Tampereen yliopisto

Jyri Hakala

KATSAUS FUNKTIONAALISEN OHJELMOINNIN TILAAN JAVASSA

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Joulukuu 2022

TIIVISTELMÄ

Jyri Hakala: Katsaus funktionaalisen ohjelmoinnin tilaan Javassa
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Joulukuu 2022

Maailma sekä tietokonejärjestelmät kehittyvät jatkuvasti, mikä muuttaa niiden vaatimuksia ohjelmointikielillekin. Pystyäkseen vastaamaan uudenlaisiin vaatimuksiin, myös ohjelmointikielten on kehityttävä. Yksi yleinen evoluutioaskel viimeisen vuosikymmenen aikana on ollut lisätä funktionaalisen ohjelmoinnin ominaisuuksia lähtökohtaisesti proseduraalisiin kieliin, mikä on luonut joukon moniparadigmaisia ohjelmointikieliä. Tässä tutkielmassa tarkastellaan yhtä näistä, Java, ja pyritään selvittämään, ovatko ominaisuudet olleet onnistunut päivitys. Tutkielma on toteutettu kirjallisuuskatsauksena, jossa pääasiallisina lähteinä toimivat aiheesta aiemmin toteutetut tutkimukset. Useimmat aiemmista tutkimuksista keskittyvät vain yhteen ominaisuuteen, joten tämän tutkielman tulokset on saatu yhdistelemällä niitä.

Aluksi perehdytään funktionaalisen ohjelmoinnin ominaisuuksiin yleisesti, jotta voidaan ymmärtää, miksi niitä sulautetaan muiden paradigmojen kieliin. Ominaisuuksien todetaan helpottavan ohjelmien testaamista sekä parantavan koodin ilmaisuvoimaa. Ne myös mahdollistavat äärettömienkin tietorakenteiden käsittelyn sekä helpottavat ohjelman rinnakkaissuorittamisen toteuttamista. Ominaisuuksien käyttö on Javassa kuitenkin merkittävän vähäistä. Tutkittujen avoimen lähdekoodin ohjelmistojen sekä kirjastojen joukossa vain 16 % hyödyntää lambdafunktioita, 2 % Optional-kääretyyppiä ja 3 % Stream-rajapintaa. Yleisin syy ominaisuuksien hyödyntämiselle todetaan olevan niiden ytimekäs syntaksi, joka johtaa ohjelmakoodin parempaan luettavuuteen. Sen sijaan niiden käytöstä kieltäytymisen syiksi annetaan muun muassa heikompi suorituskyky ja virheidenkäsittelyn vaikeus.

Yleisin syy ominaisuuksien hyödyntämättä jättämiselle on kuitenkin tarve tukea vanhempia Java-versioita ja sen päälle rakennettuja ohjelmistoja. Tästä muodostuu tutkielman keskeisin havainto. Ohjelmistokielien on kehityttävä ja lisättävä uusia ominaisuuksia, mutta ominaisuuksia ei välttämättä aleta hyödyntämään, sillä tuki taaksepäin on ensiarvoisen tärkeää. Eikä olemassa olevan ohjelmiston uudelleen kirjoittaminen usein ole kannattavaa.

Avainsanat: funktionaalinen ohjelmointi, Java, lambdafunktio, Stream, Optional

SISÄLLYSLUETTELO

1 Johdanto

Funktionaalinen ohjelmointi on ohjelmointiparadigmana jo pitkäikäinen, johtaen juurensa aina 1950-luvulle asti. Sen suosio on koko historiansa ajan jäänyt kauas imperatiivisesta ohjelmoinnista, sekä myöhemmin olio-ohjelmoinnista, ja sen käyttö onkin joitakin poikkeuksia lukuun ottamatta jäänyt lähes akateemiseksi kuriositeetiksi. Ohjelmointikielien suosiota internetsivujen sisällön perusteella arvioiva TIOBE-indeksi (TIOBE, 2022) osoittaa funktionaalisen ohjelmoinnin käytön olevan nykypäivänäkin huomattavan harvinaista.

Funktionaalinen ohjelmointi esitteli ohjelmistokehitykseen useita käsitteitä, jotka ovat hyvin tunnettuja ja tutkittuja matemaattisten funktioiden parissa. Kyseisiin ideoihin on sittemmin alettu viittaamaan juuri funktionaalisen ohjelmoinnin ominaisuuksina. Ominaisuudet muun muassa helpottavat ohjelman toiminnan esittämistä todistuksien avulla, vähentävät tarpeettomien välioperaatioiden suorittamista sekä parantavat ohjelmakoodin ilmaisuvoimaa. (Hu et al., 2015) Lisäksi ominaisuuksien käyttö tekee ohjelman laskelmien rinnakkain suorittamisesta helpompaa (Khatchadourian et al., 2020).

Maailman ja sen tarpeiden muuttuessa myös ohjelmointikielet joutuvat kehittymään vastatakseen maailman odotuksiin. Tämän takia useimpiin suosituimmista lähtökohdiltaan imperatiivisiin kieliin on alettu sisällyttämään funktionaalisen ohjelmoinnin ominaisuuksia, tehden niistä niin kutsuttuja moniparadigmaisia kieliä. Tässä tutkielmassa keskitytään yhteen näistä, Javaan, ja pyritään selvittämään kuinka hyödyllinen lisä funktionaaliset ominaisuudet ovat olleet. Tämän lisäksi pohditaan hieman motiiveja ominaisuuksien lisäämisen takana.

Tämän tutkielman tutkimuskysymys on ”Käyttävätkö sovelluskehittäjät Java-ympäristön funktionaalisen ohjelmoinnin ominaisuuksia? Miksi tai miksi ei?” Aihetta on aiemmin tutkittu jonkun verran keskittymällä yksittäisiin ominaisuuksiin (Mazinanian et al., 2017; Nielebock et al., 2019; Petrulio et al., 2021; Khatchadourian et al., 2020), mutta kokonaiskuvan tarkastelua on tehty vähän (Tanaka et al., 2019). Tutkimukset osoittavat ominaisuuksien hyödynnysasteen jääneen verrattain pieneksi. Aiempien tutkimustulosten yhdistäminen sekä käytön alhaisuuden syiden pohdinta ovat tämän tutkielman tärkeimmät osiot.

Luvussa 2 kuvataan oleellisen tutkimusaineiston etsiminen sekä rajaaminen. Luvussa 3 puolestaan perehdytään funktionaalisen ohjelmoinnin ominaisuuksiin teoreettisella tasolla ja esitellään niiden väitetyt hyötypuolet. Seuraavaksi, luvussa 4, tarkastellaan ominaisuuksien ilmentymiä Javassa. Teorian optimistisen näkemyksen eroa käytännöstä käsitellään luvussa 5. Lisäksi siinä mietitään tutkielmassa käsiteltyjen tutkimusten puutteita ja pyritään yleistämään aiheesta opitut seikat. Tutkielman päättää luvun 6 yhteenveto.

2 Tutkimusmenetelmä

Tämä tutkielma on tyypiltään kirjallisuuskatsaus, mikä tarkoittaa, että menetelmän keskeisin osuus on aineiston kerääminen. Tutkielman rakennetta mukaillen myös tiedonhaku toteutettiin kaksivaiheisena. Molemmissa vaiheissa hakuja suoritettiin tietojenkäsittelytieteen piiriin kuuluvia artikkeleita indeksoiviin tietokantoihin. Hyödynnetyt tietokannat olivat ACM Digital Library, ProQuest: Computer Science Database sekä IEEE Electronic library. Näiden lisäksi hakujen suorittamiseen käytettiin Tampereen yliopiston Andor-hakukonetta, joka sisältää materiaaleja myös muista kuin edellä mainituista tietokannoista. Hakuja suoritettiin yksinomaan englannin kielellä, sillä se on yleisesti käytetyin kieli tietojenkäsittelyalalla. Kaikki haut suoritettiin syys- ja marraskuun välisenä aikana vuonna 2022.

Tiedonhaun ensimmäinen vaihe keskittyi Javan funktionaalisien ominaisuuksien käytännön hyödyntämisen selvittämiseen. Ominaisuudet lisättiin Javaan vuonna 2014, joten myös hakutulokset rajattiin alkamaan kyseisestä vuodesta. Tärkeimmät hakutermit olivat ”java” sekä ”functional” ja ominaisuuksien nimet, eli: ”optional”, ”lambda” ja ”stream”. Hakusanoja yhdisteltiin mahdollisuuksien mukaan loogisilla operaattoreilla AND ja OR. Kaiken kattava hakulause oli muotoa ”java AND (‘functional programming’ OR optional OR lambda OR stream)”.

Toisessa vaiheessa hakua keskityttiin funktionaalisen ohjelmoinnin ja sen ominaisuuksien väitettyihin hyötypuoliin sekä niiden teoriaan. Tärkeimpinä hakutermeinä toimivat ”functional programming” sekä hyötypuolia kuvaavat synonyymit ”benefit” ja ”advantage”. Termejä yhdistämällä saatiin lopulliseksi hakulauseeksi ”’functional programming’ AND (benefit OR advantage)”.

Molemmissa vaiheissa hakutermit osoittautuivat turhan yleisluontoisiksi ja tämän vuoksi löydettyjä hakutuloksia karsittiin niiden otsikoiden sekä tiivistelmien perusteella. Ensimmäisessä vaiheessa mukaan pyrittiin ottamaan artikkeleita, jotka koskevat aihetta käsittelevää empiiristä tutkimusta. Toisen vaiheen teoriapainotteisuuden vuoksi mukaan hyväksyttiin myös muunlaisia artikkeleita.

Artikkeleiden keskeisimmät sisällöt olivat hyvin samansuuntaisia, joten niitä ei ollut tarvetta luokitella. Aineistosta onkin täten poimittu tähän tutkielmaan käsiteltäväksi niissä havaittuja yhteneväisyyksiä ja päällekkäisyyksiä.

3 Funktionaalinen ohjelmointi yleisesti

Funktionaalisen ohjelmoinnin historia yltää teoriatasolla aina 1930-luvulle asti Alonzo Churchin kehittämän lambdakalkyylin muodossa. Sen ensimmäinen käytännön esimerkki, Lisp-ohjelmointikieli, julkaistiin 1950-luvulla. Imperatiivisen ohjelmoinnin historia on vielä tätäkin pidempi ja se on läpi paradigmojen yhteisen historian ollut ohjelmistokehittäjien ensimmäinen valinta ohjelmistoja kehitettäessä. Vuosituhannen vaihteen jälkeen funktionaalinen ohjelmointi on kuitenkin alkanut kasvattamaan suosiotaan ja sen ominaisuuksia on sisällytetty myös imperatiivisiin ohjelmointikieliin. Yksi syy muutokselle on funktionaalisen ohjelmoinnin tarjoama parempi tuki rinnakkaisuudelle, joka mahdollistaa modernien moniydinprosessorien paremman hyödynnysasteen ohjelmissa. (Hinsen, 2009)

Tässä luvussa tutustutaan funktionaalisen ohjelmoinnin ydinkäsitteisiin. Niistä pyritään tuomaan esille hyötypuolet, jotta voidaan ymmärtää imperatiivisten ohjelmistokielien kehittäjien motiiveja lisätä kyseisiä ominaisuuksia kieliinsä. Aliluvuissa käsitellään tarkemmin tutkielman kannalta oleellisimmat ominaisuudet.

Funktionaalinen ohjelmointi perustuu nimensä mukaisesti funktioihin. Nämä funktiot ovat hyvin läheisesti yhteydessä matemaattisiin funktioihin, jotka kuvaavat jonkin syötearvon toiseksi, mahdollisesti eriäväksi, arvoksi. Funktiot eivät muuta saamaansa syötearvoa vaan palauttavat uuden, kuvauksen jälkeisen, arvon. Ne myös palauttavat samalla syötearvolla aina saman tuloksen, eivätkä ne vaikuta funktion ulkopuolisiin arvoihin mitenkään, eli niillä ei ole sivuvaikutuksia. Tämän kaltaista funktioiden ominaisuutta kutsutaan niiden puhtaudeksi. (Hinsen, 2009) Funktioiden puhtaus helpottaa ohjelmien oikeellisuuden todistamista, mikä puolestaan vähentää testaamiseen tarvittua aikaa. (Hu et al., 2015)

Funktioilla on myös toinen tärkeä ominaisuus, joka pohjautuu niiden puhtauteen. Niiden määritelmää sekä lopputulosta voidaan pitää täysin yhtäläisinä. Tätä kutsutaan funktioiden *viitteelliseksi läpinäkyvyydeksi* (referential transparency). Viitteellinen läpinäkyvyys mahdollistaa funktioiden kiinnittämisen nimettyihin muuttujiin ilman, että niiden määrittelemiä laskentoja suoritetaan.

3.1 Korkeamman asteen funktiot sekä nimettömät funktiot

Korkeamman asteen funktio on funktio, joka ottaa parametrinaan yhden tai useamman funktion tai palauttaa tuloksenaan funktion. Ohjelmointikielissä yleisimpiä korkeamman asteen funktioita ovat map, filter, sekä reduce. Map suorittaa parametrinaan saaman funktion tietorakenteen jokaiselle alkiolle. Filter puolestaan suodattaa tietorakenteesta pois ne alkiot, jotka toteuttavat sille annetun predikaatin. Reduce eroaa kahdesta edellisestä siten, että se yhdistää tietorakenteen alkiot halutulla tavalla yhdeksi tulokseksi, esimerkiksi las-

kemalla ne yhteen. Reducea kutsutaan usein myös nimellä fold. Nämä ovat tärkeitä konsepteja sivuvaikutuksettomassa funktionaalisessa ohjelmoinnissa tietorakenteiden läpikäymiseen imperatiivisen ohjelmoinnin silmukkarakenteiden sijaan.

Kuten edellä selvisi, korkeamman asteen funktiot voivat käyttäytyä täysin eri tavoin eri suorituskerroilla. Käyttäytymistä kuvataan niille parametrina annettavien funktioiden avulla, mikä mahdollistaa muuttuvien tilanteiden vaatimuksiin joustavasti vastaamisen. Idealla on osuva nimi: *käyttäytymisen parametrisointi* (behavior parameterization). (Urma et al., 2015)

Nimetön, tai anonyymi, funktio on funktio, jolle ei ole annettu tunnistetta. Toisin sanoen sitä ei ole sidottu muuttujaan. Ne liittyvät läheisesti Churchin lambdakalkyyliin, jossa kaikki funktiot ovat nimettömiä, minkä ansiosta niitä kutsutaan myös lambdafunktioiksi. Lambdafunktioita käytetään usein juuri edellä esiteltyyn käyttäytymisen parametrisointiin, sillä ne voivat kuvata kertaluontoisen käyttäytymismallin hyvinkin tiiviisti verrattuna nimettyihin funktioihin.

Lambda- sekä korkeamman asteen funktioiden käyttäminen mahdollistaa monimutkaistenkin operaatioiden kuvaamisen verrattain lyhyesti. Niiden parempi luettavuus ja ytimekkyys ovatkin yleisimmät syyt miksi ohjelmoijat sisällyttävät niitä ohjelmiinsa. Myös käyttäytymisen parametrisointi on ominaisuuksien yleinen käyttökohde. (Mazinanian et al., 2017) Hu ja muut (2015) esittävät, että lyhyempi ja selkeämpi koodi johtaa parempaan tuottavuuteen sekä vähentää ohjelmointivirheiden määrää.

3.2 Monadit, erityisesti maybe

Monad on abstrakti tietotyyppi, jonka avulla voidaan yhdistellä ohjelman suorittamiseen vaadittavia funktioita sekä niiden paluuarvoja halutuissa konteksteissa. Monad toimii kääreenä paluuarvon ympärillä. Yksinkertaisimmillaan monad määrittelee kaksi operaatiota: *return*, joka käärii saamansa arvon monadiin, sekä *bind*, joka puolestaan välittää monadin sisältämän arvon saamalleen funktiolle. Puhtaasti funktionaalisessa ohjelmoinnissa monadeita voidaan käyttää funktioiden yhdistämiseen liukuhihnamaisiksi rakenteiksi, jotka abstrahoivat ohjelman kontrollivuon sekä auttavat käsittelemään sen sivuvaikutuksia.

HaskellWiki (2021) esittää monadeille kolme ominaisuutta, jotka tekevät niistä hyödyllisiä. Monadit ovat modulaarisia, sillä ne mahdollistavat monimutkaisten laskelmien muodostamisen yhdistelemällä yksinkertaisempia osasia. Toiseksi ne ovat joustavia ja tekevät ohjelmista paremmin sopeutuvia alati muuttuviin tarpeisiin. Kolmanneksi ne auttavat eristämään ohjelman käytettävyyden kannalta tärkeät sivuvaikutukset ja suorituksen tilan ohjelman puhtaista osista.

Monadit ovat huomattavan monimutkainen ja laaja aihe, eikä niiden ominaisuuksien tai eri ilmentymien läpi käyminen ole tässä mahdollista kandidaatin tutkielman rajallisuuden vuoksi. Seuraavaksi esitellään kuitenkin näistä yksi, joka on tämän tutkielman aiheen kannalta oleellisin.

Maybe, tai *option*, -tyyppi on monadi, joka kuvaa laskennan mahdollista epäonnistumista. Sen sisältämät arvot ovat tyypillisesti tyhjää arvoa tai epäonnistumista kuvaava *Nothing* sekä tämän vastakohta *Just* tai *Some*. Maybe mahdollistaa liukuhihnamaisten suoritusketjujen selviämisen virhetilanteista, sillä minkä tahansa ketjun osan epäonnistuessa tulee koko liukuhihnan tulokseksi tyhjä arvo. Maybe on hyödyllinen myös imperatiivisessa ohjelmoinnissa, sillä se auttaa ohjelmoijia välttämään toistuvaa *null* -viittausten tarkastelua, mikä johtaa selkeämpään ja luettavampaan koodiin (Urma et al., 2015).

3.3 Laiska evaluointi

Laiska evaluointi on tyypillinen ominaisuus funktionaalisissa ohjelmointikielissä. Se tarkoittaa sitä, että ohjelmassa esitettyjä laskelmia suoritetaan vain silloin, kun se on ehdottoman tarpeellista. Esimerkiksi funktion kaikkien parametrien arvoa ei tarvitse välttämättä selvittää riippuen sen kontrollivuosta, eikä sen paluuarvolla ole väliä ennen kuin sitä hyödynnetään jossakin ohjelman aikakriittisessä osassa, kuten tuloksen ruudulle tulostamisessa. Laiskan evaluoinnin mahdollistavat arvojen muuttumattomuus sekä viitteellinen läpinäkyvyys.

Laiska evaluointi mahdollistaa äärettömienkin tietorakenteiden käsittelyn ohjelmissa. Niitä voidaan kuvata halutunlaisilla säännöillä, esimerkiksi ”kaikki luonnolliset luvut”, mutta koko tietorakennetta ei tarvitse kirjoittaa muistiin, vaan siitä käsitellään vain tarvittu osa. Tämä käsittelystrategia auttaa vähentämään muistin tarvetta myös äärellisiä rakenteita käsitellessä.

Jo Hughes (1989) esitteli laiskan evaluoinnin hyötypuolia artikkelissaan, joka käsittelee erilaisia numeerisia menetelmiä sekä algoritmia, joka arvioi kuinka hyvä pelaajan positio on tietyssä pelissä. Laiskan evaluointistrategian käyttö hyödyttää abstrahointia ja erityisesti ohjelman modulaarisuutta. Hughes kutsuukin sitä mahdollisesti tärkeimmäksi ”liimaksi” ohjelman osien yhdistelemiselle, mitä funktionaalisella ohjelmoijalla on hallussaan.

4 Funktionaalisia ominaisuuksia Javassa

Javasta julkaistiin versio 8 vuonna 2014. Sen muutoksien ja lisäyksien joukossa on myös tuki joillekin funktionaalisen ohjelmoinnin ominaisuuksille. (Oracle, 2022) Tässä luvussa tutustutaan lisättyihin ominaisuuksiin ja esitellään, miten ne liittyvät edellisen luvun käsitteisiin. Lisäksi kullekin ominaisuudelle pyritään selvittämään hyödyntämisaste sekä pohtimaan, onko se ollut onnistunut lisä Javan työkalupakkiin.

4.1 Lambdafunktiot

Javassa nimettömät funktiot pohjautuvat rajapintoihin, jotka määrittelevät tasan yhden abstraktin metodin. Kyseisiä rajapintoja kutsutaan funktionaalisiksi rajapinnoiksi. Rajapinnan metodilla on tietynlainen muoto, tai jalanjälki, joka esittää yksikäsitteisesti funktion parametrien ja paluuarvon tyypit. Esimerkiksi Predicate-rajapinta ottaa vastaan minkä tahansa tyypin ja tuottaa tulokseksi totuusarvon. Lambdafunktiot ovat näiden abstraktien metodien toteutuksia ja niiden tulee mukailla metodin jalanjälkeä.

Javassa oli jo entuudestaan keino luoda kertakäyttöisiä olioita, joita ei sidota muuttujaan. Näitä kutsutaan nimettömiksi luokiksi ja niitä voidaan lambdafunktioiden tapaan käyttää käyttäytymisen parametrisointiin. Nimettömän luokan luomiseksi tulee kuitenkin käyttää Javan täyttä luokan luomisen syntaksia, mikä tekee siitä kehittäjille raskasta. Lambdafunktioilla voi toteuttaa nimettömien luokkien toiminnot, ja ne leikkaavat luomisesta toistuvat osat pois, joten ne ovat hyvä vaihtoehto, kun tarvitaan lyhytikäisiä olioita.

Lambdafunktiot olivat näennäisesti onnistunut ja suosittu lisäys Javaan, sillä niiden käyttöasteessa oli selvä kasvava trendi päivityksen jälkeisinä vuosina. Yleisimmiksi syiksi lambdafunktioiden hyödyntämiselle ohjelmissa mainitaan niiden ytimekkyys, mahdollisuus välttyä määrittelemästä uusia luokkia ja käyttäytymisen parametrisointi. Vaikka niillä koetaan olevan useita hyötyjä, lambdafunktioita käytti vuonna 2017 vain noin 27 % ohjelmoijista. (Mazinanian et al., 2017)

Kasvutrendistä huolimatta lambdafunktiot ovat jääneet ohjelmistoprojekteissa melko harvinaisiksi. Sadan suosituimman avoimen lähdekoodin Java projektin joukossa vain 16 % sisälsi huomattavan määrän niitä (Tanaka et al., 2019). Java-ekosysteemin 50 suosituimman avoimen rajapinnan joukossa tilanne on vielä heikompi: vain 9 tukee lambdafunktioiden käyttöä (Petrulio et al., 2021). Nimettömiä funktioita hyödynnetään selkeästi enemmän testien toteuttamisessa kuin varsinaisessa ohjelmassa (Mazinanian et al., 2017; Nielebock et al., 2019).

Asiasta haastatellut ohjelmistokehittäjät kertovat syiksi lambdafunktioiden käyttämättömyydelle niiden aiheuttamat vaikeudet poikkeuksien käsittelyssä sekä tarpeen tukea vanhempia versioita (Tanaka et al., 2019). Osa projekteista on omaksunut lambdafunktiot, mutta myöhemmin poistanut ne käytöstä. Näiden projektien syyt poistoille eroavat käytöstä kieltäytyvistä. Kaksi yleisintä syytä poistamiselle käsittivät yhteensä lähes puolet kaikista poistoista. Yleisin näistä on suorituskyvyn havaittu heikentyminen ja toiseksi yleisin luettavuuden heikentyminen, varsinkin pitkissä lambdafunktioissa. (Zheng et al., 2021)

4.2 Optional

Optional on kääretyyppinen, muita objekteja mahdollisesti sisältävä objekti. Se tarjoaa muun muassa metodeja sisäisen objektin olemassaolon tarkasteluun, käärimiseen ja pur-

kamiseen sekä vakiopaluuarvon määrittämiseen, jos sisäistä objektia ei ole olemassa. Optionalin tarkoitus on ehkäistä ohjelman suorituksen aikana tyhjiin arvoihin viittaamista, joka johtaa poikkeustilaan ja ohjelman kaatumiseen. Urma ja muut (2015) toteavat *NullPointerException* -poikkeuksen olevan Javan poikkeuksista kaikkein yleisin.

Ohjelmistokehittäjät luottavat kuitenkin selvästi enemmän perinteiseen keinoon tarkastella viitteiden eheyttä, vaikka se johtaakin usein koodin luettavuuden alenemiseen tai sen rakenteen epäselkeyteen. Suosituimmista Java-projekteista vain 2 % käyttää aktiivisesti Optionalia (Tanaka et al., 2019). Syy suosion puutteelle saattaa olla, että Optional ei varsinaisesti ratkaise tyhjien viittausten ongelmaa vaan pakottaa ohjelmoijan kiinnittämään asiaan huomiota kehittämisen aikana. Virheelliset viittaukset on kuitenkin helppo karsia pois ohjelmasta myös testaamisen avulla. Lisäksi Optionalin hyödyntämisen tulee olla harkittua ja se kannattaa huomioida jo suunnitteluvaiheessa. Perinteinen null-tarkastelu saattaa olla myös ohjelmistokehittäjille tutumpaa muiden kielien parissa työskentelemisestä.

4.3 Stream-rajapinta

Laiskan evaluoinnin toteuttaminen imperatiivisessa ohjelmointikielessä on hankalaa, koska virheiden sekä sivuvaikutusten käsittely vaatii ohjelmalta tiettyä suoritusjärjestystä. Tämän vuoksi Javankin evaluointistrategiaksi on valittu laiskuuden sijaan tiukka, tai innokas, evaluointi. On kuitenkin mahdollista simuloida laiskaa evaluointia ja hyödyntää sitä paikoittain, kuten tietorakenteiden käsittelyyn, vaikka ympäristö olisikin rakennettu innokkaan evaluoinnin päälle.

Java 8 esitteli kieleen Stream-rajapinnan. Se tarjoaa keinot tietovirtojen luomiseen erilaisista lähteistä, kuten Javan tietorakenne-kirjaston Collection jäsenistä. Stream onkin sulautettu Collection-kirjastoon ja sitä voidaan hyödyntää kokoelmien yhteydessä saumattomasti. Luodut tietovirrat perivät joitakin lähteiden ominaisuuksista. Esimerkiksi lähteen alkioiden ollessa tietyssä järjestyksessä, myös virran alkiot ovat samassa järjestyksessä.

Tietovirtaoperaatioita yhdistelemällä voidaan luoda liukuhihnoja tiedon rajaamiseksi, muuttamiseksi tai yhdistämiseksi. Javassa operaatiot on jaettu väli- sekä päätösoperaatioihin. Välioperaatioiden tuloksena on aina uusi muunnettu tietovirta ja niitä voi ketjuttaa haluamallaan tavalla. Päätösoperaatiot puolestaan tuottavat tiedon perusteella jonkinlaisen tuloksen tai sivuvaikutuksen, kuten näytölle tulostamisen. Ne voivat myös kerätä tietovirran alkiot haluttuun tietorakenteeseen. Tietovirrat ovat laiskasti evaluoituja, eikä niille suoriteta laskentaa ennen päätösoperaatiota. Kunkin virran katsotaan evaluoinnin jälkeen olevan kulutettu, eikä sille voi enää suorittaa operaatioita. Virrat eivät myöskään pääse käsiksi alkuperäisen lähteen alkioihin eivätkä täten koskaan muuta niitä.

Oletusarvoisesti virtojen liukuhihnat suoritetaan sarjassa. Toisin sanoen operaatioita käydään läpi yksi kerrallaan. Java kuitenkin tarjoaa myös mahdollisuuden rinnakkaiselle

suorittamiselle, jossa operaatioita jaetaan samanaikaisesti suoritettavaksi usealle laskentayksikölle. Rinnakkaisuuden hyödyntämisestä on tehty helppoa: se käyttää samoja operaatioita ja rakennetta kuin sarjassa suorittaminen. Ainoa ero on virran avaamiseksi käytetty metodikutsu.

Stream-rajapintaa hyödynnetään kuitenkin vain noin 3 %:ssa suosituimmista Java-kielisistä projekteista. Useimmat käytöstä kieltäytyvät ilmoittavat syyksi huonon suorituskyvyn. (Tanaka et al., 2019) Sarjassa (eli hyödyntämättä rinnakkaisuutta) suoritettavat tietovirrat voivatkin olla moninkertaisesti hitaampia kuin tavanomainen iterointi, riippuen välioperaatioiden ja niiden aiheuttamien kutsujen määrästä. Tietovirtaoperaatiot voi kuitenkin suurimmalta osin optimoida yhtä tehokkaaksi. (Møller & Veileborg, 2020)

Herää kuitenkin kysymys siitä, miksi ohjelmistokehittäjä vaivautuisi käyttämään Stream-rajapintaa, jos imperatiivisen tavan iterointi on vähintään yhtä suorituskykyistä eikä vaadi erillisen optimoinnin suorittamista. Alhainen hyödyntämisaste viittaa monen kehittäjän pohtineen samaa kysymystä. Helppo rinnakkaiskäsittelykään ei ole houkutellut kehittäjiä käyttämään tietovirtoja. Vain 1,25 % Stream-rajapintaa hyödyntävistä projekteista sisältää rinnakkain suoritettavia virtoja (Khatchadourian et al., 2020). Kaikkia liukuhihnoja ei kannatakaan suorittaa rinnakkaisina. Jotkin välioperaatiot, kuten alkioiden järjestäminen, tarvitsevat tietoa ympäröivästä tilasta ja niiden suorittaminen rinnakkain voi vaatia alkioiden läpi käymistä useita kertoja (Oracle, 2017). Myös järjestyksessä olevasta lähteestä avatut tietovirrat voivat joissain tapauksissa kärsiä heikommasta suorituskyvystä (Khatchadourian et al., 2020).

Heikon suorituskyvyn lisäksi Stream-rajapinnan käytön harvinaisuuteen vaikuttaa tarve tukea vanhempia ohjelman osia tai kirjastoja. Yleisin virran päätösoperaatio on collect, joka kerää alkiot kokoelmaan. Collect päättää jopa 54 % tietovirroista (Tanaka et al., 2017). Tähän viittaavat myös Khatchadourian ja muut (2020) ja he näkevät sen johtuvan kehittäjien tarpeesta esitellä prosessiin sivuvaikutuksia tai siitä, että tietovirralla lasketun tuloksen on oltava yhteensopiva jo olemassa olevan ohjelmistokoodin kanssa.

5 Keskustelu

Kokonaisuutena funktionaalisen ohjelmoinnin ominaisuuksien käyttö on varsin harvinaista Javassa. Ominaisuuksien väitetyistä hyötypuolista huolimatta vain murto-osa projekteista sisältää funktionaalisia ominaisuuksia, vaikka niiden lisäämisestä kieleen oli tutkielmassa käsiteltävien tutkimusten toteuttamisen hetkellä ehtinyt kulua yli 5 vuotta. Muutoksen hitauden saattaisi selittää koodin refaktoroinnin vaikeus, mutta sen voi suurelta osin automatisoida (Gyori et al., 2013). Täten muutosnopeuden taustalla on oltava toisenlaiset tekijät.

Yksi vaikuttava tekijä on ominaisuuksien hyötyjen toteutumattomuus käytännössä. Optional ei varsinaisesti helpota puuttuvien arvojen käsittelyä, vaan pelkästään muuttaa

sen ulkomuotoa, eikä muun tyyppisiä monadeja ole sisällytetty Javan peruskirjastoihin. Stream-rajapinta ja sen tarjoamat deklaratiiviset tietovirtaoperaatiot ovat lähes aina hitaampia kuin tietorakenteiden iterointi imperatiiviseen tyyliin. Rinnakkaisuuskaan ei ole kohottanut tietovirtojen suosiota ja vain harva kehittäjä hyödyntää niitä, mikä viittaisi siihen, että ohjelman toiminnan tarkastelu on helpompaa, kun se suoritetaan sarjassa. Funktionaalisten ominaisuuksien suosituimmaksi hyödyksi on osoittautunut koodin ilmaisuvoiman paraneminen, mihin eniten vaikuttavat lambdafunktiot. Niiden lyhyt ja ytimekäs syntaksi on tehnyt niistä selvästi käytetyimmän Javan funktionaalisen ominaisuuden. Ohjelmistokehittäjät suosivat ilmaisuvoimaista koodia, sillä se säästää aikaa sekä ohjelmistojen kehittämisessä että niiden testaamisessa.

Toinen ja vielä tärkeämpi tekijä on tuki vanhalle koodille. Varsinkin yhteensopivuus vanhempien suoritusympäristöjen kanssa on taattava. Tämä nousi aihetta käsittelevissä tutkimuksissa toistuvasti esiin isoimpana tekijänä funktionaalisten ominaisuuksien käyttämättömyydelle tai niiden poistolle. Tutkielmassa tarkasteltavan Java 8:n tapauksessa tämä tarkoittaa sitä, että useissa tapauksissa ominaisuuksien hyödyntäminen ei ole kannattavaa tai edes mahdollista. Javan kaltaiseen massiiviseen ohjelmointiympäristöön, joka on suosittu isoissa ohjelmistoprojekteissa, tämä luo valtavan määrän inertiaa, joka hidastaa uudempien versioiden omaksumista huomattavasti. Toisin sanoen olemassa olevia ohjelmia ja kirjastoja kirjoitetaan uusiksi lähinnä pakon edessä.

Asiain tilasta tekee erityisen merkittävän se, että aihetta käsittelevissä tutkimuksissa keskitytään, ymmärrettävistä syistä, tarkastelemaan avoimen lähdekoodin projekteja. Avoimen lähdekoodin projektien voi olettaa olevan ketterämpiä ja nopeampia tekemään muutoksia kuin suljettujen vastineiden, sillä niiden parissa työskentelee usein monimuotoisempi joukko kehittäjiä, eikä niiden jatkuva toimiminen välttämättä ole ehtona liiketoiminnan tulevaisuudelle. Monet avoimien projektien kehittäjistä ovat kuitenkin myös tekemisissä suljetun lähdekoodin kanssa ja he tietävät, että yhteensopivuuden takaaminen on tärkeää. Tämä pätee erityisesti kirjastoihin.

Suljetun lähdekoodin tarkastelemisessa piilee aiheen kannalta tärkein jatkokehitysidea. Yrityksissä kehitettävien projektien profiloiminen tuottaisi merkittävästi enemmän dataa ja siten tarkemman kuvan funktionaalisten ominaisuuksien todellisesta hyödynnysasteesta. Laajemman datan pohjalta voisi tehdä vahvempia tulkintoja ominaisuuksien lisäämisestä ohjelmointikieliin.

Entä onko Java ainoa ympäristö, jossa muutokset omaksutaan hitaasti? Tuskin on, sillä vanhempien versioiden tukeminen on universaali ongelma. Pitkäikäisissä ohjelmointikielissä tapahtuu vääjäämättä suuriakin muutoksia ulkoisten tekijöiden, kuten muuttuvien tietokonearkkitehtuurien, vuoksi. Suosionsa säilyttäneillä pitkän historian omaavilla kielillä onkin yksi yhteinen tekijä: niitä päivitetään jatkuvasti. Tämä luo mielenkiintoisen

asettelun kielien kehittäjien kannalta. Yhdeltä kantilta kielien päivittäminen vaikuttaa turhalta, koska uusia ominaisuuksia ei välttämättä hyödynnetä, kun taas toisaalta kieltä on pakko kehittää, jotta se pystyy vastaamaan moderneihin tarpeisiin eikä joudu uudempien kielien syrjäyttämäksi. Uusien ohjelmointikielten suunnittelussa tulisikin pitää tämä mielessä.

6 Yhteenveto

Tutkielma aloitettiin esittelemällä lyhyesti funktionaalisen ohjelmoinnin historiaa sekä taustoja sen ominaisuuksien lisäämiselle muiden ohjelmointiparadigmojen kieliin. Tarkempaan tarkasteluun valikoitui Java sen jatkuvan suosion sekä löydetyn materiaalin runsauden vuoksi. Javaan lisättiin funktionaalisen ohjelmoinnin ominaisuuksia versionumerossa 8, joka julkaistiin vuonna 2014. Nämä ominaisuudet olivat tutkielmassa käsitellyt lambdafunktiot, Optional ja Stream-rajapinta.

Seuraavaksi tutustuttiin tutkielman kannalta keskeisimpiin funktionaalisen ohjelmoinnin ominaisuuksiin. Tarkastelu toteutettiin teoriapainotteisena ja siinä keskityttiin ominaisuuksien väitettyihin hyötypuoliin. Esittelyn tarkoitus oli valaista imperatiivisen paradigman ohjelmistokielien kehittäjien mahdollisia motiiveja funktionaalisten kielien ominaisuuksien lisäämiseen. Ominaisuuksien todettiin mahdollistavan funktioiden käyttäytymisen parametrisoinnin, liukuhihnamaisten suoritusrakenteiden luomisen sekä äärettömienkin tietorakenteiden käsittelyn ohjelmassa.

Teoriassa saavutettavat hyödyt eivät kuitenkaan aina toteudu käytännössä, ja näin näytettiin tapahtuvan myös Javassa. Kieleen lisätyistä ominaisuuksista käytetyimmäksi osoittautuivat lambdafunktiot, joita on helppo välittää parametrina muille funktioille, mutta niiden suurimmaksi hyödyksi todettiin niiden yksinkertaisen syntaksin tuoma ilmaisuvoima ja luettavuus. Arvon puuttumista kuvaava Optional sekä tietoalkioita ja tietorakenteita funktionaaliseen tyyliin käsittelevä Stream-rajapinta sen sijaan jäivät käyttöasteeltaan minimaalisiksi. Optional ei tarjoa ohjelmoijille merkittävää etua ja perinteisen iteroinnin todettiin olevan tehokkaampaa kuin tietovirtojen.

Tärkeimmäksi esteeksi uusien ominaisuuksien käytön omaksumiselle nähtiin vanhojen versioiden tukemisen tärkeys. Sen kokivat esteeksi niin avoimen lähdekoodin kirjastojen kuin ohjelmistojenkin kehittäjät. Tuki taaksepäin on erityisen keskeistä Javassa, sillä useat suurimmista projekteista on kehitetty jo ennen versiota 8. Oracle myös tarjoaa yhä tänäkin päivänä virallista kaupallista tukea vanhemmille versioille. Nämä seikat aiheuttavat ekosysteemin hitaan muutostahdin.

Vaikka käyttöaste jäisikin alhaiseksi ja muutoksien omaksuminen olisi hidasta, niin ohjelmistokieliin on tärkeää lisätä erilaisia uusia ominaisuuksia, jotta ne pystyvät vastaamaan ympäröivän maailman vaatimuksiin. Tämä lieneekin tutkielman tärkein havainto, vaikka sitä ei alun perin pyrittykään tutkimaan.

Funktionaalisen ohjelmoinnin ominaisuuksien lisääminen erilaisiin ohjelmistokieliin on kannattavaa, vaikka tarkastelussa ollut Java ei sitä osoittaisikaan. Niillä on vahva teoriapohja ja ne pystyvät tuomaan kieleen monia hyötyjä. Ominaisuuksien hyödyntäminen lieneekin yleisempää nopeamman evoluutiotahdin omaavissa ympäristöissä, kuten JavaScriptissä, ja ne ovat alusta alkaen osa joitakin uusia ohjelmointikieliä, esimerkiksi Rustia. Javan kanssa samalla virtuaalikoneella suoritettava Scala saattaa myös osaltaan vaikuttaa Javan tilanteeseen. Scala on lähes puhtaasti funktionaalinen ohjelmointikieli ja sen avulla on helppo rakentaa osia myös Java-pohjaisiin projekteihin. Scalaa hyödynnetään erityisesti tiedon käsittelyyn ja se toimii pohjana Apache Spark -ohjelmistolle, joka on laajasti käytetty datatieteessä. (Apache, 2022)

Lähdeluettelo

Apache (2022). Apache Spark. *https://spark.apache.org/ (haettu 20.12.2022)*

Gyori, A., Franklin, L., Dig, D., & Lahoda, J. (2013). Crossing the gap from imperative to functional programming through refactoring. 2013 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013 - Proceedings, 543–553. *https://doi.org/10.1145/2491411.2491461*

HaskellWiki (2021). All about monads. *https://wiki.haskell.org/All_About_Monads (haettu 7.11.2022)*

Hinsen, K. (2009). The Promises of Functional Programming. Computing in Science & Engineering, 11(4), 86–90. *https://doi.org/10.1109/MCSE.2009.129*

Hu, Z., Hughes, J., & Wang, M. (2015). How functional programming mattered. National Science Review, 2(3), 349–370. *https://doi.org/10.1093/nsr/nwv042*

Hughes, J. (1989) Why functional programming matters. Computer Journal, 32(2), 98–107. *https://doi.org/10.1093/comjnl/32.2.98*

Khatchadourian, R., Tang, Y., Bagherzadeh, M., & Ray, B. (2020). An empirical study on the use and misuse of java 8 streams. Fundamental Approaches to Software Engineering (FASE 2020), 12076, 97–118. *https://doi.org/10.1007/978-3-030-45234-6_5*

Mazinanian, D., Ketkar, A., Tsantalis, N., & Dig, D. (2017). Understanding the use of lambda expressions in Java. Proceedings of ACM on Programming Languages, 1(OOPSLA), 1–31. *https://doi.org/10.1145/3133909*

Møller, A., & Veileborg, O. H. (2020). Eliminating abstraction overhead of Java stream pipelines using ahead-of-time program optimization. Proceedings of ACM on Programming Languages, 4(OOPSLA), 1–29. *https://doi.org/10.1145/3428236*

Nielebock, S., Heumüller, R., & Ortmeier, F. (2019). Programmers do not favor lambda expressions for concurrent object-oriented code. Empirical Software Engineering:

an International Journal, 24(1), 103–138. *https://doi.org/10.1007/s10664-018-9622-9*

Oracle (2017). java.util.stream (Java SE 9 & JDK 9) *http://docs.oracle.com/javase/9/docs/api/java/util/stream/package-summary.html (haettu 26.11.2022)*

Oracle (2022). What's New in JDK 8. *https://www.oracle.com/java/technologies/javase/8-whats-new.html (haettu 20.11.2022)*

Petrulio, F., Sawant, A. A., & Bacchelli, A. (2021). The indolent lambdification of Java: Understanding the support for lambda expressions in the Java ecosystem. Empirical Software Engineering: an International Journal, 26(6). *https://doi.org/10.1007/s10664-021-10039-9*

Tanaka, H., Matsumoto, S., & Kusumoto, S. (2019). A Study on the Current Status of Functional Idioms in Java. IEICE Transactions on Information and Systems, E102.D(12), 2414–2422. *https://doi.org/10.1587/transinf.2019MPP0002*

TIOBE (2022). TIOBE Index for December 2022. *https://www.tiobe.com/tiobe-index/ (haettu 15.12.2022)*

Urma, R-G., Mycroft, A., & Fusco, M. (2015). Java 8 in action: lambdas, streams, and functional-style programming (1st edition). Manning.

Zheng, M., Yang, J., Wen, M., Zhu, H., Liu, Y., & Jin, H. (2021). Why Do Developers Remove Lambda Expressions in Java? 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) 2021, 67–78. *https://doi.org/10.1109/ASE51524.2021.9678600*