointia, jossa funktio kutsuu itse itseään, kunnes itsensä sisällä määritelty lopetusehto täyttyy. Toisin kuin silmukassa, jossa lopetusehto on asetettu silmukan luontivaiheessa, täytyy rekursiivisen funktion määritellä lopetusehtonsa itsensä sisällä.

Klassinen esimerkki listojen iteroinnista voidaan siis toteuttaa ”for”-silmukkaa tai rekursiota käyttäen. Kuvitellaan ostoslista, jossa jokaisen tuotteen kohdalla on luku, joka kertoo, kuinka monta kappaletta kutakin tuotetta täytyy ostaa. Tahdomme nyt lisätä jokaisen ostettavan tuotteen määrää yhdellä. ”For”-silmukan kulku määritellään jo sen luontivaiheessa ja lisäysoperaatio vasta sen sisällä. Siteerattu koodi on itsetuotettua.

```
const ostosLista = [
    {tuote: ”maito”, määrä: 1},
    {tuote: ”leipä”, määrä: 3},
    {tuote: ”kahvi”, määrä: 2},
]

for (let i = 0; i < ostosLista.length; i++) {
    ostosLista[i].määrä += 1
}

console.log(ostosLista)
```

Suorituksen jälkeen tulostuu ostoslista, jossa jokaisen tuotteen ”määrää” on nostettu yhdellä. Rekursiivinen menetelmä sisältää lopetusehdon sekä lisäysoperaation itsensä sisällä.

```
const ostosLista = [
    {tuote: ”maito”, määrä: 1},
    {tuote: ”maito”, määrä: 1},
    {tuote: ”maito”, määrä: 1}
]

function lisääListaan(lista, index = 0) {
    const uusiLista = […lista]
    uusiLista[index].määrä += 1
    if (index == uusiLista.length - 1) {
        return uusiLista
    }
    return lisääListaan(uusiLista, index + 1)
}

console.log(lisääListaan(ostosLista))
```

Suorituksen jälkeen tulos on sama kuin klassisessa iteroinnissa. Vaikka suoritusvaiheessa tapahtuvan koodin pituus on merkittävästi suurempi, on huomioitavaa, kuinka funktioksi muodostettu osa on uudelleenkäytettävissä. Funktio ”lisääListaan” ei myöskään muuta alkuperäistä listaa toisin kuin iteratiivinen lähestymistapa, vaan se luo uuden listan palautusarvona ja alkuperäisen ostoslistan tulostaminen tuottaakin listan ilman muutettuja määriä.

2.1.5 Funktionaalisen ohjelmoinnin käsitteet käytännössä

Koska ohjelmistot koostuvat monista liikkuvista osista ja kokonaista ohjelmistoa ilman muuttuvaa tilaa on mahdotonta toteuttaa, funktionaalinen ohjelmointi jakautuu kahteen pääkategoriaan, puhdas- ja epäpuhdas funktionaalinen ohjelmointi. Puhtaassa muodossaan toteutettuna, funktionaalinen ohjelmointi

toteuttaa funktionsa ilman sivuvaikutuksia ja täyttää käsitteiden tarkkaan rajatut ehdot niitä soveltaessa. Epäpuhtaassa lähestymistavassa suodaan mahdollisuus vain osittaiseen ehtojen toteutumiseen, jolloin ohjelmistokehitys on joustavampaa kehittäjän näkökulmasta. Sivuvaikutuksia sallitaan ja tilan muokkaaminen on osa ohjelmiston toimintaa epäpuhtaassa menetelmässä.

Puhtaasti toteutetun funktionaalisen ohjelmoinnin kannattajat puolustavat paradigmaa toteamalla, että sivuvaikutusten välttäminen tuottaa vähemmän virheitä ohjelmistossa, on ennustettavampaa ja ovat täten testattavampia (John Hughes 1990, 2).

2.2 Mihin funktionaalinen ohjelmointi soveltuu

Funktionaalisen ohjelmoinnin historia juontaa juurensa matemaattisista operaatioista, kuten lambda lausekkeesta (Church 1936, 346). Tämän vuoksi, funktionaalista ohjelmointia on toteutettu tuottamalla ohjelmistoja matemaattisten tehtävien ratkaisemiseen, data-analytiikassa, tilastotieteessä, rahoitusanalyysissä ja tekoälyssä.

Tyypilliset ohjelmointikielet, joiden perusperiaate toteuttaa pääsääntöisesti olio-ohjelmointia paradigmanaan, lähtökohtaisesti rajoittavat käyttäjän tuottaman koodin olemaan muuta, kuin funktionaalista. Tästä syystä, funktionaaliselle ohjelmoinnille ja sen toteutukselle on kehitetty ohjelmointikieliä, joissa kieli kulkee funktionaalinen paradigma edellä. Näitä ovat esimerkiksi Lisp, Scheme, Clojure, Wolfram, Racket, Erlang ja Haskell (Indeed 2023). Moni näistä kielistä on yleispäteviä, eli kieliä, joilla on mahdollista toteuttaa useita paradigmoja, jotka ovat kuitenkin suunniteltu funktionaalista ohjelmointia ajatellen.

Konkreettisia käytännön esimerkkejä funktionaalisten kielten käytöstä ovat mm. Erlang, joka kehitettiin telekommunikointiyhtiö Ericssonin tuotannon kehitykseen ja nykypäivänä kommunikointiin keskittynyt alusta WhatsApp käyttää Erlangia pohjanaan (Erlang 2024). R-kieli on statistiikkaan ja graafisiin operaatioihin

erikoistunut kieli, jonka käytäntöjä ovat akateemiset tahot, talouslaskenta ja tilastotiede (Worsley Summer 2023).

2.3 Funktionaalista ohjelmointia koskevat näkemykset

Ohjelmointiparadigmojen tarkoitus on mallintaa ohjelmointia tiettyihin rajoitteisiin, joiden kautta ongelmanratkaisua toteutetaan näiden rajoitteiden sisällä. Kuten on olemassa eri paradigmoja, tulee niiden mukana myös näkemyksiä siitä, mikä on paras tapa mallintaa ohjelmointia ja mitkä menetelmät tuottavat ongelmiin parhaat ratkaisut. Näkemykset funktionaalisen ohjelmoinnin vahvuuksista ja heikkouksista verrattuna muihin paradigmoihin, kuten olio-ohjelmointiin ovat aiheena ajan saatossa jääneet varjoon ja monet puhuvatkin sen puolesta, että eri paradigmat soveltuvat parhaiten eri tehtäviin, sekä ottavat neutraalin näkökannan siitä, minkä paradigman tulisi ottaa keulakuva ohjelmoinnin viitekehyksessä. Vaikka eri paradigmat loistavatkin parhaiten eri tehtävissä, voidaan sama ongelma ratkaista eri tavoin ja jopa paradigmoja yhdistellen (Deren David 2019).

2.3.1 Näkemykset puolesta

Funktionaalisen ohjelmoinnin myyntipuheena toimivat usein piirteet, jotka erottavat sen muista paradigmoista, pääasiallisesti puhtaat funktiot, sivuvaikutusten puutos ja ennustettavuus. Funktionaalisen ohjelmoinnin puolustajat puhuvatkin siitä, kuinka tällainen tapa hoitaa ongelmanratkaisua luo itsessään vähemmän ongelmia ohjelmistokehityksessä. Charles Scalfani (2022) puhuu sen puolesta, että funktionaalisella lähestymistavalla voidaan välttää kompleksisuuden kasvua projektissa ja kuinka perinteisen matemaattisen ajatustavan käyttöönotto lisää koodin luettavuutta ja täten ymmärrettävyyttä. Matemaattisella ajatustavalla hän viittaa siihen, kuinka matematiikka on täynnä funktionaalisia ilmaisuja, joissa muuttujat eivät vaihda omaa tilaansa. Hänen mukaansa, kun ohjelma sisältää funktionaalisia ilmaisuja, oliomaisten komentojen sijasta, on helpompi saada

selvää siitä, mitä ohjelma milläkin hetkellä tekee ja kuinka jokin osa vaikuttaa muualla.

Richard Feldman (2019) ilmaisee puheessaan, kuinka funktionaalinen ohjelmointi selvästi on osa ohjelmistokehityksen ja sen kulttuurin tulevaisuutta. Hänen mukaansa 90-luvulla olio-ohjelmoinnin kulta-aikana, ei olisi tullut kuuloonkaan mainostaa muita ohjelmointiparadigmoja varsinkaan olio-ohjelmointiin perustuvien kielten rinnalla. Kuitenkin nykypäivänä on enenevissä määrin kirjoja, jotka ohjaavat funktionaaliseen ohjelmointiin, jopa kielillä, kuten Javalla, joiden alkuperä on oliomaisessa paradigmassa. Jopa Kotlin, Javan jälkeläinen, tukee funktionaalista ohjelmointia siinä missä oliomaista. Hänen puheensa keskeinen kysymys onkin, miksei funktionaalinen ohjelmointi nauti samankaltaista suosiota. Hän arvioi, että kyseessä on monisyinen tausta, joka voidaan kuitenkin kiteyttää pääasiallisesti sattumaan. Mikään olio-ohjelmoinnissa olennainen osa, kuten periytyvyys tai kapselointi (eng. encapsulation) ei ole sille niin uniikki, ettei vastaavaa menetelmää voitaisi toteuttaa muualla. Olio-ohjelmointi sattui hyvään aikaan ja paikkaan, jonka ansiosta sen suosio on nauttinut suurta kasvua tasaisella tahdilla.

2.3.2 Näkemykset vastaan

Kuten näkemyksillä on puoltajansa, tulee sen mukana tyypillisesti edes jonkin verran vastustusta. Puhtaasti funktionaalisen ohjelmoinnin heikkouksista on puhuttu sivuten paradigmasta kertovissa artikkeleissa, mutta niihin syventyminen on jäänyt vielä puutteelliseksi.

Alvin Alexander (2024) käsittelee paradigman heikkouksia pintaa syvemmältä. Hänen mukaansa puhtaiden funktioiden kirjoittaminen on helppoa, mutta niiden yhdistely osoittautuu haasteelliseksi, sillä sen toteuttaminen voi nopeasti johtaa sekavasti luettavaan koodiin.

Muuttumattomuuden heikkoudesta hän mainitsee, että funktionaalisen ohjelmoinnin tapa kopioida tietoa muuttamisen sijasta johtaa ongelmiin varsinkin

syvästi mallinnettujen tietorakenteiden kanssa. Jos tietomalli ”Perhe” sisältää lapsia, joilla on omia muuttujia kuten ”lelut”, on jokaisen muutoksen kopioitava tämä sisennetty tietorakenne yhä uudelleen ja uudelleen, vaikka muutettavia attribuutteja olisikin vain yksi per muutos.

Puhtaat funktiot eivät myöskään sovi hänen mukaansa yhteen käyttäjän syötteiden kanssa. Kun ohjelmisto muuttaa jotain, kuten käyttöliittymän näkymää, tietokannan arvoja, laitteiden asemaa tai yhteyksiä, se toimii epäpuhtaasti muuttaen tilaa. Täten täysin puhtaan funktionaalisen ohjelmoinnin toteuttaminen sellaisessa ohjelmistossa, jossa on mitään käyttäjäpohjaisia valintoja, on mahdotonta toteuttaa puhtaasti funktionaalisella tavalla.

Jonas Neumann (2022) yhtyy Alvin Alexanderin väitteen kohdalla suorituskyky- ja muistiongelmien ympärillä. Heidän mukaansa puhtaat funktiot ja muuttumattomuus luovat uusia muuttujia ja tietorakenteita vanhojen käyttämisen sijasta. Tämä tuo mukanaan muistin käytön lisäämistä, sillä uuden muuttujan luominen vie tilaa ja suorituskykyä, kun taas edellisen muokkaaminen varaa tilaa muistista luomishetkellä vaan kerran ja muuttaminen on vain yksi askel suorituksessa. Funktionaalisen ohjelmoinnin jatkuva kopioiminen varaa aina uutta tilaa jokaisella muutoshetkellä.

Jonas, Alvin ja Richard ovat kaikki samaa mieltä siitä, että funktionaalisen ohjelmoinnin osaajia on maailmassa vielä suhteellisen vähän. Tämä selittyy ainakin osittain sillä, että muut paradigmat ovat olleet vallassa jo pitkään, jolloin ohjelmoinnin opetuskin on toteutettu olio-ohjelmointia käyttäen. Täten funktionaalista ohjelmointia toteutetaan vielä rajallisesti, sen oppiminen voi osoittautua haasteeksi ja osaajia on vaikeaa löytää.

3 Tavoite ja tehtävä

Työn ensisijainen tavoite on selvittää, kuinka kahden vertailtavan paradigman ympärille kehittynyt keskustelu ja väittely hyödyistä sekä haitoista pitää paikkaansa käytännössä. Väittämät asetetaan vertailuun paradigmojen käsitteitä

rinnastaen ja jokaisen väittämän sekä käsitteen kohdalla toteutetaan ohjelmistollinen testaus, jossa arkinen ohjelmointiin liittyvä ongelma ratkaistaan niin olio-ohjelmointia kuin funktionaalista paradigmaa toteuttaen. Vertailulla selvitetään, kuinka funktionaalinen ohjelmointi pärjää tavanomaisissa tai opiskelun kannalta tyypillisissä ongelmanratkaisutilanteissa.

Tavoitteena on myös saavuttaa käsitys siitä, kuinka luettavaa funktionaalinen ohjelmointi on. Tällä haetaan vastausta siihen, onko funktionaalista ohjelmointia kannattavaa ottaa käyttöön esimerkiksi ohjelmointikursseilla tai opetuksessa ensisijaisena paradigmana vai tulisiko olio-ohjelmoinnin pysyä opiskelun kannalta ensikosketuksena alaan.

Funktionaalisen ohjelmoinnin väitettyjä hyötyjä, joita testataan suhteessa olio-ohjelmointiin, ovat ohjelmiston suorituksen ennustettavuus ja koodin yksinkertaisempi luettavuus. Näitä arvotetaan koodin luettavuuteen kohdennetuilla mittareilla. Funktionaalisen ohjelmoinnin väitetyt puutteet ovat funktioiden yhdistelyn tuottama epäselvyys, muuttumattomuuden tuottama kopiointi ja tilattomuuden toteuttamisen haasteellisuus käyttäjäsyötteitä vaativassa ohjelmistossa.

3.1 Innovaation mahdollistaminen

3.1.1 Paradigma laatikon ulkopuolella

Lopullisen pohdinnan yhteydessä tavoitellaan vastausta kysymykseen siitä, mitä ohjelmointiparadigmaa olisi parasta lähteä edistämään tai yleensä toteuttaa ohjelmistokehityksessä. Onko olio-ohjelmoinnin paikka ensisijaisena paradigmana oikeutettu ja ymmärrettävä, tulisiko funktionaalisen ohjelmoinnin jalansijaa nostattaa vai onko aiheellista sekä ajankohtaista nostaa esiin kokonaan uusi paradigma näiden rajoitteiden ulkopuolella tai jopa niitä yhdistellen?

3.1.2 Hyöty opiskelussa

Olio-ohjelmointi on edelleen ensisijainen paradigma ohjelmointiin liittyvässä opetuksessa. Työn hyöty muille opiskelijoille on tarjota riittävän kattava tarkastelu funktionaaliseen ohjelmointiin ja sen mahdollisiin hyötyihin, jotta aihetta opiskelevan on helppo vetää omat johtopäätöksensä siitä, millä tavalla ohjelmointia olisi hyvä tai mahdollisesti voisi toteuttaa tulevaisuudessa. Tällä mahdollistetaan niin innovaatiota kuin myös luovuutta alalla.

4 Menetelmät

4.1 Ohjelmointitehtävät

Opinnäytetyön tekijä mittaa mielipide- ja näkemyserojen paikkansapitävyyttä ohjelmistotehtävien kautta. Ohjelmallisesti toteutettuun testaukseen asetetaan arkinen ohjelmointiin liittyvä ongelma tai opetuksessa esiintyvä ohjelmointitehtävä, joka toteutetaan niin olio-ohjelmoinnin kautta, kuin myös funktionaalista ohjelmointia toteuttaen. Ohjelmointikielenä toimii javascript, jotta ohjelmointitehtävien sekä ratkaisujen seuraaminen olisi mahdollisimman seurattavaa.

4.2 Vertailtavat käsitteet

Kahden paradigman sisältämien käsitteiden vertailua toteutetaan asettamalla rinnakkain sellaiset käsitteet, jotka ovat tyypillisesti vastaavissa asemissa ohjelmointia suorittaessa. Funktionaalisen ohjelmoinnin rekursiivinen lähestymistapa vertautuu perinteiseen iteraation, funktioiden kompositio korvaa oliopohjaisen periytyvyyden ja olio-ohjelmoinnin luokkarakennetta sekä niiden sisältämiä metodeja on toteutettu funktionaalisessa ohjelmoinnissa yksittäisten funktioiden uudelleenkäytöllä. Näiden lisäksi muuttumattomuus, puhtaat funktiot ja tilattomuus ovat vertailun alla olio-ohjelmoinnin tapaan sallia muuttuvuutta, sivuvaikutuksia sekä ohjelman tilaa.

Kaikkia vertailtavia käsitteitä sekä konsepteja, jotka paradigmoista löytyvät ei voida määritellä vain niille kuuluviksi ja käsitteitä kuten rekursio on mahdollista toteuttaa molempien paradigmojen rajoissa, siksi vertailuun asettuukin niin paradigmojen omat käsitteet, kuin myös niissä tyypillisesti esiintyvät, mutta kuitenkin toisilleen sallitut konseptit.

4.2.1 Oliot/luokat ja funktiot

Olio-ohjelmoinnin malli toteuttaa olioita sekä luokkia edellyttää uuden olion luomista sen sisältämien metodien käyttöä varten. Funktionaalinen ohjelmointi jättää oliomaisen rakenteen kokonaan pois ja suosii yksittäisten funktioiden käyttöä luokkien sisältämien metodien sijasta.

Ohjelmointitehtäväksi valitaan yksinkertainen prosenttilaskuri, jossa olion tai funktion syötteeksi annetaan kaksi lukua, osoittaja ja nimittäjä, joiden perusteella lasketaan, kuinka monta prosenttia osoittaja on nimittäjästä.

4.2.2 Iteraatio ja rekursio

Vaikka rekursiota ja sen käyttöä ei ole rajoitettu vain funktionaalisen paradigman rajoihin, on se kuitenkin olennainen osa funktionaalista ohjelmointia, jossa funktioiden uudelleenkäytettävyys on keskiössä. Perinteinen iteratiivinen silmukka on taas tyypillisesti määriteltävä uudelleen sitä käytettäessä.

Ohjelmointitehtävänä toimii tietyn nimen ja sen esiintymisen määrän hakeminen annetusta nimilistasta.

4.2.3 Periytyvyys ja funktioiden kompositio

Olio-ohjelmoinnissa toiminnallisuutta ikään kuin ketjutetaan periytyvyyden kautta, jossa olio perii metodin toiminnallisuuden toiselta. Funktioiden kompositio toteuttaa vastaavan tavan mallintaa uudelleenkäytettävyyttä, jossa funktiot ottavat toisiaan parametreina ja toteuttavat sisällään toistensa toiminnallisuutta.

Ohjelmointitehtävänä toimii matemaattinen toiminnallisuus, jossa numeron kertominen potenssiin neljä perii numeron kuution laskemisen ja desimaaliluvun pyöristämisen toiminnallisuutta.

4.2.4 Muuttujat ja muuttumattomuus

Työtä ja testausta varten funktionaalisen paradigman käyttöä on rajattu puhtaasti funktionaaliseen, jotta muuttumattomuutta ja puhtaiden funktioiden toteutusta muuttujien arvoa kopioimalla voidaan testata sekä verrata perinteisten muuttujien toteuttamiseen, jossa arvoa muutetaan suoraan. Ohjelmointitehtäväksi valitaan listan alkioiden järjestyksen kääntäminen.

4.2.5 Tila ja tilattomuus

Tilatonta ohjelmaa verrataan ohjelmaan, joka muuttaa tilaansa. Tilaton ohjelma ei ole suorituksensa aikana riippuvainen edellisestä suorituskerrasta, vaan tuottaa saman lopputuloksen huolimatta siitä, miten usein ohjelmaa on uudelleen suoritettu. Tilaton ohjelma voi kuitenkin tallentaa tietoa muistiin ja hakea tietoa seuraavaa suoritusta varten. Tilallinen ohjelma taas ottaa huomioon myös viimeisen suorituskerran ja lopputulos on riippuvainen edellisten suorituskertojen tapahtumista. Tilallisen ohjelman on mahdollista pitää tietoa sisällään nojaamatta ulkopuolisen muistintallennuksen apuun. Ohjelmointitehtävänä toimii käyttäjän kirjautumistietojen tallentaminen ja vahvistaminen.

4.3 Vertailutavat ja mittaus

Vertailtavien käsitteiden painoarvoa mitataan ohjelmointitehtävien aikana ilmenneiden havaintojen kautta, heijastellen niitä paradigmoista keskusteleviin artikkeleihin, näkemyksiin sekä kannanottoihin. Mittareina toimivat ohjelmointitehtävän ratkaisuun käytetty aika ja koodin kognitiivinen luettavuus/ymmärrettävyys. Vaikka jotkin mitattavista seikoista ovat subjektiivisia, kuten luettavuus, niiden mittausmenetelmää selkeytetään.

4.3.1 Käytetty aika

Niin oliopohjaisen ratkaisun kuin funktionaalisen toteuttamista ajastetaan, jotta voidaan tulkita kuinka kauan funktionaalisen ohjelmoinnin toteuttaminen kestää käytännössä. Tällä tavoitellaan vastausta siihen, onko funktionaalisen ohjelmoinnin käyttöönotto ajallisesti ohjelmistokehityksessä varteenotettavampi vaihtoehto.

4.3.2 Luettavuus

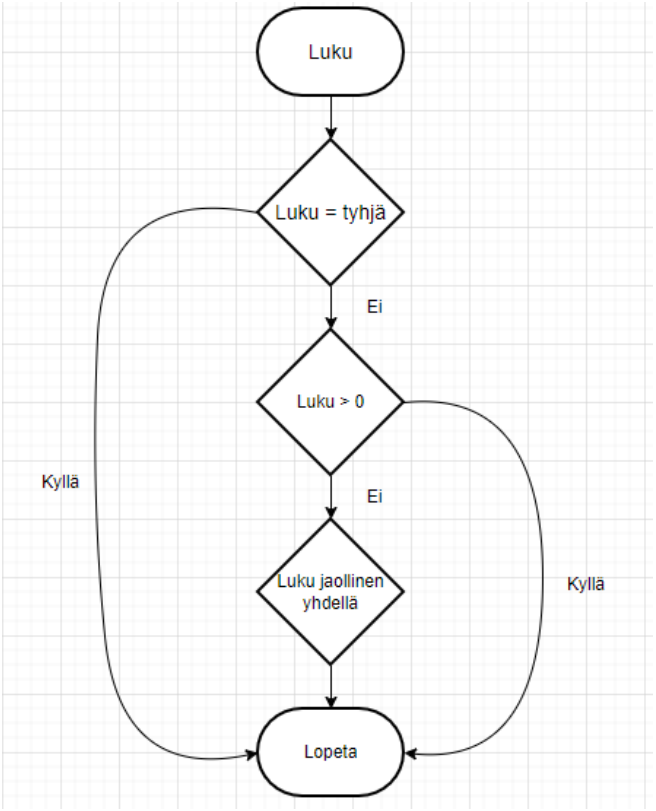
Tuotetun koodin luettavuus on subjektiivinen mittari, jolle kuitenkin voidaan antaa numeerisia arvoja.

Luettavuutta mitataan kognitiivisen kompleksisuuden kautta (Tiobe 2024). Kognitiivisen kompleksisuuden tulos saadaan laskemalla yhteen funktion, funktioiden tai ohjelmiston osan ehtolauseiden ja loogisten operaatioiden määrän. Luettavuuden testaamisen tavoitteena on selvittää, kuinka jatkokehitettävää koodi on.

```
function lukuOnPositiivinenKokonaisluku(luku) {
  if (luku === null) {
    return false
  } else if (luku < 0) {
    return false
  } else return luku % 1 === 0;
}
```

Esimerkkifunktio ”lukuOnPositiivinenKokonaisluku” sisältää kaksi ehtolausetta ja yhden oletusarvoisen lauseen. Ehtolauseilla on kaksi mahdollista haaraa, kun taas oletusarvoinen ”else” vie ohjelman aina samaan lopputulokseen. Funktion kognitiivisen kompleksisuuden arvo ja ehtolauseiden määrä on siis 2.

Toinen tapa mitata koodin luettavuutta on ”Cyclomatic complexity” (Asif 2024), jossa koodin luettavuutta ja ylläpidettävyyttä lasketaan koodissa olevien solmujen, niiden yhteyksien ja ehtolauseiden laskutulos. Laskukaava on CC = E – N + 2P, missä E on kaarien määrä, N on solujen määrä ja N on yhdistetyt komponentit.



Esimerkkifunktio ”lukuOnPositiivinenKokonaisluku” on vain yksittäinen komponentti, se sisältää 5 solmua, joiden välillä on 6 kaarta. Laskutoimitus 6 – 5 + 2(1) = 3. Syklomaattisen kompleksisuuden mukaan koodi on vielä yksinkertaisesti luettavan rajoissa.

5 Luotettavuus ja eettisyys

5.1 Subjektiiviset mittarit

Koska osa ohjelmistokehityksen haasteista pohjautuu subjektiivisille näkemyksille, on hyvin hankala sanoa, kuinka luotettavan lopputuloksen niiden mittaaminen todellisuudessa tuottaa. Subjektiivisille mittareille on kuitenkin yritetty antaa numeerisia arvoja, joita voidaan tuloksissa vertailla. Mielipiteitä tarkastellessa on mahdollista muodostaa kuva siitä, että jokin näkemys on kuin objektiivinen totuus asian laidasta. Opinnäytetyön tarkoitus ei ole osoittaa funktionaalisen ohjelmoinnin tai minkään muunkaan paradigman paremmuutta missään mielessä, vaan avartaa lukijan käsitystä siitä, pitävätkö väitetyt hyödyt paikkaansa, missä määrin ja miten paradigmojen hyötyjä, niiden käyttöä ja toteutusta voidaan avartaa tulevaisuudessa.

6 Ohjelmistotestaus

Ohjelmistollinen testaus on jaettu testattaviin osa-alueisiin, joiden kohdalla toteutus sekä testaus oliopohjaisella tavalla esitetään ensin ja funktionaalinen versio sen jälkeen.

6.1 Oliot/luokat ja funktiot

Olio-ohjelmoinnin olioita/luokkia ja niiden sisältämiä metodeja rinnastetaan funktionaaliseen ratkaisuun, jossa toiminnallisuuden edellytyksenä on pelkkä uudelleenkäytettävä funktio. Ohjelmistotehtävänä toimii prosenttilaskurin toteuttaminen

6.1.1 Oliopohjainen ratkaisu

```
class PercentageClass {
    getPercentage(nominator, denominator) {
        return (nominator / denominator) * 100
    }
}
```

Määritellään olio nimeltä ”PercentageClass”, joka sisältää metodin ”getPercentage”. Metodin parametreina ovat ”nominator” (osoittaja) ja ”denominator” (nimittäjä). Palautusarvoksi saadaan, kuinka monta prosenttia osoittaja on nimittäjästä.

```
const percentageObject = new PercentageClass()
const objectPercentageResult = percentageObject.getPercent-
age(50, 250)

console.log(objectPercentageResult)
```

Oliotestaus alkaa prosenttilaskuriolion luomisella ja asettamisella muuttujaan ”percentageObject”. Uuteen muuttujaan ”objectPercentageResult” tallennetaan prosenttiolion sisältämän metodin kutsusta palautunut tulos. Metodille annetaan argumentteina luvut 50 ja 250, joista saadaan tulos 20%.

6.1.2 Funktionaalinen ratkaisu

```
const percentageFunction = (nominator, denominator) =>
    (nominator / denominator) * 100
```

Funktionaalinen ratkaisu sisältää pelkän funktion, joka ottaa sisäänsä parametrit ”nominator” (osoittaja) ja ”denominator” (nimittäjä). Palautusarvona saadaan tulos siitä, kuinka monta prosenttiyksikköä osoittaja on nimittäjästä.

```
const functionalPercentageResult = percentageFunction(25, 200)

console.log(functionalPercentageResult)
```

Funktionaalisen testauksen tulos otetaan suoraan ylös muuttujaan, jonka arvoksi tulee prosenttilaskurin kutsun palautusarvo. Argumenteiksi annetaan luvut 25 ja 200. Tuloksena laskutoimitukselle on 12.5%.

6.2 Iteraatio ja rekursio

Iteratiivista lähestymistapaa verrataan rekursiiviseen funktioon. Iteraation ja rekursion rinnastamista tavoitellaan tietorakenteiden läpikäynnin kautta. Ohjelmointitehtäväksi on valittu annetun nimen, sekä sen ilmentymien määrän hakeminen nimilistasta.

6.2.1 Oliopohjainen ratkaisu

```
class SearchClass {
    searchNames(names, nameToSearch) {
        let occurrences = 0
        for (let i = 0; i < names.length; i++) {
            if (names[i] === nameToSearch) {
                occurrences++
            }
        }
        return occurrences
    }
}
```

Määritellään hakuolio ”SearchClass”, jonka sisällä on metodi ”searchNames”. Metodi ottaa vastaan parametreina nimilistan ja nimen, jota nimilistasta yritetään hakea. Metodin alussa alustetaan muuttuja ”occurrences” (ilmentymät) jonka arvo on lähtökohtaisesti 0. For-silmukassa käydään nimilista alkio kerrallaan läpi ja jos ehtolause, jossa kyseinen alkio on yhtä kuin haettu nimi toteutuu, lisätään ilmentymien arvoa yhdellä, muussa tapauksessa siirrytään seuraavaan alkioon. Lopullinen palautusarvo on nimen ilmentymien määrä.

```
const namesList = [
    "Michael",
    "Jonathan",
    "Laura",
    "Charlie",
    "Maria",
    "Laura",
    "Laura"
]

const searchObject = new SearchClass()
const iterativeResult = searchObject.searchNames(namesList,
"Laura")

console.log(iterativeResult)
```

Testausta varten luodaan nimilista, joka alustetaan niin, että nimi ”Laura” esiintyy kolmesti. Luodaan hakuolio ja otetaan metodin kutsun tulos ylös muuttujaan. Metodin argumenteiksi annetaan luotu nimilista ja haettava nimi ”Laura”. Lopuksi tulostuu tieto siitä, että nimi ”Laura” on esiintynyt listassa kolmesti.

6.2.2 Funktionaalinen ratkaisu

```
const recursiveSearch = (names, nameToSearch, index = 0) => {
    if (index > names.length)
        return 0
    else if (names[index] === nameToSearch)
            return 1 + recursiveSearch(names, nameToSearch, in-
    dex + 1)
    else
        return recursiveSearch(names, nameToSearch, index + 1)
}
```

Rekursiivinen lähestymistapa alkaa useammalla parametrilla. On nimilista, haettu nimi, sekä sen hetkinen listan indeksi. Koska puhtaasti funktionaalisessa ratkaisussa muuttujien arvoa ei voida muokata, täytyy tieto sen hetkisestä alkion indeksistä antaa rekursiokutsun parametreina seuraavalle kutsulle. Funktio

sisältää kolme ehtolausetta. Funktion lopetusehto on tilanne, jossa annettu indeksi on suurempi kuin nimilistan pituus. Tässä tapauksessa palautusarvona on nolla. Mikäli haettu nimi täsmää sen hetkisen alkion arvoa, palautusarvona on numero yksi, sekä rekursiivinen funktiokutsu, jonka parametrina ovat nimilista, haettava nimi sekä kokonaisluku, jonka arvo on indeksi plus yksi. Jos muut ehdot eivät toteudu, palautusarvona on funktiokutsu parametreilla nimilista, haettava nimi ja luku indeksi plus yksi.

```
const listOfNames = [
    "Steven",
    "Max",
    "Susan",
    "Anne",
    "Christopher",
    "Max"
]

const recursiveResult = searchNamesRecursively(listOfNames,
"Max")

console.log(recursiveResult)
```

Funktionaalisessa ratkaisussa luodaan nimilista, jossa nimi ”Max” esiintyy kahdesti. Funktiokutsun tulos tallennetaan muuttujaan, jossa kutsulle annetaan nimilista ja haettava nimi ”Max”. Indeksin arvoa tai ilmentymien määrää ei tarvitse kutsun aikana kertoa, sillä niille on annettu oletusarvot funktiossa itsessään. Lopuksi tulostetaan tieto, että nimi ”Max” esiintyy listassa kahdesti.

6.3 Periytyvyys ja funktioiden kompositio

Olioiden tapaa periä toistensa toiminnallisuutta ja dataa verrataan funktioiden kompositioon, jossa funktion palautusarvoa käytetään toisen funktion argumenttina. Ohjelmointitehtävänä on toteuttaa matemaattinen lopputulos, jossa luvun potenssiin neljä laskemisen toiminnallisuutta jaetaan luvun kuution laskemisen, sekä luvun desimaalien pyöristämisen kanssa

6.3.1 Oliopohjainen ratkaisu

```
class Decimal {
    roundDown(number) {
        return number - number % 1
    }
}
```

Luodaan desimaaliluokka, joka sisältää metodin ”roundDown” (pyöristä alaspäin). Palautusarvona on kokonaisluku ilman desimaaleja.

```
class Cube extends Decimal {
    cube(number) {
        return number * number * number
    }
}
```

Luokka nimeltä ”Cube” (kuutio), jonka metodi antaa palautusarvona luvun potenssiin kolme. Luokka perii myös desimaaliluokan toiminnallisuuden pyöristää lukuja.

```
class FourthPower extends Cube {
    fourthPowerRounded(number) {
        const cube = super.cube(number)
        return super.roundDown(cube * number)
    }
}
```

Luokka nimeltä ”FourthPower” (potenssiin neljä), luokka perii kuution toiminnallisuuden ja sen mukana myös desimaaliluokan. Luokan metodi nimeltä ”fourthPowerRounded” (potenssiin neljä pyöristettynä), tallentaa numeron kuution tuloksen muuttujaan ja palautusarvona on desimaaliluvun pyöristämisfunktion kutsun tulos, jolle annetaan luvun kuutio kerrottuna luvulla.

```
const decimalValue = 2.5

const fourthPowerObject = new FourthPower()
const fourthPowerObjectResult = fourthPowerObject.fourthPower-
Rounded(decimalValue)

console.log(fourthPowerObjectResult)
```

Testaus alkaa desimaaliluvun alustamisella luvulla 2,5. Luodaan uusi olio luokasta potenssiin neljä, otetaan talteen metodin kutsun tulos, jonka argumentiksi annetaan luotu desimaaliluku. Lopuksi tulostuu luvun 2,5 potenssiin neljä pyöristettynä alimpaan kokonaislukuun, eli 39.

6.3.2 Funktionaalinen ratkaisu

```
const roundDown = num => num - num % 1

const cube = num => num * num * num

const fourthPowerRounded = num => roundDown(cube(num) * num)
```

Funktionaalisessa ratkaisussa määritellään kolme funktiota. Funktio ”roundDown” (pyöristä alaspäin), joka antaa luvun alemman kokonaisluvun. Funktio ”cube” (kuutio), joka palauttaa luvun potenssiin kolme. Funktio ”fourthPowerRounded” (potenssiin neljä pyöristettynä), jonka palautusarvona on kompositio funktioista ”roundDown” ja ”cube”. Pyöristämisfunktion argumentiksi annetaan kuutiofunktion kutsun tulos kerrottuna luvulla.

```
const decimalValue = 2.5

const fourthPowerFunctional = fourthPowerRounded(decimalValue)

console.log(fourthPowerFunctional)
```

Funktionaalinen testaus alkaa desimaaliluvun 2,5 luomisella. Funktiota ”fourthPowerRounded” kutsutaan antamalla sille luotu desimaaliluku ja arvoksi tulostuu 39.

6.4 Muuttujat ja muuttumattomuus.

Ohjelmointia muuttujilla ja niiden arvojen muokkaamisella verrataan ohjelmaan, jossa muuttujien arvoa ei muuteta niiden luomisen jälkeen. Ohjelmistotehtävänä luodaan listan järjestyksen kääntäminen.

6.4.1 Oliopohjainen ratkaisu

```
class Reverse {
    reverse(array) {
        for (let i = 0; i < array.length / 2; i++) {
            const replaceIndex = (array.length - 1) - i
            const temp = array[i]
            array[i] = array[replaceIndex]
            array[replaceIndex] = temp
        }
        return array
    }
}
```

Luodaan luokka “Reverse” (käänteinen), jonka metodi ottaa sisäänsä listan, jonka alkioiden järjestys käännetään.

Silmukan lopetusehtona on tilanne, jossa indeksi on suurempi kuin alkioiden määrä jaettuna kahdella. Tämä siksi, että jokaisella iteraatiolla sen hetkinen alkio vaihtaa paikkaa alkion kanssa, joka on yhtä kaukana listan keskipisteestä sen loppupäässä.

Silmukan alussa määritellään sen hetkisen alkion peilikuvan indeksi, joka on yhtä kaukana listan keskipisteestä, mutta listan loppupuolella. Luodaan väliaikainen alkio ”temp”, jonka arvoksi tulee sen hetkisen alkion arvo. Käsiteltävän alkion arvo muutetaan peilikuvan arvoksi ja peilikuvan arvo muutetaan

väliaikaismuuttujan arvoksi. Näin alkioiden paikka vaihdetaan yhden iteraation aikana. Lopuksi palautetaan muutettu lista.

```
const listOfLetters = ["a", "b", "c", "d"]

const reverseObject = new Reverse()
reverseObject.reverse(listOfLetters)

console.log(listOfLetters)
```

Testauksessa luodaan lista kirjaimia aakkosjärjestyksessä. Luodaan kääntämisolio, ja kutsutaan sen funktiota antaen argumentiksi lista kirjaimia. Lopuksi tulostetaan lista, joka on muuttunut käänteiseksi.

6.4.2 Funktionaalinen ratkaisu

```
const reverse =
(array, index = array.length - 1, tempArr = []) => {
    const replace = array[index]
    const newArr = [...tempArr, replace]
    if (index === 0) {
        return newArr
    }
    return reverse(array, index - 1, newArr)
}
```

Funktionaalinen toteutus käyttää rekursiota listan läpikäyntiin. Funktion parametreina ovat lista, indeksi, jonka oletusarvo on listan viimeisen alkion indeksi ja väliaikainen lista. Funktion alussa muuttujaan ”replace” (korvaaja) tallennetaan indeksin kohdalla oleva alkio listasta, ensimmäisellä kutsulla listan viimeinen alkio. Toiseen muuttujaan tallennetaan väliaikaisen listan alkiot ja sen viimeiseksi alkioksi korvaaja alkio. Lopetusehtona on tilanne, jossa annettu indeksi on nolla, eli listan alku. Mikäli lopetusehto ei täyty, kutsutaan funktiota uudelleen alkuperäisellä listalla, indeksin arvoa vähentämällä ja uudella listalla.

```
const listOfNumbers = [1, 2, 3, 4, 5]

const functionalReverse = reverse(listOfNumbers)

console.log(functionalReverse)
```

Funktionaalinen testaus aloitetaan listalla numeroita yhdestä viiteen. Kääntämisfunktiota kutsutaan listalla ja tulostuu lista numeroita käänteisessä järjestyksessä.

6.5 Tila ja tilattomuus

Tilatonta ohjelmaa, jossa ohjelman sen hetkinen tieto ei tallennu ohjelmaan, vaan sen ulkoiseen tahoon, verrataan sellaiseen ohjelmaan, joka on täysin vastuussa tilastaan. Toteutetaan käyttäjän kirjautumiseen liittyvä toiminnallisuus.

6.5.1 Oliopohjainen ratkaisu

```
class UserInformation {
    constructor(userName, password) {
        this.userName = userName
        this.password = password
        this.loggedIn = false
        this.attemptedSignIns = 0
    }

    /*...*/
}
```

Luodaan käyttäjätietoluokka ”UserInformation”, jonka konstruktorina on tiedot käyttäjänimestä, salasana, onko käyttäjä kirjatunut sisään ja kuinka monta kertaa käyttäjätunnuksilla on yritetty kirjautua sisään epäonnistuneesti.

```
login(userName, password) {
    if (userName === this.userName && password === this.pass-
    word) {
        this.loggedIn = true
    } else if (userName === this.userName) {
        this.attemptedSignIns += 1
    }
}
```

Käyttäjätietoluokan metodi “login”, joka vastaa sisäänkirjautumisesta. Jos kirjautumisen yhteydessä annetut käyttäjänimi ja salasana täsmäävät olion sisälle tallennettuihin tietoihin, annetaan kirjautumistilanteeksi arvo ”tosi”. Jos vain nimi täsmää, nostetaan epäonnistuneiden kirjautumisten arvoa yhdellä.

```
logOut() {
    this.loggedIn = false
}
```

Metodi ”logout”, joka kirjaa käyttäjän ulos.

```
get isLoggedIn() {
    return this.loggedIn
}
```

Metodi, jolla saadaan tieto siitä, onko käyttäjä kirjautunut sisään vai ei.

```
get status() {
    return {
        userName: this.userName,
        isLoggedIn: this.isLoggedIn,
        attemptedSignIns: this.attemptedSignIns
    }
}
```

Lopuksi metodi “status”, joka palauttaa objektin käyttäjän tilanteesta.

```
const userInfo = new UserInformation("Tuomas", "salaSana123")

userInfo.login("Tuomas", "VääräSalaSana")
userInfo.login("Tuomas", "salaSana123")
userInfo.logOut()

const signedStatus = userInfo.status

console.log(
    `The user ${signedStatus.userName} is
    ${signedStatus.loggedIn === "true" ? "logged in" : "signed
out"}
    and has tried to sign in ${signedStatus.attemptedSignIns}
    times unsuccessfully`
)
```

Testauksessa luodaan käyttäjätieto olio tiedoilla ”Tuomas” ja ”salaSana123”. Yritetään kirjautua sisään väärällä salasanalla kerran, jonka jälkeen kirjaudutaan kerran onnistuneesti ja lopuksi ulos. Tulostetaan viesti, jossa kerrotaan, että käyttäjätilille ”Tuomas” ei ole kirjauduttu sisään ja epäonnistuneita sisäänkirjautumisia on yksi.

6.5.2 Funktionaalinen ratkaisu

```
const createAccount = (userName, password) => {
    localStorage.setItem("userName", userName)
    localStorage.setItem("password", password)
    localStorage.setItem("attemptedSignIns", "0")
}
```

Funktio “createAccount”, joka luo annetun käyttäjätunnuksen ja salasanan. Myös tieto virheellisistä kirjautumisista alustetaan luvulla nolla.

```
const getInfo = () => {
    return {
        userName: localStorage.getItem("userName"),
        password: localStorage.getItem("password"),
        loggedIn: localStorage.getItem("loggedIn"),
        attemptedSignIns: parseInt(
            localStorage.getItem("attemptedSignIns")
        )
    }
}
```

Funktio “getInfo”, jonka tehtävänä on palauttaa kaikki käyttäjätiedot ja niiden tila.

```
const login = (userName, password) => {
    const correctInfo = getInfo()
    if (
        userName === correctInfo.userName &&
        password === correctInfo.password
    ) {
        localStorage.setItem("loggedIn", "true")
    } else if (userName === correctInfo.userName)  {
        localStorage.setItem(
                "attemptedSignIns",
               `${correctInfo.attemptedSignIns + 1}`
        )
    }}
```

Sisäänkirjautumisfunktio “login”, jossa tarkastetaan annettujen käyttäjätietojen oikeellisuus. Mikäli annettu salasana ei täsmää, nostetaan virheellisten kirjautumisten arvoa yhdellä.

```
const logout = () => {
    localStorage.setItem("loggedIn", "false")
}
```

Funktio käyttäjän uloskirjaamista varten.

```
const getStatus = () => {
    const userInfo = getInfo()

    return {
        userName: userInfo.userName,
        loggedIn: userInfo.loggedIn,
        attemptedSignIns: userInfo.attemptedSignIns
    }
}
```

Funktio käyttäjän tilasta kertovan viestin palauttamiseksi.

```
createAccount("Joonas", "12345")

login("Joonas", "54321")
login("Joonas", "abcdefg")
login("Joonas", "12345")
logout()

const userStatus = getStatus()

console.log(
    `The user ${userStatus.userName} is
    ${userStatus.loggedIn === "true" ? "logged in" : "signed
    out"}
    and has tried to sign in ${userStatus.attemptedSignIns}
    times unsuccessfully`
)
```

Testaus alkaa uuden käyttäjän “Joonas” luomisella salasanalla ”12345”. Väärällä salasanalla kirjaudutaan sisään kahdesti, oikealla kerran ja lopuksi ulos.

Tulostetaan tieto siitä, että käyttäjälle ”Joonas” ei ole kirjauduttu sisään ja epäonnistuneiden kirjautumisten määrä on kaksi.

7 Tulokset

Tulokset on jaettu jokaisen aihealueen kohdalla kolmeen kategoriaan. Käytetty aika, kognitiivinen kompleksisuus ja syklomaattinen kompleksisuus.

7.1 Oliot/Luokat ja funktiot

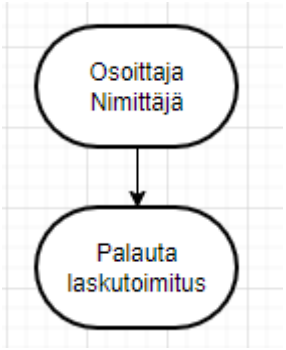
7.1.1 Aika

Ajallisesti toteutukset olivat hyvinkin vastaavat. Prosenttiolion metodi ja prosenttifunktio ovat toiminnallisuudeltaan tismalleen samanlaiset, joten luokan nimen kirjoittamiseen käytetty aika on vain marginaalinen ero näiden välillä. Oliollinen toteutus vei aikaa 1 minuutti ja 3 sekuntia, kun taas funktionaalinen 1 minuutti ja 15 sekuntia.

7.1.2 Kognitiivinen kompleksisuus

Kumpikaan ratkaisu ei sisällä ehtolauseita, sisennyksiä tai silmukoita. Ne kuitenkin toteuttavat kaksi matemaattista operaatiota, nimittäjän ja osoittajan jakolaskun, sekä jakolaskun tuloksen kertomisen sadalla. Solmujen yhteenlaskun tuloksena kognitiivinen kompleksisuus on siis molemmissa 2.

7.1.3 Syklomaattinen kompleksisuus



Niin oliomainen kuin funktionaalinen sisältävät 2 solmua ja yhden kaaren niiden välillä. Laskutoimitus, jossa kaarien määrästä 1 vähennetään solmujen määrä kaksi ja johon lisätään 2 kertaa komponenttien määrä 1 antaa tuloksen 1 – 2 + 2(1) = 1.

7.2 Iteraatio ja rekursio

7.2.1 Aika

Iteratiivinen toteutus oli ajallisesti suoraviivaisempi. Listan läpikäyntiin toteutettu silmukka ja ehtolause veivät ajallisesti 2 minuuttia ja 11 sekuntia.

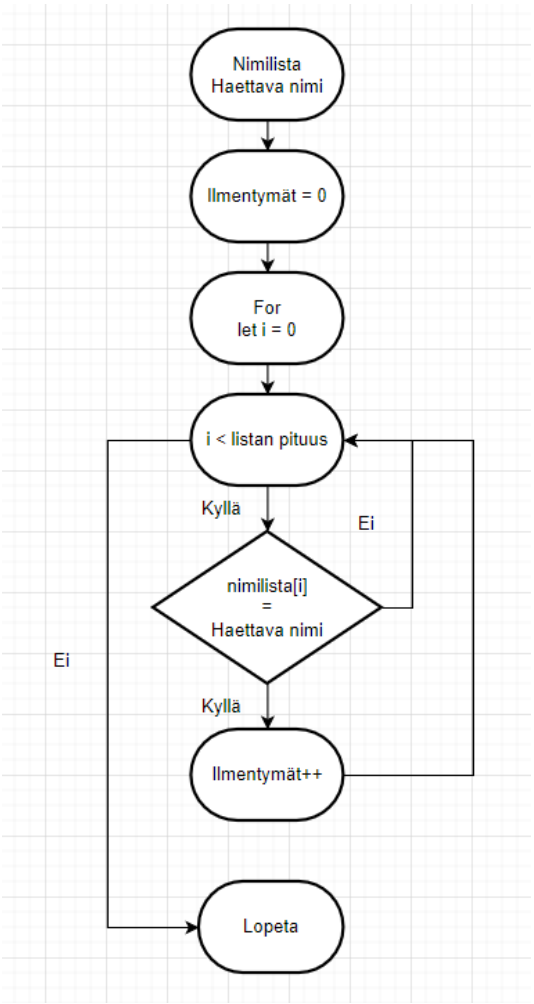
Rekursiivinen lähestymistapa on todettu olevan haasteellisempi hallita, sillä lopetusehdon täyttymisen toteutuminen täytyy pitää mielessä ja funktion uudelleen kutsumisessa on varauduttava loputtomaan silmukkaan. Tästä syystä rekursiivinen toteutus vei aikaa 7 minuuttia ja 6 sekuntia

7.2.2 Kognitiivinen kompleksisuus

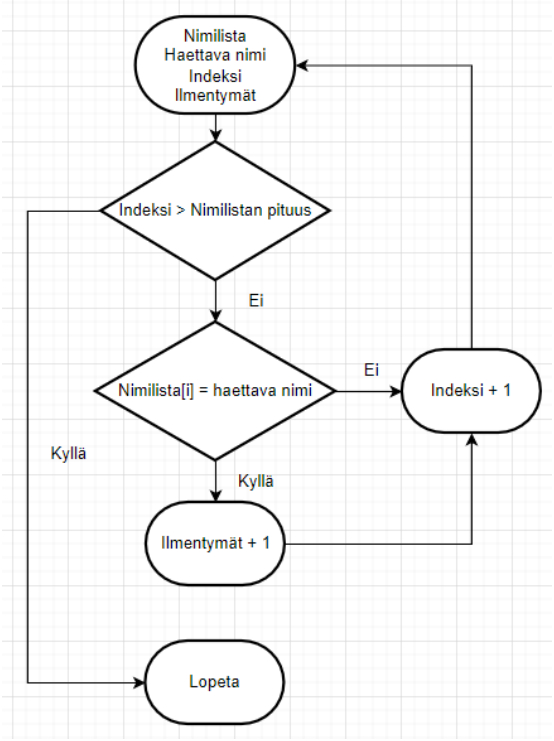
Iteraatiossa on yksi silmukka ja yksi ehtolause. Hakuluokan kognitiivinen kompleksisuus on 2.

Rekursiivisessa hakufunktiossa on kolme ehtolausetta, joista saadaan kognitiivisen kompleksisuuden arvo 3.

7.2.3 Syklomaattinen kompleksisuus



Oliomainen ratkaisu sisältää 7 solmua ja 8 kaarta niiden välillä. Syklomaattinen kompleksisuus on kaarien määrä 8 – solmujen määrä 7 + 2(1) = 3.



Funktionaalinen rekursio sisältää 6 solmua ja 7 kaarta. Syklomaattinen kompleksisuus on kaarien määrä 7 – solmujen määrä 6 + 2(1) = 3.

7.3 Periytyvyys ja funktioiden kompositio

7.3.1 Aika

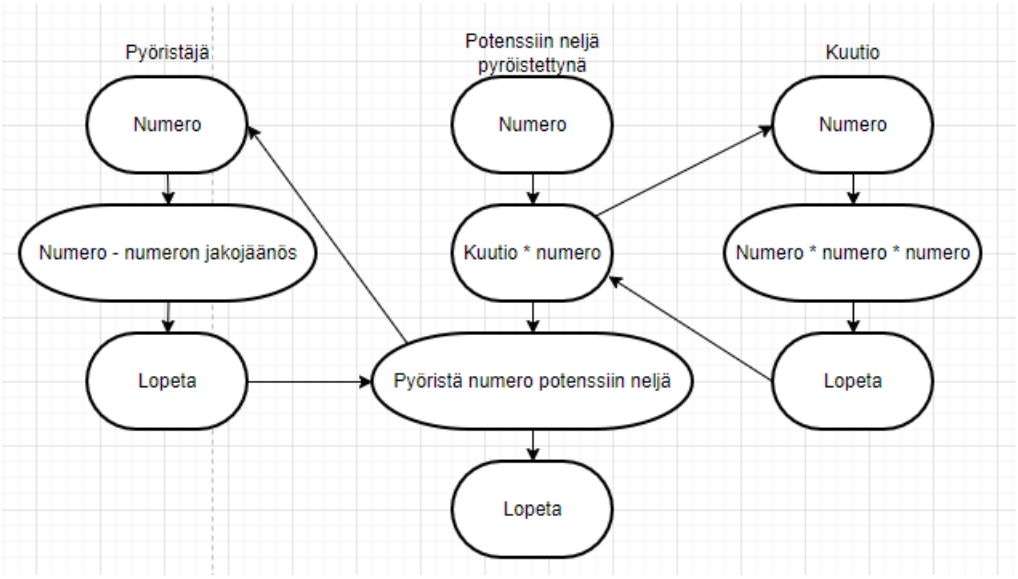
Olioiden kirjoittamine, niiden sisäiset metodit ja periytyvän toiminnallisuuden luonti on suoraviivainen prosessi, joka vei aikaa 3 minuuttia ja 13 sekuntia.

Funktionaalinen toteutus on toiminnallisuudeltaan täysin sama, mutta koska koodia kirjoitettiin lopulta kokonaisuudessaan vähemmän, aikaa käytettiin vain 2 minuuttia 42 sekuntia.

7.3.2 Kognitiivinen kompleksisuus

Kummassakaan matemaattisessa toteutuksessa ei esiinny sisennystä, ehtolauseita tai operaatioita. Sisennettyjä komponentteja on kuitenkin 3 kappaletta, joten kognitiivinen kompleksisuus molemmissa on 3.

7.3.3 Syklomaattinen kompleksisuus



Kaavio pätee niin olioratkaisun, kuin myös funktionaalisen kohdalla. Yhdisteltyjen komponenttien määrä on 3, joissa yhteenlaskettujen solmujen määrä on 10 ja kaarien 11. Syklomaattinen kompleksisuus on kaarien määrä 11 – solmujen määrä 10 + 2 * yhdistettyjen komponenttien määrä 3 = 7.

7.4 Muuttujat ja muuttumattomuus

7.4.1 Aika

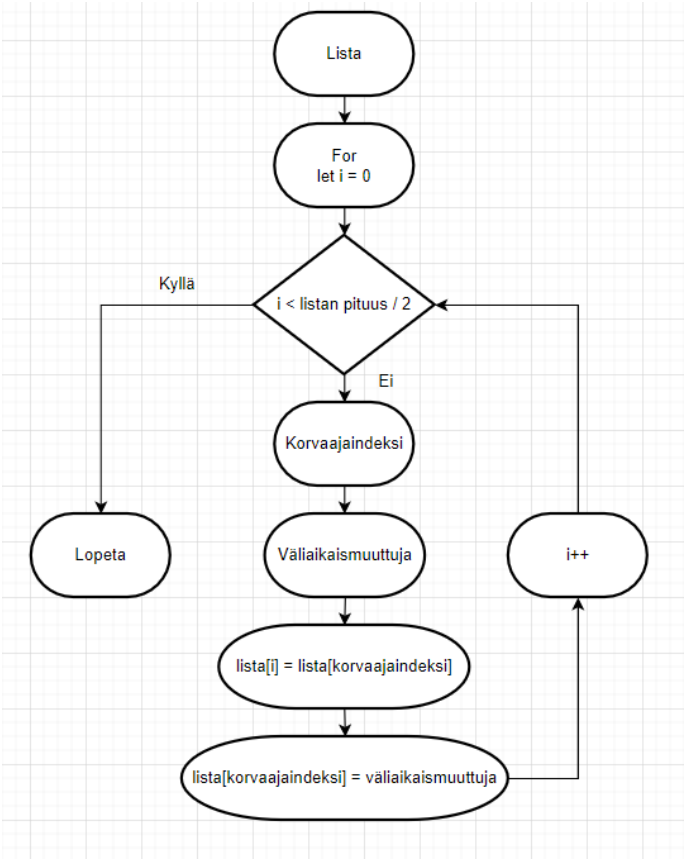
Iteraation ja rekursion verrattavat oppimiseen ja ymmärrettävyyteen liittyvät erot on huomioitu jo aiemmin. Näistä syistä kyseiset ratkaisut olivat ajallisesti myös merkittävästi erilaiset. Oliomainen toteutus kesti iteratiivisessa lähestymistavassaan 2 minuuttia ja 6 sekuntia, kun taas funktionaalinen rekursioratkaisu vei 5

minuuttia ja 2 sekuntia. Rekursiivinen lähestymistapa vaati useamman toteutuksen ja testauksen ennen lopulliseen päätymistä.

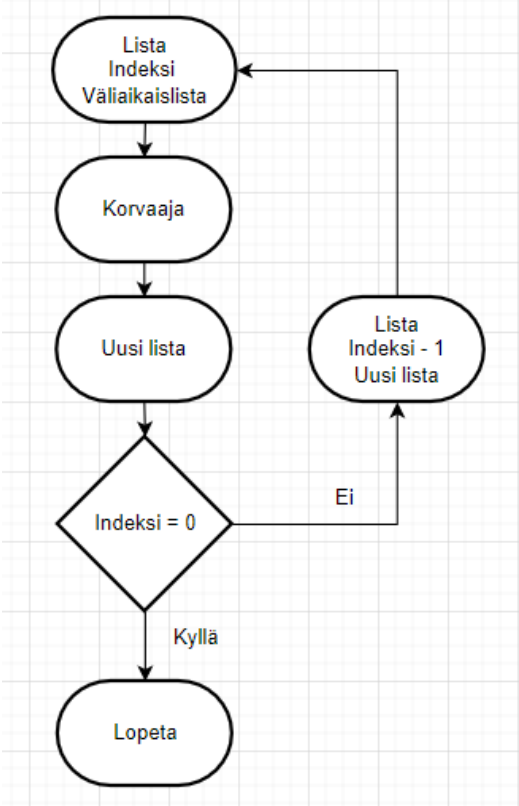
7.4.2 Kognitiivinen kompleksisuus

Listan kääntämiseen toteutettu iteraatio sisältää yhden silmukan ja rekursiivinen ratkaisu vain yhden ehtolauseen. Muuta sisennystä ei kummassakaan toteutuksessa ilmene, joten kummankin kompleksisuus on 1.

7.4.3 Syklomaattinen kompleksisuus



Olioratkaisussa on 9 solmua ja 9 kaarta, joten syklomaattinen kompleksisuus on 9 – 9 + 2(1) = 2.



Kääntämisfunktio sisältää 6 solmua ja 6 kaarta, joten 6 – 6 + 2(1) = 2.

7.5 Tila ja tilattomuus

7.5.1 Aika

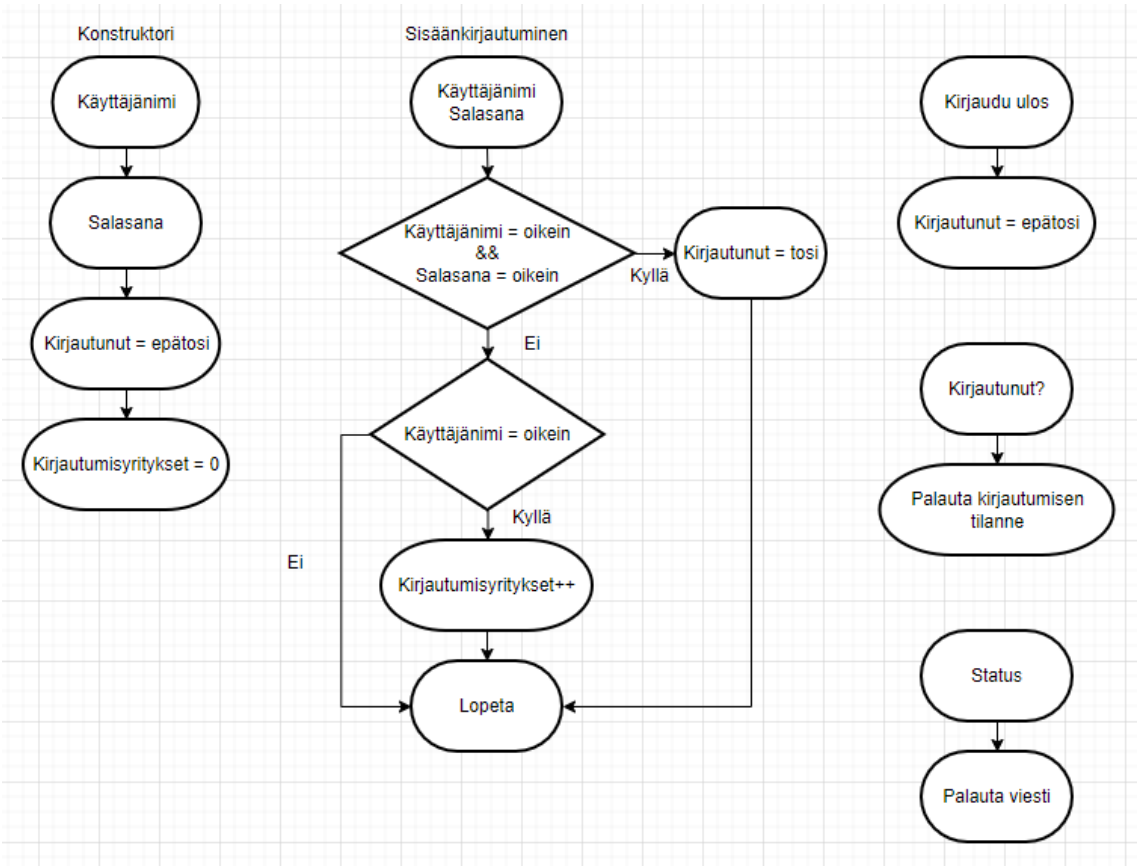
Vaikka kirjautumistietojen toiminnallisuus on melkein tismalleen sama oliopohjaisessa ratkaisussa kuin funktionaalisessa, oli funktionaalisessa varmistettava myös tiedon tallentamiseen liittyvän toiminnallisuuden oikeellisuus. Oliomainen ratkaisu vei aikaa 4 minuuttia ja 49 sekuntia, kun taas funktionaalinen 5 minuuttia ja 21 sekuntia.

7.5.2 Kognitiivinen kompleksisuus

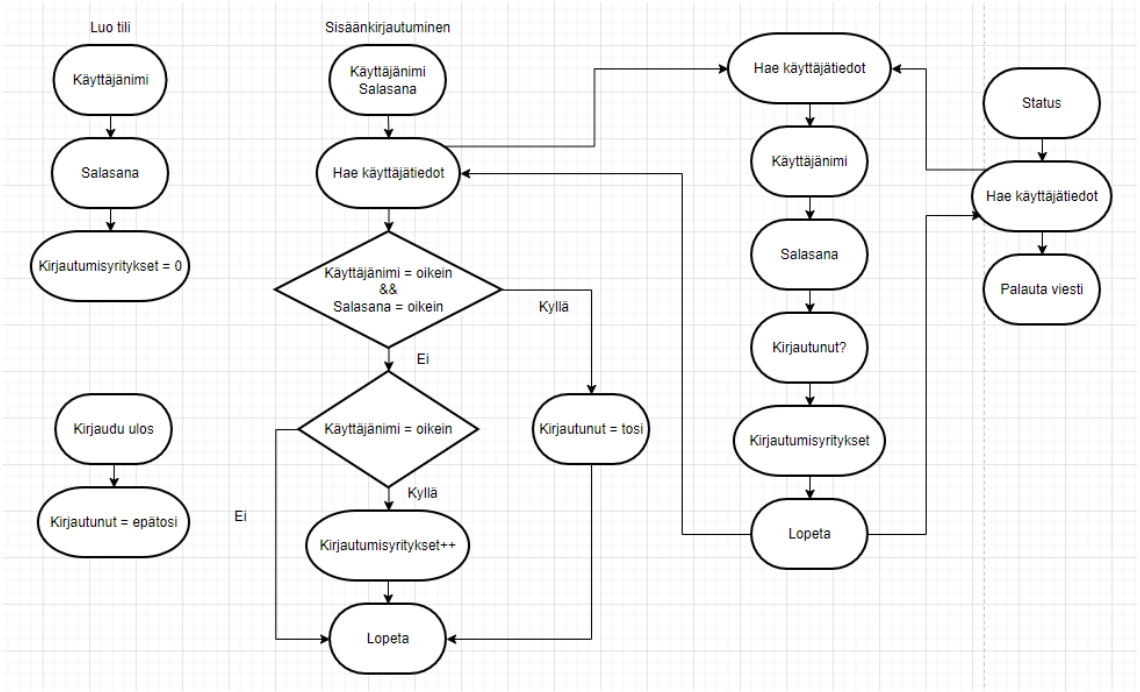
Kumpikin toteutus sisältää kaksi ehtolausetta, joista toisessa on yksi looginen operaatio AND. Molemmissa on 5 erillistä komponenttia, joten molempien

kompleksisuus on ehtolauseiden määrä 2 + loogisten operaattoreiden määrä 1 plus 5 komponenttia = 8.

7.5.3 Syklomaattinen kompleksisuus



Olion metodit ovat toisistaan erillisiä komponentteja. Sisäänkirjautumisen toiminnallisuus sisältää 6 solmua ja 7 kaarta, joten sen syklomaattinen kompleksisuus on 7 – 6 + 2(1) = 3.



Funktionaalinen toiminnallisuus sisäänkirjautumisessa sisältää kolme yhdistettyä komponenttia, joiden yhteenlaskettujen solmujen määrä on 16 ja kaarien 19, joten 19 – 16 + 2(3) = 9.

7.6 Taulukko

Tehtävä	Aika	Kognitiivinen kompleksisuus	Syklomaattinen kompleksisuus
Oliot/luokat Funktiot	1min 3 s 1min 15 s	2 2	1 1
Iteraatio Rekursio	2min 11 s 7min 6 s	2 3	3 3
Periytyvyys Funktioiden kompositio	3min 13 s 2min 42 s	3 3	7 7
Muuttujat Muuttumatto-muus	2min 6 s 5min 2 s	1 1	2 2
Tila Tilattomuus	4min 49 s 5min 21 s	8 8	3 9

8 Pohdinta

Niin artikkelien läpikäynti kuin ohjelmistollinen testaus on osoittanut, että kysymys paradigmojen paremmuudesta on kovin subjektiivinen kysymys. Luettavuus, soveltuvuus sekä muu ongelmallisuus ovat sellaisia tekijöitä, jotka ovat todella yksilökohtaisia riippuen yksittäisistä käsitteistä. Tästä syystä on hankala vetää yksiselitteistä johtopäätöstä jonkin paradigman kokonaisvaltaisesta paremmuudesta.

8.1 Väitteiden ja näkemysten pätevyys

8.1.1 Hyödyt

Funktionaalisen ohjelmoinnin väitetyt hyödyt paradigman selkeydestä ja ennustettavuudesta ovat parhaimmillaan vain osittain totta. Eriytetty toiminnallisuus omiksi funktioikseen tuo yksinkertaisuutta ja sen mukana luettavuutta, jossa jokainen toiminnallisuus on erillinen osansa sen sijaan, että luotaisiin kokonaisia luokkia sisältäen montaa toiminnallisuutta ja periytyvyyttä.

8.1.2 Haitat

Selkeyttä haittaavaksi tekijäksi osoittautui rekursio, joka vaatii ajattelua niin sanotusti ajassa eteenpäin, kun taas iteraation eri vaiheet ovat selkeästi luettavissa silmukassa itsessään. Rekursion kohdalla täytyy ottaa selvää, mitä potentiaalisia vaihtoehtoja seuraavalla funktiokutsulla on, eikä se ole aina luettavissa funktiosta itsessään.

Muuttumattomuuden ja puhtaiden funktioiden toteuttaminen yksittäisinä osina on todella helppoa, täytyy vain ottaa sisään syötetty arvo ja palauttaa sen perusteella jonkin kaavan lopputulos. Kuitenkin puhtaiden funktioiden ja niiden eri osien yhdistely osoittautuu haasteelliseksi siinä kohtaa, mitä enemmän niitä on.

Tilakoneet ja muut muuttuvat toiminnallisuudet melkein vaativat sen, että syötteiden arvot vaihtuvat ohjelmiston osien keskustellessa keskenään.

8.2 Testauksen havainnot

Ohjelmistotestauksen aikana suurin haaste oli keksiä sellaisia ohjelmointitehtäviä, jotka parhaiten toimivat verrattujen käsitteiden kohdalla. On selkeää, että funktionaalisen ohjelmoinnin vahvuus on matemaattisissa toiminnoissa, eikä sen keskeinen ajattelumalli ehkä ole suoraan verrannollinen olio-ohjelmoinnin yksittäisiin käsitteisiin. On hyvin haasteellista ajatella sellaista tehtävää, jonka ratkaisussa eri paradigmojen käsitteet ovat suoraan rinnasteisia toiminnallisuudeltaan tai soveltuvuudeltaan.

On huomioitavaa, kuinka puhtaasti funktionaalinen paradigma asettaa kehittäjän hyvin kapeisiin rajoihin. Muuttumattomuuden varmistaminen kehityksen aikana luo vaatimustason, josta poikkeaminen on jatkuvasti houkuttelevaa, mutta kiellettyä. Esimerkiksi tietorakenteen läpikäynnissä ja alkioiden järjestyksen muokkaamisessa olisi paljon kätevämpää vaihtaa muuttujien arvoa, kuin miettiä, kuinka yksinkertainen ohjelmiston osa voidaan toteuttaa ilman, että mitään arvoa muutetaan.

Tuloksissa ei ole mitään sellaisia merkittäviä eroja, jotka osoittaisivat kummankaan paradigman paremmuuden millään mitattavalla tavalla. Kuten mielipiteille ja näkemyksille on tyypillistä, niin paremmuus, sopeutuvuus kuin myös tarpeellisuus kunkin paradigman kohdalla enimmäkseen subjektiivinen näkemys, kuin mitään selkeästi vertailtavaa.

8.3 Katse tulevaisuuteen

Vaikka funktionaalinen ohjelmointi ja sen käsitteet ovat saaneet ajan saatossa enemmän tukea niin ohjelmointikieliltä, kuin erinäisiltä kirjastoilta ja paradigmaa kyetään toteuttamaan millä tahansa ohjelmointikielellä, on hyvin hankala nähdä

funktionaalista ohjelmointia minään muuna kuin osana suurempaa kokonaisuutta.

8.3.1 Ohjelmistokehitys

Funktionaalista, tai ainakin puhtaasti funktionaalista ohjelmointia on hyvin hankala toteuttaa sellaisissa ohjelmistoissa, joissa tila on olennaista. Mikä tahansa käyttöliittymä sisältää käyttäjäsyötteitä niin paljon, että tilaa on välttämätöntä ylläpitää. Puhtaasti funktionaalinen ohjelmointi olisikin varmasti vahvimmillaan osana suurempaa järjestelmää, esimerkiksi tilanteessa, jossa jokin ohjelma tarvitsee visualisoidun datan laskemista sekä jäsentelyä matemaattisesti vaikkapa graafeissa.

Muuttumattomuus osana ohjelmistoa, jossa erilaiset tietorakenteet ja niiden sisältö on alati muuttuvaa, on hyvin työläs prosessi. Puhdas funktionaalisuus on toimivaa silloin, kun kyse on hyvin yksittäisestä funktiosta, jossa syötteenä on arvoja, kuitenkin tietorakenteiden käsittelyyn muuttujien muokkaaminen on paljon kätevämpi ratkaisu.

8.3.2 Opiskelu

Funktionaalinen ohjelmointi on hyödyllinen osa opiskelua siinä kontekstissa, kun ohjelmoinnin perusteet on jo käsitelty. Perusteet, jotka sallivat muuttujien muuttamisen ja käsittelevät tilakoneita on parasta tietää ennen funktionaalista paradigmaa, jotta aihetta opiskelevalle ohjelmoinnin testaaminen olisi suhteellisen helppoa.

8.3.3 Paradigmat ja niiden rajat

Mitä enemmän paradigmoihin liittyviä näkemyksiä käsitteleviin artikkeleihin perehtyy, sen myötämielisemmäksi tulee ajatukselle, että paradigmoja tulisi yhdistellä vapaammin. Paradigmat asettavat helposti rajat ohjelmoijan toiminnalle,

monesti perustellusti, mutta kuitenkin rajoittaen mielekkäät tai usein käytännölliset ratkaisut paradigman ulkopuolelle. On turhauttavaa tuntea ratkaisseensa jotain, kuitenkin löytäessään itsensä tuskailemasta sen todellisuuden kanssa, että kyseinen ratkaisu on joko kielletty tai ainakin vahvasti paheksuttu suunnittelu- tai ongelmanratkaisumalli.

Paradigmat ovat kuitenkin olemassa ymmärrettävästä syystä, sillä niiden kautta rajoitetaan pois sellaisia päätöksiä ja toteutuksia, jotka hankaloittavat kehitystä, tekevät koodin luettavuudelle haittaa tai muuten luovat ongelmia ohjelmistossa. Tästä huolimatta, sellainen paradigma, jonka pohjana on vahva suositus tietyille käsitteille, joka kuitenkin sallii siitä poikkeamisen perustelluissa tapauksissa, olisi kaikista paras niin kehityksen tehokkuudelle kuin myös luovuudelle, jossa ongelmanratkaisuun voitaisiin innovoida uusia tekniikoita.

Lähteet

Alexander, A. 2024. Disadvantages of Functional Programming. https://alvinalexander.com/scala/fp-book/disadvantages-of-functional-programming/. 12.2.2024.

Anand, A. 2023. Function Composition in Javascript: Exploring the Power of compose. Medium. https://medium.com/@akhilanand.ak01/function-composition-in-javascript-exploring-the-power-of-compose-4114da8b9875#:~:text=In%20functional%20programming%2C%20function%20composition,role%20in%20enabling%20function%20composition. 12.2.2024.

Asif, K. 2024. What is cyclomatic complexity? https://www.educative.io/answers/what-is-cyclomatic-complexity. 13.3.2024.

Church, A. 1936. An Unsolvable Problem of Elementary Number Theory. https://ics.uci.edu/~lopes/teaching/inf212W12/readings/church.pdf. 4.4.2024.

Deren, D. 2019. The Terms Explained: Programming Paradigms. Medium. https://blog.hackages.io/tech-terms-explained-programming-paradigms-8c4072404f8e. 17.3.2024.

Erlang. 2024. About. Erlang.org. https://www.erlang.org/about. 12.2.2024.

Feldman, R. 2019. “Why Isn’t Functional Programming the Norm?”. Clojutre. YouTube-video. https://www.youtube.com/watch?v=QyJZzq0v7Z4. 12.2.2024.

Fernández, M. 2009. Models of Computation: An Introduction to Computability Theory. Google-kirjat. 33. https://books.google.fi/books?id=FPFsnzzebhQC&pg=PA33&redir_esc=y#v=onepage&q&f=false. 17.3.2024.

Indeed editorial team 2023. What Are Functional Programming languages? With 27 Examples. Indeed. https://www.indeed.com/career-advice/career-development/functional-programming-languages. 17.3.2024.

Hughes, J. 1990. Why Functional Programming Matters. 2. The University of Glasgow. https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf. 14.2.2024.

Liyan, A. 2023. What is a programming paradigm? Medium. https://medium.com/@Ariobarxan/what-is-a-programming-paradigm-ec6c5879952b. 17.3.2024.

Neumann, J. 2022. Advantages and disadvantages of functional programming. Medium. https://medium.com/twodigits/advantages-and-disadvantages-of-functional-programming-52a81c8bf446. 13.2.2024.

Ruiz, B. 2022. Side effects. Medium. https://dev.to/ruizb/side-effects-21fc. 17.3.2024.

Scalfani, C. 2022. Why Functional Programming Should Be the Future of Software Development. IEEE Spectrum.

https://spectrum.ieee.org/functional-programming . 18.2.2024.
Thulie. 2024. Introduction to Functional Programming. Turing. https://www.turing.com/kb/introduction-to-functional-programming. 12.2.2024.
Tiobe. 2024. Demystifying Code Complexity. https://www.tiobe.com/knowledge/article/demystifying-code-complexity/. 4.4.2024.
Worsley, S. 2023. What is R? – An Introduction to The Statistical Computing Powerhouse. Datacamp. https://www.datacamp.com/blog/all-about-r 18.2.2024.