Rami Pasanen

Lambda-lausekkeet tietorakenteiden käsittelyssä C#-kielessä: satunnaistettu vertailukoe

Tietotekniikan pro gradu -tutkielma

3. helmikuuta 2021

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Rami Pasanen

Yhteystiedot: rami_p@outlook.com

Ohjaajat: Jonne Itkonen ja Antti-Jussi Lakanen

Työn nimi: Lambda-lausekkeet tietorakenteiden käsittelyssä C#-kielessä: satunnaistettu vertailukoe

Title in English: Lambda expressions in handling data structures in C#: a randomized controlled trial

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmistotekniikka

Sivumäärä: 67+10

Tiivistelmä: Ohjelmointikielten eri ominaisuuksia on historiallisesti tutkittu vähän ihmislähtöisestä ja myös opetuksellisesta näkökulmasta. Työssä verrattiin lambda-lausekkeita silmukoihin C#-kielessä tietorakenteiden käsittelyssä. Tutkimus toteutettiin jakamalla yliopiston ensimmäisen ohjelmointikurssin opiskelijat (n = 187) arpomalla kahteen ryhmään. Yksi ryhmä suoritti tutkimusta varten laaditut tehtävät lambda-lausekkeilla, ja toinen ryhmä suoritti tehtävät silmukoilla. Lambda-lausekkeita hyödynnettiin C#-kielen listan metodien kanssa. Tehtäviä oli yhteensä neljä. Osallistujien tehtävistä antamat aika-arviot ja vaikeustasoarviot sekä osallistujien tehtävistä saamat pistemäärät ja tehtävien yrityskerrat eivät sisältäneet tilastollisesti merkitseviä eroja. Poikkeuksena oli viimeinen tehtävä, johon lambda-lausekkeita käyttäneet opiskelijat tarvitsivat enemmän yrityskertoja. Laadullisessa analyysissa lambda-lausekkeet saivat opiskelijoilta kehuja, mikä yhdistettynä tilastollisen analyysin tuloksiin voi antaa perusteita lambda-lausekkeiden opettamiseen alkutason ohjelmointikursseilla sekä niiden hyödyntämiseen koodia kirjoitettaessa.

Avainsanat: funktionaalinen ohjelmointi, iteraatio, lambda-lauseke, ohjelmoinnin opetus, ohjelmointi, ohjelmointikielten suunnittelu, ohjelmointikielten tutkimus, satunnaiskoe, silmukka

Abstract: Features of programming language have historically been researched poorly from a human factors and pedagogical perspective. In the study, lambda expressions were compared to loops within the context of handling data structures in C#. The study was performed by dividing CS1 students (n = 187) into two groups by random assignment. One group performed tasks created for the study by using lambda expressions, while the other group performed the tasks by using loops. Lambda expressions were used with methods of the C# list. There were four tasks in total. The participants' given assessments of task times and difficulty as well as their points and task attempts had no statistically significant differences, aside from the final task, which took more attempts to complete for students in the lambda group. In the qualitative analysis, lambda expressions received praise from students. Combined with the results of the statistical analysis, the positive response could make using lambda expressions in code and teaching them on early programming courses a justifiable choice.

Keywords: functional programming, iteration, lambda expression, teaching of programming, programming, programming language design, programming language research, randomized trial, loop

Kuviot

Kuvio 1. Tehtävä TIM-järjestelmässä koodikomponenttina 33
Kuvio 2. Esimerkki lambda-lausekkeen opettamista varten tehdystä materiaalista. Koko materiaali on sisällytetty liitteeseen B 35
Kuvio 3. Esimerkki luentomonisteen silmukoita koskevasta osiosta 36
Kuvio 4. Opiskelijoiden koetehtävistä saamien pisteiden jakauma 42
Kuvio 5. Opiskelijoiden tehtävistä antama vaikeustasoarvio 43
Kuvio 6. Keskimääräinen opiskelijoiden antama aika-arvio vaikeusarvion funktiona. Käyrä näyttää aika-arvion keskiarvon eri vaikeusarvion antaneiden kesken. Mitä vaikeammaksi opiskelijat kokivat tehtävät, sitä enemmän aikaa heillä kului tehtävien tekemiseen 44
Kuvio 7. Keskimääräinen opiskelijoiden koetehtävistä saama kokonaispistemäärä vaikeusarvion funktiona. Käyrä näyttää pistemäärän keskiarvon eri vaikeusarvion antaneiden kesken 45
Kuvio 8. Aika- ja vaikeusarvioille tehty normaalijakautuvuustesti 46
Kuvio 9. Aika- ja vaikeustasoarvioiden erojen tilastolliselle merkitsevyydelle tehdyt testit 46
Kuvio 10. Mann-Whitneyn U-testi ryhmien tehtävien yrityskertojen erojen tilastollisesta merkitsevyydestä. Tehtävässä 4 (värin vaihto) on ainoa tilastollisesti merkitsevä ero ryhmien välillä (sig. < 0.05) 47
Kuvio 11. Osallistujien tehtävän 4 yrityskerrat. Lambdaryhmässä oli suuri määrä opiskelijoita, joilla meni tehtävään yli 9 yrityskertaa 48
Kuvio 12. Speamanin rho -menetelmällä tehty korrelaatioanalyysi 49
Kuvio 13. Kendallin tau -menetelmällä tehty korrelaatioanalyysi 50

Taulukot

Taulukko 1. Jyväskylän yliopiston kevään 2020 Ohjelmointi 1 -kurssin luentojen
sisältö viikoittain 31
Taulukko 2. Tutkimusaineiston keskeiset muuttujat sekä niiden keskiarvot ja keskihajonnat. Tehtävien maksimipistemäärät löytyvät luvun 4.4 tehtäväkuvauksista 41

Sisällys

1 JOHDANTO 1

2 OHJELMOINTIKIELTEN TUTKIMUS JA LAMBDA-LAUSEKKEET 3
2.1 Aiempi tutkimus ja motivaatio 3
2.2 Tutkittava ominaisuus 5
2.3 Lambda-laskenta ja laskennan mallit 6
2.4 Lambda-lausekkeet ja käyttötarkoitukset 8
2.5 Lambda-lausekkeuden toteutus C#-kielessä 11

3 OHJELMOINNIN OPETUKSEN TUTKIMUS 14
3.1 Paradigmat käsitteenä 14
3.2 Paradigmat opetuksessa ja tutkimuksessa 15
3.3 Paradigmat ja tietorakenteiden käsittely 18
3.4 Ongelmanratkaisu 22

4 TUTKIMUS 26
4.1 Tutkimusmenetelmä 26
4.2 Konteksti 30
4.3 Aineiston keruu 33
4.4 Materiaali ja tehtävät 34

5 TULOKSET 41
5.1 Tilastollinen analyysi 41
5.2 Laadullinen analyysi 51
5.3 Vääristävät tekijät 53

6 JOHTOPÄÄTÖKSET JA POHDINTA 55

LÄHTEET 57

LIITTEET 63
A Tietosuojaseloste 63
B Lambda-lausekkeiden opettamista varten tehty materiaali 66

1 Johdanto

Ohjelmointikieliä on käytetty ja kehitetty jo 50-luvulta lähtien. Huolimatta pitkästä aikajänteestä ja ohjelmointikielten merkityksellisyydestä, ohjelmointikieliä on useiden lähteiden mukaan tutkittu heikosti ihmislähtöisestä näkökulmasta. Esimerkiksi tietellisesti laadukkaita satunnaiskokeita erilaisten kielten suunnittelijoiden väitteiden todistamiseksi on tehty vain vähän (Kaijanaho 2015). Myersin ym. (2016) mukaan edes suosittuihin ohjelmointikieliin, esimerkiksi Java ja C++, edellisen vuosikymmenen aikana tehtyjen muutosten hyödyllisyyttä ihmislähtöisestä näkökulmasta ei ole todistettu tieteellisesti. Kun ottaa huomioon ohjelmistojen merkityksen yhteiskunnassa ja ohjelmistomarkkinan satojen miljardien dollarien arvon (Statista 2019), on perusteltua katsoa, että ohjelmointikielten yksittäisilläkin ominaisuuksilla voi olla huomattavia taloudellisia vaikutuksia. Myöskään ohjelmoinnin opetukseen liittyvä tutkimus ei ole pysynyt ohjelmointikielten nopean kehityksen mukana, ja suuri osa ohjelmointikielten ominaisuuksista on tutkimatta opetuksellisesta näkökulmasta.

Ohjelmointikielten merkityksellisyyden ja niiden ihmislähtöisen tutkimuksen vähäisyyden motivoimana tämän tutkimuksen tarkoituksena on selvittää, ovatko lambda-lausekkeet hyödyllisiä C#-kielessä tietorakenteiden käsittelyssä verrattuna silmukoihin. Mahdollista hyödyllisyyttä tarkastellaan tehtävien koetun vaikeuden, toteutukseen käytetyn ajan ja toteutuksessa tehtyjen virheiden näkökulmasta. Uesbeckin ym. (2016) mukaan C++-kielessä lambda-lausekkeet lisäävät toteuttamisessa tehtyjen virheiden määrää verrattuna tietorakenteiden käsittelyyn iteraattoreilla. Lambda-lausekkeiden toteutus C#-kielessä on kuitenkin erilainen kuin C++-kielessä, minkä vuoksi tutkimus antaa uutta tietoa aiheesta. Aiempi tutkimus myös antaa tuloksen, johon tutkimuksen tuloksia voidaan verrata.

Tutkimusmenetelmänä on satunnaistettu vertailukoe (englanniksi *randomized controlled trial*, lyhennettynä *RCT*). Tutkimuksen aineisto kerättiin Jyväskylän yliopiston Ohjelmointi 1 (*CS1*) -kurssin yhteydessä kurssin osallistujien tekemien aihetta käsittelevien tehtävien vastauksista.

Tutkimuksen toisessa luvussa kerrotaan enemmän ohjelmointikielten tutkimuksesta ja lambda-lausekkeista. Kolmannessa luvussa käsitellään tutkimusta ohjelmoinnin opetuksen näkökulmasta. Neljännessä luvussa kuvataan tarkka tutkimusmenetelmä ja tutkimuksen kulku, jonka jälkeen viidennessä luvussa esitellään tulokset. Lopuksi kuudennessa luvussa tehdään pohdinta ja johtopäätökset tulosten pohjalta.

2 Ohjelmointikielten tutkimus ja lambda-lausekkeet

Tässä luvussa käsitellään ohjelmointikielten ihmislähtöistä tutkimusta ja perustellaan, miksi tutkimukseen valittiin juuri lambda-lausekkeet. Lisäksi selvitetään lambda-lausekkeiden historiaa, käyttötarkoituksia ja nykyistä toteutusta.

2.1 Aiempi tutkimus ja motivaatio

Lambda-lausekkeiden tutkiminen liittyy ohjelmointikielten tutkimukseen. Koska tutkimukseen liittyy myös ohjelmointikurssilla tehty koe, tutkimus liittyy myös ohjelmoinnin opetuksen tutkimukseen. Aiemman tutkimuksen kartoittamiseksi tehtiin kirjallisuuskatsaus, jossa katsottiin läpi julkaistuja tutkimuksia eri ohjelmointikielten suunnitteluun ja opetukseen liittyvistä konferensseista. Läpi käytävät konferenssit valittiin selaamalla ACM:n (*Association for Computing Machinery*) konferensseja aihepiireittäin ja valitsemalla ne, joiden nimi ja kuvaus olivat lähellä tutkimuksen aihepiiriä. Lisäksi mukaan otettiin konferenssit, joissa tutkija tiesi jo valmiiksi olevan aihepiirin kannalta olennaisia tutkimuksia. Valituista konferensseista käytiin läpi artikkeleiden nimet ja tiivistelmät, ja tarkempaan tutkimukseen otettiin kaikki artikkelit, jotka vaikuttivat liittyvän tutkimuksen aihepiiriin.

Kokonaisuutena kirjallisuuskartoituksessa tutkittiin opetukseen liittyen seuraavat konferenssit vuosilta 2017–2019:

- Special Interest Group of Computer Science Education (SIGSCE)
- Innovation and Technology in Computer Science Education (ITiCSE)
- International Computing Education Research (ICER)

Ohjelmointikielten tutkimukseen liittyen tutkittiin läpi International Conference on Software Engineering (ICSE)- ja Systems, Programming, and Applications (SPLASH) -konferenssien seuraavat alakonferenssit vuodelta 2019:

- Cooperative and Human Aspects on Software Engineering (CHASE)
- Formal Methods in Software Engineering (FormaliSE)

- Programming Language Design and Implementation (PLDI)
- International Conference on Global Software Engineering (ICGSE)
- Program Comprehension (ICPC)
- International Conference on Software Engineering (ICSE)
- Software Engineering Research and Industrial Practice (SER&IP, SERIP)
- Programming based on Actors, Agents, and Decentralized Control (AGERE)
- Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems (AI-SEPS)
- Dynamic Languages (DLS)
- New Ideas, New Paradigms, and Reflections on Programming and Software (ONWARD)
- Evaluation and Usability of Programming Languages and Tools
- Programming Experience (PX)
- Reactive and Event-based Languages & Systems (REBLS)
- Software Language Engineering (SLE)
- Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) / SPLASH Companion

Lisäksi kirjallisuuskatsauksessa tehtiin hakuja Google Scholar -palvelussa aihepiiriin liittyvien hakusanojen yhdistelmillä. Löydettyihin, lähteiksi kelpuutetuihin artikkeleihin on sovellettu myös *backward search* ja *forward search* -menetelmiä.

Tämän kirjallisuuskatsauksen perusteella lambda-lausekkeita ei ole tutkittu ihmislähtöisesti vuosina 2017–2019. Uesbeckin ym. (2016) mukaan niin ei ole kokeellisesti tehty myöskään aikaisemmin, lukuun ottamatta heidän tutkimustaan. Lambda-lausekkeiden teoriaa ja teknistä puolta käsitteleviä tutkimuksia löytyy, mutta niiden käyttöä käsitteleviä tutkimuksia ei. Opetuksellisesta näkökulmasta lambda-lausekkeiden käytöstä ohjelmointikursseilla ei myöskään löytynyt yhtään tutkimusta. Yleisemmin ohjelmointiparadigmoja vertailevia tutkimuksia tehdään myös vain vähän. Erityisesti funktionaalista ohjelmointiparadigmaa verrataan muihin paradigmoihin harvoin: Luxton-Reilly ym. (2018) löysivät opetuksen tutkimusta kartoittavassa kirjallisuuskatsauksessaan vuosilta 2003–2017 vain kaksi artikkelia, jotka

vertailivat funktionaalista ohjelmointia ja olio-ohjelmointia.

Ohjelmointikielten suunnitteluun liittyviä ihmislähtöisiä empiirisiä tutkimuksia tehdään yleensäkin vähän. Markstrumin (2010) mukaan ohjelmointikielten suunnittelijat eivät historiallisesti ole pyrkineet todistamaan väitteitään, vaan ovat luottaneet siihen, että kielten arvostelijat ovat sympaattisia ja uskovat väitteiden pätevän. Ihmislähtöisen tutkimuksen puutetta kritisoivat myös Hanenberg (2010),
Myers ym. (2016) ja Stefik ja Hanenberg (2014). Lisäksi ihmislähtöisissä tutkimuksissa käytetyt tutkimusmenetelmät ovat usein epätarkoituksenmukaisia tai virheellisiä (Stefik ja Hanenberg 2017; Ko, Latoza ja Burnett 2015). Tutkimuksen yhtenä tarkoituksena on osaltaan täydentää ohjelmointikielten ihmislähtöisen suunnittelun tutkimuksen aukkoa.

2.2 Tutkittava ominaisuus

Tutkittavaa ominaisuutta valitessa pohdittiin useita vaihtoehtoja. Suuren otoskoon varmistamiseksi tutkimus haluttiin tehdä Jyväskylän yliopiston ohjelmointikurssin yhteydessä. Yliopiston suurista ohjelmointikursseista yksi käyttää kielenään C#:ia ja toinen Javaa. Tutkittavia ominaisuuksia siten haettiin näistä kielistä. Lopulta tutkimus päädyttiin tekemään C#-kieltä käyttävällä kurssilla, koska se oli tutkijalle tutumpi. C#-kieleen oli edellisissä versioissa lisätty paljon mahdollisia tutkittavia ominaisuuksia, esimerkiksi (Microsoft 2019c, 2019d):

- *pattern matching* `switch`-lauseissa
- yksittäisestä lauseesta koostuvat aliohjelmat (*expression bodies*)
- Tuple-tyypit
- muuttujien luominen `using`-lauseen kanssa ilman näkyvyysalueen luomista
- viitemuuttujat, jotka eivät voi saada `null`-arvoa
- taulukoiden alueellinen indeksointi
- *null-coalescing* -operaattori

Yliopiston ohjelmointi 1 -kurssin yhteydessä opiskelijat toteuttavat pääosin yksinkertaisia algoritmeja yksittäisinä aliohjelmina tai funktioina. Monia näistä ominai-

suuksista olisi ollut vaikea hyödyntää kurssilla, mikä olisi tehnyt niiden tutkimisesta kurssin yhteydessä vaikeaa.

Lambda-lausekkeita ohjelmointikurssilla sen sijaan käytetään vähäisessä määrin. Lambda-lausekkeita tutkimalla tutkimuksen tuloksia pystyi myös vertaamaan aiempaan tutkimukseen paremmin, sillä tuoreita ominaisuuksia ei todennäköisesti ole tutkittu lainkaan, mutta lambda-lausekkeita on: Uesbeck ym. (2016) tutkivat lambda-lausekkeiden hyödyllisyyttä C++-kielen yhteydessä ja totesivat lambda-lausekkeiden käytön sekä hidastavan tehtävien tekoa että vaikeuttavan tehtäviä. Vertailun ja tutkimustulosten toistamisen lisäksi C#-kielen lambda-lausekkeiden syntaksi on hieman erilainen kuin C++-kielessä, joten tutkimus tuottaa myös uutta tietoa. Lambda-lausekkeiden tutkimiseen päädyttiin siten sen vuoksi, että ne sopivat pidettyyn ohjelmointikurssiin ja tutkimuksen tuloksia olisi helppo verrata aiempaan tutkimukseen.

2.3 Lambda-laskenta ja laskennan mallit

Alkujaan käsite *lambda-lauseke* on määritelty 1930-luvulla Alonzo Churchin toimesta osana lambda-laskentaa (Church 1932). Lambda-laskenta luotiin matemaattisen logiikan formaaliksi järjestelmäksi, jolla voidaan kuvata laskentaa funktioiden abstraktioiden (engl. *abstraction*) ja soveltamisen (engl. *application*) kautta. Lambda-laskennan määritelmään kuuluu, että sen kautta voidaan laskea kaikki mahdolliset tietokoneella laskettavat (engl. *computable*) tulokset ja ratkaista päätettävät ongelmat (Barendregt ja Barendsen 1984).

Samaan aikaan 1930-luvulla Alan Turing kehitti nykyisin Turingin koneena tunnetun laskennan mallin. Toisin kuin funktioihin perustuva lambda-laskenta, Turingin koneen toiminta pohjautuu tilamuutoksiin ja merkkejä sisältävään nauhaan. Nauhan yksittäisiä merkkejä lukemalla, koneen tilan muuttamisella luetun merkin perusteella ja nauhalla oleviä merkkejä muokkaamalla voi laskea minkä tahansa tietokoneella laskettavan tuloksen (Turing 1936). Vaikka nykyiset tietokoneet pystyvät tekemään kerralla paljon monimutkaisempia operaatioita, ja toisin kuin käsit-

teisellä Turingin koneella, fyysisellä tietokoneella on myös rajalliset resurssit, pohjimmiltaan nykyisissä tietokoneissa käytetty laskennan malli perustuu Turingin koneeseen: suoritin tekee operaatioita yksi kerrallaan lukemiensa käskyjen pohjalta ja samalla muokkaa muistissa tai rekistereissä olevaa dataa päästäkseen haluttuun lopputulokseen.

Ajoituksesta huolimatta Turing ja Church kehittivät molemmat omat laskennan teoriansa itsenäisesti (Barendregt ja Barendsen 1984). Vuonna 1937 Turing todisti, että lambda-laskenta ja Turingin koneet pystyvät laskemaan samat tulokset, ja siten mallit ovat laskennallisesti yhtä vahvoja (Turing 1937; Barendregt ja Barendsen 1984). Nämä mallit kuitenkin poikkeavat lähtökohdiltaan ja muilta ominaisuuksiltaan niin voimakkaasti, että tulosten laskeminen niiden avulla on merkittävästi erilainen prosessi. Näiden mallien pohjalta onkin kehittynyt erilaisia ohjelmointikieliä. Imperatiiviset ohjelmointikielet ja monet alkujaan imperatiivisiin kieliin perustuvat olio-ohjelmointikielet noudattavat Turingin koneen laskentamallia, kun taas funktionaaliset ohjelmointikielet on rakennettu lambda-laskennan pohjalta
(Barendregt ja Barendsen 1984).

Alkujaan Turingin koneen laskentamalliin perustuvia tietokoneita ohjelmoitiin syöttämällä tietokoneelle suoraan konekieltä (engl. *machine code*) tai kääntämällä symbolista konekieltä (engl. *assembly language*). Ensimmäiset korkean tason ohjelmointikielet, jotka eivät olleet tarkoitettu vain yhdelle tietylle tietokoneelle, kehitettiin 1950-luvulla (Wexelblat 2014). Näistä kaksi merkittävää kieltä, FORTRAN ja Lisp, ovat nykyään edelleen käytössä. Ennen FORTRANia käytössä oli erilaisia automatisointijärjestelmiä ohjelmoinnin helpottamiseksi, mutta ongelmana oli näiden työkalujen generoiman koodin hitaus: FORTRANin suunnittelijoiden mukaan ohjelmoijat olivat 1950-luvun alussa vakuuttuneita siitä, että suoritusajaltaan nopeiden ohjelmien tekemistä ei voisi automatisoida. FORTRAN kehitettiinkin erityisesti käännettyjen ohjelmien suorituskyky mielessä, ja itse kielen suunnittelua pidettiin helppona ja toissijaisena tehtävänä (Backus 1978). FORTRANin laskentamalli pohjautui Turingin koneeseen.

Lisp kehitettiin vuosi FORTRANin jälkeen. Ideana oli tehdä laskentaa numeroiden

sijasta symbolisilla lausekkeilla sekä listoilla ja koostettavilla, rekursiota tukevilla funktioilla muistin tilaa muokkaavien komentojen sijaan (McCarthy 1978). Funktioita haluttiin myös käyttää funktioiden argumentteina, mikä inspiroi Lispin suunnittelijoita ottamaan funktioiden notaation Churchin lambda-laskennasta. Aluksi Lispiä suunniteltiin sisällytettäväksi FORTRANiin, mutta laskennan malli osoittautui niin erilaiseksi, että erillisen kielen tekeminen todettiin helpommaksi (McCarthy 1978). Lispin katsotaan olevan ensimmäinen laajamittaisesti käytetty funktionaalinen ohjelmointikieli.

FORTRANista ja Lispistä lähtien laskentamallit ovat siirtyneet eteenpäin uusiin ohjelmointikieliin aina nykyisin suosituimpiin kieliin asti (Lévénez 2021; Watt ja Wong 1990, 2012). Vaikka perinteisesti tietyt kielet ovat pitäytyneet tietyssä laskentamallissa ja siihen yleensä liitetyssä syntaksissa, erityisesti edellisen vuosikymmenen aikana lambda-laskennasta johdettavia ominaisuuksia on lisätty myös perinteisesti imperatiivisiin ja oliosuuntautuneisiin kieliin. Siten esimerkiksi C#-kielessä, C++:ssa ja Javassa monia tuloksia voi nykyään laskea sekä silmukoiden, iteraattoreiden että lambda-lausekkeiden avulla. Lambda-lausekkeet eivät muuta näiden kielten laskentamallia, mutta ne muokkaavat kieltä syntaksin osalta lähemmäksi lambda-laskentaa. Tätä käsitellään lisää luvussa 2.4.

Lambda-laskennan ja Turingin koneiden lisäksi on kehitetty myös muita laskennan malleja. Esimerkiksi Turing itse määritteli Turingin koneiden eli alkujaan a-koneiden (engl. *a-machines, automatic machines*) lisäksi c-koneet (engl. *c-machines, choice machines*) ja u-koneet (engl. *u-machines, unorganized machines*), mutta Turingin koneiden merkittävyyden vuoksi näitä malleja ei ikinä tarkasteltu enempää (Wegner ja Goldin 2003).

2.4 Lambda-lausekkeet ja käyttötarkoitukset

Nykyaikaisten ohjelmointikielten kontekstissa lambda-lausekkeen merkitys on lähimpänä lambda-laskennan abstraktiota (engl. *abstraction*). Käytännössä lambda-lauseke on yleisimmin funktio tai aliohjelma, jolla ei ole nimeä ja joka määritel-

lään ohjelmakoodissa funktion käytön yhteydessä, kun tavalliset funktiot tai aliohjelmat määritellään erillään. Ohjelmointikielestä riippuen lambda-lausekkeita voidaan myös viedä parametreina toisille funktioille tai aliohjelmille, ja ne saattavat voida myös viitata paikallisen näkyvyysalueensa muuttujiin (Uesbeck ym. 2016).

Lambda-lausekkeet ovat kauan olleet yksi funktio-ohjelmoinnin ydinasioista (Mazinanian ym. 2017). Funktio-ohjelmoinnin ominaisuuksia on ajan myötä sisällytetty myös useisiin suosittuihin imperatiivisiin ja olio-ohjelmointikieliin. C#-kieleen lambda-lausekkeet sisällytettiin version 3.0 myötä vuonna 2007, C++-kielen standardiin vuonna 2011 ja Javaan vuonna 2014 version 8 myötä. Uesbeckin ym. (2016) mukaan lambda-lausekkeiden käyttöä on tutkittu vähän, joten niiden lisääminen olio-ohjelmointikieliin on voinut merkitä mahdollisesti kalliita muutoksia kyseenalaisin hyödyin.

Virallisia lähteitä lambda-lausekkeiden käyttötarkoituksista ja hyödyllisyydestä on löydettävissä rajallisesti. Oraclen (2014b) mukaan lambda-lausekkeet Javassa tarjoavat selvän ja ytimekkään ilmaisun yhden metodin rajapinnoille, ja lisäksi lambda-lausekkeet helpottavat tietorakenteiden käsittelyn yhteydessä rakenteen iterointia, alkioiden suodattamista sekä datan hakemista. C#-kielen osalta virallisista lähteistä ei vaikuta löytyvän perusteluja lambda-lausekkeiden lisäämiselle kieleen, mutta Internetistä on löydettävissä runsaasti lambda-lausekkeiden potentiaalisia hyötyjä käsitteleviä artikkeleita. Lambda-lausekkeet tarjoavat C#-kielessä lyhyen ja ytimekkään tavan ilmaista (nimettömiä) funktioita arvoina, joita voi antaa parametrina aliohjelmille. Microsoftin C++-kielen dokumentaatiosivulla (Microsoft 2019b) lambda-lausekkeet mainitaan kätevänä keinona nimettömän funktio-olion luomiseen samassa paikassa, missä sitä kutsutaankin. Lambda-lausekkeiden mainitaan tyypillisesti sisältävän muutaman rivin koodia, jotka viedään parametrina algoritmille tai asynkroniselle operaatiolle. Yhteistä useille lähteille on lambda-lausekkeilla tehtyjen ratkaisujen syntaktisen ytimekkyyden – eli vähäisen merkkimäärän – mainitseminen hyvänä puolena verrattuna ratkaisuihin, jotka on toteutettu käyttämättä lambda-lausekkeita.

Nielebock, Heumüller ja Ortmeier (2019) käsittelevät lambda-lausekkeen mahdolli-

sia hyötyjä osin funktio-ohjelmoinnin taustaan vedoten. Lambda-lausekkeiden sanotaan olevan yksinkertaisia ja ytimekkäitä, sekä niillä väitetään olevan helpompi ohjelmoida aiheuttamatta sivuvaikutuksia. Tämän myötä lambda-lausekkeet helpottavat heidän mukaansa myös rinnakkaistuvan (engl. *concurrent*) koodin tuottamista. Helpottamisen määrän mittaamisen he toteavat olevan vaikeaa, mutta he selvittivät asiaa tutkimalla C#:lla, Javalla ja C++:lla kirjoitettuja GitHub-projekteja. Projekteja käytiin läpi 1000 kappaletta jokaista kieltä kohti. Tuloksena he saivat, että suurin osa ohjelmoijista ei hyödynnä lambda-lausekkeita rinnakkaistuvan koodin kirjoittamisessa. Eri kielten välillä erot olivat tosin suuria. C#-kielisistä projekteista enemmistö hyödynsi lambda-lausekkeita, ja C#-kieliset projektit hyödynsivät lambda-lausekkeita selvästi yleisemmin kuin C++-kieliset projektit, jotka edelleen hyödynsivät lambda-lausekkeita selvästi yleisemmin kuin Javalla kirjoitetut projektit. Tutkijat arvelivat syynä olevan C#-kielen kehitysympäristöjen paremmat työkalut koodin uudelleenjärjestelyyn (engl. *refactoring*). Muitakin syitä tosin on haettavissa: yhtenä mahdollisena syynä voisi pitää sitä, että lambda-lausekkeet ovat olleet C#-kielessä selvästi kauemmin kuin C++:ssa, jossa ne ovat edelleen olleet kauemmin kuin Javassa. Tätä syytä tukee Mazinanian ym. (2017) toteama havainto, että vuodesta 2015 vuoteen 2016 lambda-lausekkeiden käyttö Java-koodissa yleistyi 204 %.

Mazinanian ym. (2017) tutkivat lambda-lausekkeiden käyttöä Javalla kirjoitetuissa avoimen lähdekoodin projekteissa ja vahvististivat tutkimuksessaan joidenkin kehittäjien käyttävän lambda-lausekkeita koodinsa lyhentämiseen ja syntaksin yksinkertaistamiseen. Lisäksi lambda-lausekkeiden käytön hyötynä nähtiin koodin asettaminen siihen paikkaan lähdekoodia, jossa koodia kutsutaankin, minkä koetaan parantavan koodin ylläpidettävyyttä. Lambda-lausekkeita käytettiin usein ohjelman käyttäytymisen parametrisointiin, minkä voidaan katsoa täsmäävän Oraclen (2014b) käyttötarkoitukseen tietorakenteiden käsittelyn helpottamisessa. Esimerkiksi sekä Javan että C#:n standardikirjastoissa on olemassa metodit, joilla listan jokaiselle alkiolle voi suorittaa tietyn, parametrisoidun toiminnon (Oracle 2014a; Microsoft 2020). Mazinanian ym. (2017) löysivät lambda-lausekkeille yleisenä käyttötarkoituksena myös tietyn aliohjelman suorittamisen ajoittamisen jonkin tapahtuman yhteyteen takaisinkutsuna (engl. *callback*). Edellä mainittujen käyttötarkoitusten lisäk-

si lambda-lausekkeet soveltuvat hyvin toisteisen koodin vähentämiseen parametrisoimalla toisteisessa koodissa esiintyviä eroavaisuuksia (Tsantalis, Mazinanian ja Rostami 2017).

Viitattujen lähteiden perusteella voi todeta, että lambda-lausekkeet on suunniteltu merkkimäärältään lyhyeksi ja syntaktisesti yksinkertaiseksi tavaksi parametrisoida ohjelman käyttäytymistä. Lambda-lausekkeiden mahdollistaman funktioiden kirjoittamisen niiden kutsumisen yhteyteen nähdään usein tekevän koodista helpompaa lukea. Yksi yleinen sovellus lambda-lausekkeille on tietorakenteiden käsittely, jossa lambda-lausekkeita voi hyödyntää esimerkiksi muokatessa, suodattaessa ja hakiessa tietorakenteen sisältöä.

2.5 Lambda-lausekkeuden toteutus C#-kielessä

Koska tutkimus käsittelee lambda-lausekkeiden hyödyllisyyttä C#-kielessä, on perusteltua esitellä C#:n lambda-lausekkeiden syntaksi.

Microsoftin virallisessa dokumentaatiossa (Microsoft 2019a) lambda-lausekkeet jaetaan kahteen kategoriaan sen mukaan, sisältävätkö ne yhden vai useamman lauseen. Erona lausekkeissa on, että useamman lauseen sisältävät lambda-lausekkeet (engl. *statement lambdas*) sisältävät aaltosulut lauseiden ympärillä, ja yhden lauseen sisältävät lambda-lausekkeet (engl. *expression lambdas*) voivat myös palauttaa arvonsa ilman `return`-lauseen käyttöä. Yleisesti lambda-lausekkeet ovat seuraavaa muotoa:

```
(parametrit) => { <lauseet> }
```

Lambda-lausekkeissa funktion parametrit erotetaan funktion sisällöstä nuolella `=>`. Parametrit määritetään nuolen vasemmalla puolella suluissa. Parametrit erotetaan toisistaan pilkuilla. Halutessaan parametreihin voi kirjoittaa mukaan parametrien tyypit, mutta ne voi usein myös jättää kääntäjän pääteltäväksi. Mikäli parametreja on vain yksi ja parametrin tyyppiä ei erikseen määritellä, sulut parametrien ympäriltä voidaan jättää pois. Mikäli funktio ei ota parametreja vastaan lainkaan, laitetaan parametrien paikalle pelkät sulut `()` ilman sisältöä. Kaikki seuraavat ovat siten

muodoltaan oikeita lambda-lausekkeita:

```
(parametri1, parametri2, ...) => { <lauseet> }
parametri1 => { <lauseet> }
(<tyyppi> parametri1) => { <lauseet> }
() => { <lauseet> }
```

Funktion sisältö tulee nuolen => oikealle puolelle. Mikäli sisältöön kuuluu vain yksi lause, sisällön ympärille ei tarvitse aaltosulkuja. Mikäli funktion sisällön lauseesta tällöin palautuu arvo, funktio palauttaa arvon ilman `return`-lauseen käyttöä. Useamman puolipisteeseen päättyvän lauseen sisältävä lambda-lauseke tarvitsee aaltosulut `{ }` sisältönsä ympärille.

Yksinkertaisimmillaan lambda-lauseke on siten muotoa `() => <lause>`. Esimerkiksi aliohjelman, joka tuottaa konsoliin rivinvaihdon, voi luoda ja tallentaa `line`-nimiseen muuttujaan seuraavanlaisella koodilla:

```
Action line = () => Console.WriteLine();
```

Funktion, joka ottaa parametrina kaksi kokonaislukua ja tarkistaa, onko ensimmäinen niistä suurempi kuin toinen, voi kirjoittaa seuraavasti: `(x, y) => x > y`. Tämänkin funktion voi tallentaa muuttujaan, jolloin sitä voisi käyttää esimerkiksi seuraavasti:

```
1 Func<int, int, bool> tarkistaja = (x, y) => x > y;
2 int ensimmainenLuku = 3;
3 int toinenLuku = 5;
4 bool onkoSuurempi = tarkistaja(ensimmainenLuku, toinenLuku);
```

Lambda-lausekkeet soveltuvat erityisen hyvin hyödynnettäviksi monien C#:n listaluokan `List<T>` metodien kanssa. Seuraavassa esimerkissä merkkijonolistan `ForEach`-metodille annetaan parametrina funktio, jonka avulla tulostetaan jokaisesta alkiosta konsoliin tieto, että onko alkion merkkimäärä parillinen:

```
1 List<string> merkkijonot = new List<string>() { "appelsiini", "omena", "banaani" };
2 merkkijonot.ForEach(s =>
3 {
```

```
4       if (s.Length % 2 == 0)
5           Console.WriteLine($"'{s}': parillinen");
6       else
7           Console.WriteLine($"'{s}': pariton");
8   });
```

3 Ohjelmoinnin opetuksen tutkimus

Tässä luvussa käsitellään paradigma-käsitettä ja sen käyttöä ohjelmoinnin opetuksen tutkimuksessa. Paradigmoja käsitellään erityisesti tutkimuksessa tehdyn kokeen motivoinnin ja taustoittamisen näkökulmasta. Koska kokeen yhteydessä imperatiivisen CS1-kurssin opiskelijoille opetettiin lambda-lausekkeiden toiminta, tarkastellaan luvussa kirjallisuudesta löytyvää tukea funktionaalisen paradigman ja lambda-lausekkeiden opettamiselle sekä paradigmojen yhdistämiselle opetuksessa. Opettamisratkaisun lisätueksi luvussa tarkastellaan erilaisten ohjelmointiongelmien ratkaisua imperatiivisella paradigmalla verrattuna funktionaaliseen paradigmaan.

3.1 Paradigmat käsitteenä

Paradigmat ovat yleisesti käytetty nimitys erilaisten ohjelmointikielten luokitteluun ja ryhmittelyyn. Käsite oli jo aiemmin olemassa muilla tieteenaloilla, mutta ohjelmointiin sen toi Floyd (2007) vuonna 1979. Yleisyydestään ja historiallisesta merkityksestään huolimatta *paradigma* käsitteenä ei ole kirjallisuudessa selkeästi ja yksikäsitteisesti määritelty, ja käsitteen mielekkyyttä onkin osittain tämän johdosta kyseenalaistettu (Krishnamurthi ja Fisler 2019). Krishnamurthi ja Fisler (2019) toteavat paradigmojen olevan ryhmiä, jotka erottavat tietynlaisia ohjelmointikieliä toisenlaisista ohjelmointikielistä yleensä tietynlaista käyttäytymistä ilmentävien ominaisuuksien perusteella. Van Roy ym. (2009) taas määrittävät paradigman matemaattiseen teoriaan tai periaatteiden yhtenäiseen joukkoon pohjautuvaksi tietokoneen ohjelmoinnin lähestymistavaksi, mutta eivät tarkemmin määrittele periaatteiden yhtenäisyyttä tai sen kriteerejä.

Määritelmällisen ongelman lisäksi paradigman käytössä käsitteenä voi nähdä useita käytännöllisiä ongelmia. Usein paradigmoista puhuttaessa viitataan sekä kielen syntaksiin että sillä kirjoitettujen ohjelmien käyttäytymiseen, vaikka toisaalta nämä ulottuvuudet usein myös erotetaan toisistaan. Esimerkiksi visuaalinen, lohkoihin

perustuva ohjelmointi nähdään joskus omana paradigmanaan, vaikka sitä voi myös pitää vain erilaisena tapana kirjoittaa jotain olemassa olevaa, tiettyyn paradigmaan kuuluvaa kieltä (Krishnamurthi ja Fisler 2019). Myös yksittäisen kielen nähdään joskus mahtuvan vain yhteen paradigmaan, mutta toisaalta monet kielet voivat myös sisältää useampaan paradigmaan liittyviä ominaisuuksia (Krishnamurthi ja Fisler 2019; Kaijanaho 2015). Esimerkiksi suurin osa olio-ohjelmointikielistä on myös imperatiivisia kieliä (Kaijanaho 2015). Lisäksi moniin perinteisesti imperatiivisiin olio-ohjelmointikieliin, kuten Javaan, on ajan myötä lisätty ominaisuuksia funktionaalisen paradigman pohjalta (Krishnamurthi ja Fisler 2019).

Myös paradigmojen määrästä on poikkeavia näkemyksiä. Yleisesti tunnistetaan ainakin imperatiivinen, funktionaalinen, oliosuuntautunut, ja looginen paradigma (Krishnamurthi ja Fisler 2019). Kaijanaho (2015) tunnistaa väitöskirjansa kannalta olennaisiksi myös rakenteellisen (engl. *structured*), proseduraalisen (engl. *procedural*) ja näkökulmasuuntautuneen (engl. *aspect-oriented*) paradigman. Van Roy ym. (2009) tunnistavat jopa 27 erilaista paradigmaa.

Paradigma-käsitteen ongelmista huolimatta kirjallisuudessa ei ole vakiintunut sille korvaavaa tai tarkemmin määriteltyä käsitettä, minkä vuoksi käsitettä käytetään myös tässä tutkimuksessa. Koska tutkimuksessa toteutettiin koe CS1-kurssilla, on olennaista tarkastella paradigma-käsitteen käyttöä opetuksen tutkimuksen yhteydessä. Tällöin on syytä kiinnittää huomiota lähinnä yleisimmin tunnustettuihin ja opetettuihin paradigmoihin, eli imperatiiviseen, funktionaaliseen ja oliosuuntautuneeseen paradigmaan.

3.2 Paradigmat opetuksessa ja tutkimuksessa

Davies, Polack-Wahl ja Anewalt (2011) toteuttivat kyselytutkimuksen alkutason ohjelmointikursseilla käytettyihin kieliin liittyen. Näitä kursseja kutsutaan yleesä CS0, CS1- ja CS2-kursseiksi, ja vaikka ne eroavat sisällöiltään huomattavasti eri oppilaitosten välillä, termien käytöstä on olemassa hatara kansanvälinen yhteisymmärrys. Davies, Polack-Wahl ja Anewalt (2011) määrittelivät CS1-kurssin ensimmäisek-

si pakolliseksi korkeakoulussa tarjotuksi ohjelmointikurssiksi, ja CS2:n tämän kurssin jatkokurssiksi. CS0 määriteltiin alkeiskurssiksi, jolla tutustutaan ohjelmointiin, mutta joka ei ole pakollinen tai edistä tutkintoa. Kyselyyn vastasi 27,5 % Yhdysvalloissa tietotekniikkaa pääaineenaan opettavista korkeakouluista. Vastanneista oppilaitoksista 40 % ei tarjonnut CS0-kurssia. Tulosten perusteella ohjelmoinnin johdantokursseilla on suosittu olio-ohjelmointikieliä ainakin vuodesta 2001 lähtien. Olio-ohjelmointikielet ovat tosin yleensä myös proseduraalisia kieliä (Kaijanaho 2015), ja osalla olio-ohjelmointikieliä opettavista kursseista kirjoitettiinkin proseduraalista koodia oliosuuntautuneen koodin sijaan. Tästä huolimatta oliosuuntautunut paradigma oli erittäin suosittu: vastanneista 65 % hyödynsi olioparadigmaa CS1-kurssilla, ja CS2-kurssilla osuus nousi 92 %:iin. Vain 6 % vastaajista ei hyödyntänyt olioparadigmaa lainkaan kummallakaan kurssilla. Tutkimuksen vastaajien kielivalintojen perusteella on mahdollista päätellä, että tämä 6 % jakautuu imperatiivisen ja funktionaalisen paradigman välillä, joskin tutkimus ei suoraan ota asiaan kantaa.

Mason ym. (2018a) myös vahvistivat oliokielten suosion alkutason ohjelmointikursseilla Australiassa ja Iso-Britanniassa, joskin totesivat 2010-luvulla Pythonin kasvattaneen suosiotaan merkittävästi, osittain Javan kustannuksella. Python koettiin hyödylliseksi erityisesti sen työkalujen saatavuuden ja helppokäyttöisyyden sekä pedagogisten hyötyjen vuoksi, kun taas Java koettiin hyödylliseksi erityisesti olioparadigman vuoksi. Tutkimuksessa ei suoraan kerrota tarkemmin Pythonin koetuista pedagogisista hyödyistä, joskin Python koettiin aloitteleville ohjelmoijille helpoksi kieleksi. Pythonin helppous kielenä voi tehdä kielen opettamisen yksinkertaiseksi, mikä voi jättää enemmän aikaa ohjelmoinnin muiden osa-alueiden, kuten algoritmisen ajattelun, opettamiseen. Toisaalta ristiriitaisesti Pythonia ei koettu muita kieliä hyödyllisemmäksi ohjelmoinnin käsitteiden (engl. *concepts*) opettamisessa. Australiassa, jossa Python oli suosittu, Python koettiin itse asiassa selvästi vähemmän hyödylliseksi ohjelmoinnin käsitteiden opettamisessa kuin esimerkiksi Java, C# tai Haskell (Mason ym. 2018b). Kun Iso-Britannian vastaajien antamaa arviota vaikeustasosta ja hyödyllisyydestä verrattiin toisiinsa, Pythonin hyödyllisyys ohjelmoinnin opettamisessa suhteutettuna kielen vaikeustasoon koettiin paremmaksi kuin muissa kielissä. Kokonaisuutena tutkimus kuitenkin jättää Pythonin pedagogiset hyödyt

epäselviksi. Vaikka oliosuuntautuneisuus koettiin Javan hyödyksi, oliosuuntautuneisuutta ei juurikaan koettu Pythonin vahvuudeksi (Mason ym. 2018b), mistä voisi päätellä Pythonia opettavien mahdollisesti käyttävän enemmän imperatiivista paradigmaa. Toisaalta myös Python mahdollistaa oliosuuntautuneen ja luokkia hyödyntävän koodin kirjoittamisen (Python Software Foundation 2020).

Luxton-Reilly ym. (2018) toteavat itse opetuksen lisäksi myös opetuksen tutkimuksen painottuvan olio-ohjelmointikieliin. Heidän mukaan kirjallisuuskatsauksen perusteella ei ole selvää, että yksikään paradigma olisi toista parempi ohjelmoinnin opettamiseen alkutason kursseilla. He myös pohtivat, että sopivimmasta paradigmasta, samoin kuin ohjelmointikielestä, voi olla mahdoton päästä yhteisymmärrykseen alan tutkimuksessa.

Kirjallisuudesta löytyy oliosuuntautuneen paradigman suosiosta huolimatta tukea myös erilaisten paradigmojen yhdistämiselle. Van Roy ym. (2009) kehottavat ohjelmoimaan useampaa paradigmaa käyttäen, sillä erilaiset paradigmat soveltuvat erilaisten ongelmien ratkaisemiseen. Dougherty ja Wonnacott (2005) kokivat useamman paradigman opettamisen yhdellä kielellä (C++) hyödylliseksi tavoitteessaan luoda opiskelijoille laaja pohja, josta erikoistua tietotekniikan eri osa-alueille. Kereki ja Adorjan (2020) suunnittelivat oppilaitoksensa CS1-kurssin uudelleen hyödyntäen imperatiivista ja oliosuuntautunutta paradigmaa sekä myös joitain funktionaalisen ohjelmoinnin elementtejä. Uudistuksen onnistumista mitattiin kysymällä opiskelijoiden mielipiteitä ja haastattelemalla kurssin opettajia. Vastanneista opiskelijoista yli 96 % arvioi kurssin hyväksi tai erinomaiseksi. Opiskelijoista 92 % arvioi myös oppineensa ohjelmointia vähintään hyvin. Opettajat taas totesivat opiskelijoiden aktiivisuuden kasvaneen uudistuksen myötä. Uudistus ei nostanut kurssin läpäisseiden opiskelijoiden osuutta, joskin Kereki ja Adorjan (2020) arvioivat osuuden nousevan kurssin jatkokehittämisen myötä. Kurssilla toisaalta uudistettiin myös paljon muuta kuin paradigmat, eikä useamman paradigman käyttämisen vaikutusta kurssin positiivisessa vastaanotossa tutkittu tarkemmin.

Zuhud, Rahman ja Ismail (2013) käsittelevät paradigmojen opettamisen tärkeyttä ja haasteita erillisenä ohjelmointikielten opettamisesta. Yhtenä paradigmojen opet-

tamisen haasteena he löysivät liiallisen keskittymisen ohjelmointikielten syntaksiin ja semantiikkaan. Uuden paradigman opettamisen jo tuttua kieltä käyttämällä, kuten tämän tutkimuksen kokeen tapauksessa, voi olettaa vähentävän kielen yksityiskohtiin kuluvaa aikaa. Toisaalta tutun kielen käyttäminen mahdollistaa ongelman ratkaisemisen joko kokonaan tai osittain jo tutulla paradigmalla, jolloin kokemus uudesta paradigmasta voi jäädä ohueksi.

3.3 Paradigmat ja tietorakenteiden käsittely

Tutkimuksessa tehdyssä kokeessa käsiteltiin tietorakenteita C#-kielessä lambda-lausekkeita käyttäen. Koe tehtiin CS1-kurssilla, jolla käytettiin imperatiivista paradigmaa. Siten on syytä tarkastella kirjallisuudesta perusteita lambda-lausekkeiden opettamiselle. Yleisesti tunnetuin ja myös kurssilla opetettu imperatiivinen vaihtoehto lambda-lausekkeiden käytölle tietorakenteiden käsittelyssä on käsittely silmukoiden avulla.

Qian ja Lehman (2017) tekivät kirjallisuuskatsauksen opiskelijoiden haasteista alkutason ohjelmointikursseilla. He löysivät useita silmukoihin liittyviä haasteita. Silmukoiden näkyvyysalueen (engl. *scope*) ja toisteisten, monta kertaa suoritettavien rivien hahmottaminen voi olla opiskelijoille haastavaa. Opiskelijat eivät välttämättä hahmota silmukan suorituskertojen määrää. Yleensä silmukoita myös on monenlaisia (esimerkiksi *for*- ja *while*-silmukat), ja opiskelijat usein uskovat, että yksi silmukkatyyppi on toisia parempi, vaikka ne on tarkoitettu ratkaisemaan erilaisia ongelmia. Opiskelijoilla on siten haasteita valita optimaalinen silmukkatyyppi tietyssä kontekstissa. Osin tähän voi vaikuttaa Chenin ym. (2007) tekemä huomio, että `do-while` -silmukat ovat aloitteleville ohjelmoijille `while` -silmukoita intuitiivisempia, vaikka kokeneet ohjelmoijat suosivat `while` -silmukoita. Silmukkatyypeistä `do-while` -silmukat tekevät ensin jonkin toimenpiteen, ja toimenpiteen tekemisen jälkeen tarkistavat, että onko tehty asia syytä toistaa; `while` -silmukat toimivat muuten samoin, mutta tarkistavat ehdon jo ennen toimenpiteen ensimmäistä suorituskertaa. Kun opiskelijat, joilla ei ole varsinaista ohjelmointikokemusta, suunnittelevat iteroivia algoritmeja, he yleensä miettivät tekevänsä toimenpiteen ensin ja

sitten tarkistavansa, tarvitseeko sitä toistaa (Chen ym. 2007). Funktioiden käytössä Qian ja Lehman (2017) totesivat haasteina parametrien välittämisen ja funktion paluuarvon sijoittamisen ymmärtämisen. Heidän tekemän haasteiden tunnistamisen perusteella voisi päätellä funktioiden ja siten lambda-lausekkeiden käytössä olevan vähemmän mahdollisia haasteita tai kohtia virheiden tekemiseen kuin silmukoiden käyttämisessä. Toisaalta he eivät erityisesti yrittäneet etsiä haasteita funktionaalisesta ohjelmoinnista, ja viitaten Fislerin, Krishnamurthin ja Siegmundin (2016) tutkimukseen, he toteavat funktionaalisten ohjelmoijien mahdollisesti törmäävän erilaisiin virheisiin kuin imperatiivisten tai oliosuuntautuneiden ohjelmoijien.

Bruce, Danyluk ja Murtagh (2005) argumentoivat rakenteellisen rekursion (esimerkiksi linkitetyn listan ja muiden rekursiivisten tietorakenteiden käytön) opettamisen ennen taulukoita vahvistavan opiskelijoiden ymmärrystä olio-ohjelmoinnista, ja tukevat argumenttiaan opiskelijoiden kokemuksella heidän oman oliosuuntautuneen CS1-kurssin vaikeustasosta. Opiskelijat kokivat kurssin helpommaksi, kun rekursio opetettiin tietorakenteiden avulla ja ennen taulukoita. Rekursiiviset rakenteet mahdollistavat rajapintojen hyödyntämisen taulukoita paremmin, kun rakenteita käsitellään metodeilla. Lisäksi rekursiiviset rakenteet rakennetaan luokkina, jolloin opiskelijat näkevät nämä rakenteet omina olioinaan. Rakenteiden kanssa toimitaan rajapintojen kautta ja niiden sisäinen toiminta ei näy luokan ulkopuolelle (Bruce, Danyluk ja Murtagh 2005), mitä pidetään yhtenä olio-ohjelmoinnin kulmakivistä. Mikäli taulukot opetettiin ensin, opiskelijat ennemmin käyttivät taulukoita ikään kuin suoraan kuin tekivät omia rakenneluokkiaan, jotka käyttivät taulukoita sisäisesti. Vaikka lambda-lausekkeet eivät suoraan liity rekursioon tai rakenteisiin, ne sopivat hyvin erillisinä luokkina toteutettujen rakenteiden käsittelyyn, kuten luvun 2.5 viimeisessä esimerkissä esitetään. Siten, huolimatta lambda-lausekkeiden taustasta funktionaalisen ohjelmoinnin paradigmassa, niitä on mahdollista hyödyntää olio-ohjelmoinnin opettamisessa erillisinä luokkina toteutettujen rakenteiden käsittelyssä.

Altadmri ja Brown (2015) tutkivat yli 250 000 opiskelijan tekemiä käännösvirheitä Blackbox-aineistosta (BlueJ 2020). Aineisto sisälsi tiedot yli 37 miljoonasta käännök-

sestä. Analysoidessaan aineistoa he tunnistivat 18 erilaista virhettä, ja tutkivat, mitä näistä virheistä oli tehty eri käännöskerroilla – sekä onnistuneilla että epäonnistuneilla. Jokaiselle virheelle annettiin tunnisteeksi erilaiset kirjaimet. Vaikka he eivät tarkemmin analysoineet virheiden kontekstia, virheiden kuvauksista on mahdollista päätellä tilanteita, joissa virheitä on mahdollista tai yleistä tehdä. Heidän tunnistamista virheistä seuraavien voi nähdä olevan yleisiä silmukoita tehdessä:

- **C**: Epätasaiset tai virheelliset sulut, esimerkiksi `while (a == 0]`, kun pitäisi olla `while (a == 0)`
- **E**: Puolipiste virheellisesti ehtolauseen tai silmukan määrittelyrivin lopussa, esimerkiksi `while (a < b);`
- **F**: Väärä erotinmerkki `for`-silmukassa, esimerkiksi `for (int i = 0, i < 6, i++)`, kun pitäisi olla `for (int i = 0; i < 6; i++)`
- **L**: Lukujen vertailuoperaattorin kirjoittaminen väärin, esimerkiksi `i =< 10`, kun pitäisi olla `i <= 10`

Näistä virheistä **F** ja **L** olivat harvinaisia, ja opiskelijoilla myös kesti vain vähän aikaa niiden korjaamiseen. **F** esiintyi 2 719 kertaa, ja sen korjausajan mediaani oli 36 sekuntia. **L** taas esiintyi 4 214 kertaa, ja sen korjaaminen kesti mediaanina 12 sekuntia. Altadmri ja Brown (2015) arvioivat opiskelijoiden oppivan näistä virheistä kerralla, minkä vuoksi opiskelijat eivät toista virhettä uudelleen. Näitä virheitä voi siten pitää kokonaiskuvan kannalta epäolennaisena. **C** oli 793 232 esiintymiskerrallaan kaikista tutkimuksessa tarkastelluista virheistä selvästi yleisin, mutta myös nopea korjata: mediaanina **C**:n korjaaminen kesti 17 sekuntia. **E** oli esiintyvyyden suhteen virheiden puolivälissä 49 375 esiintymisellä, ja sen korjaaminen vei keskimäärin melkein 7 minuuttia. Merkittävimpänä syynä **E**:n pitkään korjausaikaan Altadmri ja Brown (2015) pitävät sitä, että **E**:n kaltainen looginen virhe ei lähtökohtaisesti aiheuta käännösvirhettä, vaan ohjelma kääntyy ja käynnistyy. Siten ohjelmoija huomaa virheen vasta ohjelmaa ajaessa.

Jos Altadmrin ja Brownin määrittelemiä virheitä miettii lambda-lausekkeiden osalta, **E**-virhe olisi mahdollista tehdä ehtolauseessa tai silmukassa monilauseisen lamb-

da-lausekkeen sisällä. Erityisesti silmukoiden kirjoittamista lambda-lausekkeiden sisällä voi tosin pitää suhteellisen harvinaisena, joten lambda-lausekkeiden käytön silmukoiden sijasta voisi olettaa harvinaistavan **E**-virhettä. Koska sekä lambda-lausekkeissa että silmukoissa tarvitaan sulkuja, voi virheen **C** olettaa koskevan molempia tapoja vastaavalla tavalla. Altadmrin ja Brownin virheistä on myös tunnistettavissa kaksi virhettä, joiden voi nähdä tapahtuvan useammin lambda-lausekkeella tehdyssä rakenteiden käsittelyssä verrattuna silmukoilla tehtyyn rakenteiden käsittelyyn. Lambda-lausekkeiden syntaksi mahdollistaa tyyppien jättämisen pois niitä käytettäessä (ks. luku 2.5), jolloin tyyppivirheet **I** ja **Q** voivat esiintyä yleisemmin. **I**:ssä opiskelija on käyttänyt väärän tyyppistä argumenttia kutsuessaan aliohjelmaa, ja **Q**:ssa funktion paluuarvon tyyppi ei sovi yhteen sen muuttujan tyypin kanssa, johon funktion paluuarvoa yritetään sijoittaa. Erityisesti **I** on yleinen virhe, jonka korjaaminen on myös kestänyt mediaanina suhteellisen kauan, eli minuutin. **Q** on suhteellisen harvinainen, mutta sen korjaaminen on vienyt keskimäärin melkein kaksi minuuttia. On oletettavaa, että tutkimuksessa suurin osa **I**- ja **Q**-virheistä on tapahtunut muiden kuin lambda-lausekkeiden kontekstissa. Vastaavasti voi olettaa, että näiden virheiden yleisyyttä lambda-lausekkeiden opetuksessa voisi laskea kehottamalla opiskelijoita sisällyttämään lambda-lausekkeisiin parametrien tyypit. Mikäli lambda-lausekkeet eivät merkittävästi lisäisi **C**-, **I**- ja **Q**-virheiden määrää tai nämä virheet olisivat nopeampia korjata lambda-lausekkeiden yhteydessä, lambda-lausekkeiden oletettu **E**-virheiden pienempi esiintyvyys voisi tehdä lambda-lausekkeista silmukoita vähemmän virhealttiin vaihtoehdon rakenteiden läpikäynnille.

Sorva ja Vihavainen (2016) kokoavat artikkelissaan erilaisia argumentteja CS1-kurssilla käytettävien lähestymistapojen hyödyistä ja haitoista ja esittävät argumentit keskustelumuodossa. Silmukoiden osalta he käsittelevät `break`-lausetta. Viitaten Robertsin (1995) artikkeliin he toteavat, että oikein käytettynä `break`-lause voi selkeyttää silmukoiden yhteydessä käytettyä koodia ja vähentää sen toisteisuutta. Toisaalta he argumentoivat, että `break`-lauseen käyttö voi olla haitaksi ratkaistaessa monesta pienestä ongelmasta koostuvia isompia ongelmia, sillä `break`-lauseen käyttö kehottaa opiskelijoita yhdistämään monen aliongelman ratkaisun saman silmukan alle. Tämä voi vaikeuttaa ongelman ratkaisemista kokonaisuutena, sillä eri-

laisten ratkaisusuunnitelmien yhdistäminen on opiskelijoille haastavaa (Soloway 1986). Vertaillen mahdollisia ratkaisuja Solowayn (1986) määrittelemään Rainfall-ongelmaan, he tarjoavat yhden silmukan ratkaisun vaihtoehdoksi ratkaisua, jossa ongelma on jaettu pieniin osiin, jotka ratkaistaan pienillä, yksinkertaisilla funktioilla. Yhdistämällä näitä funktioita ja viemällä niitä parametreina sopiville standardikirjaston tietorakenteita käsitteleville funktioille, Rainfall-ongelman voi saada ratkaistua selkeämmän näköisellä koodilla. Tätäkin ratkaisua kritisoidaan esittelyn lisäksi, sillä Solowayn, Bonarin ja Ehrlichin (1983) mukaan opiskelijat yleensä suosivat ratkaisuja, jotka `break`-lauseen kaltaisesti mahdollistavat silmukan suorituksen lopettamisen silmukan keskellä. Lisäksi he viittaavat Greenin ja Petren (1996) artikkeliin, jossa he toteavat apufunktioiden muodossa erilaisten abstraktioiden rakentamisen olevan opiskelijoille vaikeaa. Sorva ja Vihavainen (2016) eivät lopulta päädy suosimaan kumpaakaan ratkaisua, mutta kokoavat useita argumentteja molempien puolesta ja vastaan. He myös toteavat anonyymien funktioiden menevän luultavasti liian pitkälle aloittelijoiden näkökulmasta, ilmeisesti syntaksin osalta. Lambda-lausekkeet C#-kielessä ovat tapa toteuttaa anonyymeja funktioita, minkä perusteella Sorva ja Vihavainen (2016) eivät erityisesti kannattaisi lambda-lausekkeiden opettamista CS1-kurssilla. Muilta osin lambda-lausekkeita hyödyntäen toteutettu ratkaisu Rainfall-ongelmaan voisi täsmätä Sorvan ja Vihavaisen (2016) esittelemään ongelman pilkkomisratkaisuun, ja siten sillä voi nähdä olevan vastaavat hyödyt ja haitat verrattuna silmukalla tehtyyn ratkaisuun.

3.4 Ongelmanratkaisu

Tutkimuksessa tehtiin koe, jossa verrattiin silmukoilla ja lambda-lausekkeilla tehtyjä ongelmien ratkaisuja toisiinsa. Tämän vuoksi katsotaan kirjallisuudesta, millaisia samankaltaisia kokeita on jo tehty, ja millaisia tuloksia niistä on saatu. Yleisesti ottaen kirjallisuudesta on löydettävissä vain vähän ongelmia, joiden ratkaisemista olisi tutkittu sekä funktionaalisella että imperatiivisella paradigmalla.

Ohjelmoinnin opetuksen tutkimuksessa yleisesti hyödynnetyn, alkujaan Solowayn (1986) määrittelemän Rainfall-ongelman, ratkaisu on paljon tutkittu kysymys. Al-

kuperäisessä ongelmassa ohjelmoijan tulee lukea lukuja jostain lähteestä ja laskea luvuista keskiarvoa, kunnes vastaan tulee luku 99999, jolloin käsittely lopetetaan. Ongelman määrittelemisen jälkeen siitä on kehitetty erilaisia variaatioita, joista esimerkiksi osassa negatiiviset luvut tulee jättää huomiotta, ja toisissa tulee laskea keskiarvon lisäksi myös maksimi (Seppälä ym. 2015). Yhteisiä haasteita variaatioille on kuitenkin lukujen lukeminen, yhteenlasku, lukujen määrän laskeminen, keskiarvon laskeminen ja lopputuloksen tulostaminen (Fisler 2014). Fisler (2014) tarkasteli Rainfall-ongelmaa funktionaalisella CS1-kurssilla useassa oppilaitoksessa. Tutkimuksessa opiskelijat suoriutuivat Rainfall-ongelman ratkaisemisessa merkittävästi paremmin kuin imperatiivisia kieliä käyttäneissä tutkimuksissa, joihin hän vertasi tuloksiaan. Seppälän ym. (2015) tekemässä tutkimuksessa kolmen imperatiivisen kurssin opiskelijat toisaalta suoriutuivat vielä paremmin. Myös opiskelijoiden suoriutuminen Lakasen, Lappalaisen ja Isomöttösen (2015) toteuttamalla imperatiivisella CS1-kurssilla oli likimain yhtä vahvaa kuin Fislerin tutkimuksessa. Tuloksia verratessa on huomioitava, että kurssitoteutusten välillä oli merkittäviä eroja, ja myös olosuhteet, jossa tehtävä tehtiin, vaihtelivat merkittävästi. Tuloksia vertaamalla kuitenkin vaikuttaa selvältä, että Rainfall-ongelman kontekstissa ei voi selkeästi sanoa funktionaalisen paradigman tuovan etua imperatiiviseen paradigmaan verrattuna. Funktionaalisen paradigman käytön ei toisaalta voi myöskään todeta heikentävän tehtävästä suoriutumista.

Fisler, Krishnamurthi ja Siegmund (2016) tutkivat palindromiongelman ja lisäyskone (engl. *adding machine*) -ongelman ratkaisua useammalla CS1-kurssilla, joista osa oli imperatiivisia ja osa funktionaalisia. Ongelmien laatimisessa oli tasapuolisuuden vuoksi mukana sekä funktionaalisia että imperatiivisia kieliä suosivia tutkijoita. Palindromiongelmassa tuli ongelman nimen mukaisesti ratkaista, onko parametrina annettu merkkijono palindromi. Lisäyskone-ongelmassa ohjelma sai listan lukuja, jossa oli olemassa nollalla erotettuja osalistoja, ja osalistojen lukujen summat piti laskea ja palauttaa uudessa listassa. Kurssit järjestettiin eri maissa. Tutkimukseen päätyi yksi imperatiivinen kurssi Yhdysvalloista ja toinen Saksasta sekä yksi funktionaalinen kurssi Yhdysvalloista ja toinen Ranskasta. Tutkimuksen pääpaino oli opiskelijoiden ratkaisujen tutkimisessa, mutta tuloksista on nähtävissä, että im-

peratiivisilla kursseilla virheelliset ratkaisut olivat yleisempiä kuin funktionaalisilla kursseilla.

Erityisenä virheenä lisäyskoneongelman yhteydessä Fisler, Krishnamurthi ja Siegmund (2016) tunnistivat `foreach` -silmukan käytössä opiskelijoiden sekoittavan indeksin ja indeksiä vastaavan arvon keskenään. Yhtenä syynä tähän arveltiin huono nimeäminen, sillä kokonaislukutaulukkoa läpikäydessä opiskelijat usein nimittivät arvoa `i`:ksi, joka on yleinen tunnus indeksille. Tavallisemmassa `for`-silmukassa opiskelijat taas törmäsivät ongelmiin indeksiaritmetiikan kanssa tunnistaessa taulukossa olevia nollia. Toisaalta he rekursiota hyödyntäneet funktionaalisten kurssien opiskelijat, jotka pitivät alilistan summan listan ensimmäisessä alkiossa, epäonnistuivat useammin käsittelemään sellaisia epätyhjiä alilistoja, joiden lukujen summa oli nolla. Tutkimuksen yhteydessä toteutessa kyselyssä imperatiivisten kurssien opiskelijat suosivat funktionaalisten kurssien opiskelijoita useammin yksittäistä silmukkaa hyödyntäviä ratkaisuja verraten monivaiheisimpiin ratkaisuihin. Yksittäisen silmukkarakenteen suosion syitä ei tosin tarkennettu. Yleisesti opiskelijat perustelivat valitsemiaan rakenteita suorituskyvyllä, koodin luettavuudella, koodin tekemien laskutoimitusten selkeydellä ja koodin helpolla muokattavuudella tulevaisuudessa.

Fisler, Krishnamurthi ja Siegmund (2016) totesivat, että imperatiivisten kurssien opiskelijat olivat usein huolissaan ohjelman suorituskyvystä. Suorituskykyä ei kuitenkaan oltu kursseilla erityisemmin painotettu, vaan jopa kehotettu olemaan miettimättä sitä. Erityisesti imperatiivisten kurssien opiskelijat kritisoivat välitulosten laskemista epätehokkaana. Huolimatta suorituskykyhuolesta, imperatiivisten kurssien opiskelijat hyödynsivät usein sellaisia Java-kielen standardikirjaston metodeja, jotka tekevät tietorakenteista välituloksia tai tietorakenteiden useita läpikäyntejä taustalla. Fisler, Krishnamurthi ja Siegmund (2016) tulkitsevat tämän tarkoittavan, että opiskelijoilla on heikko ymmärrys standardikirjaston metodien suorituskyvystä. Koska nykyaikaiset tietokoneet ovat yleensä riittävän nopeita huolimatta optimoinnin puutteesta, suorituskykyyn ei yleensä tarvitse nykyaikaisilla tietokoneilla erityisemmin keskittyä, vaan koodin ylläpidettävyys nähdään tärkeämpänä.

Siten suorituskykyyn keskittymisen CS1-kurssilla voi nähdä negatiivisena piirteenä. Fisler, Krishnamurthi ja Siegmund (2016) pohtivat huolen suorituskyvystä pohjautuvan opiskelijoiden omaan, kurssia edeltävään kokemukseen. Funktionaalisten kurssien opiskelijoiden keskuudessa ei huomattu vastaavaa huolta suorituskyvystä.

4 Tutkimus

Tässä luvussa kerrotaan tutkimusmenetelmä yksityiskohtaisesti. Lisäksi esitellään tutkimuksen kontekstina toimiva ohjelmointikurssi, aineiston keruumenetelmä ja tutkimusta varten laadittu materiaali ja tehtävät.

4.1 Tutkimusmenetelmä

Tutkimuksen menetelmänä on satunnaistettu vertailukoe (engl. *randomized controlled trial*, lyhennettynä *RCT*). Satunnaistetut vertailukokeet ovat menetelmänä tietotekniikassa harvinaisia (Kaijanaho 2015). Erityisen tunnettuja ne ovat lääketieteen parissa, jossa RCT-tutkimuksia on historiallisesti pidetty muilla menetelmillä suoritettuja tutkimuksia merkitsevämpinä (Stanley 2007). Vaikka satunnaistettu vertailukoe on tutkimusmenetelmänä saanut myös kritiikkiä (Deaton ja Cartwright 2018), sitä pidetään edelleen erittäin arvostettuna tutkimusmenetelmänä.

Käyttäen Hanenbergin (2015) luokitusta, tämän tutkimuksen lähestymistapa on empiirinen ihmiskeskeinen tuotetutkimus (*empirical human-centered product study*). Lähestymistavassa yritetään saada lisää tietoa tutkittavasta ilmiöstä tutkimalla jotain ihmisten tuottamaa tulosta, tutkimuksen tapauksessa opiskelijoiden kirjoittamaa ohjelmakoodia. Hanenberg jakaa empiiriset menetelmät vielä laadullisiin ja määrällisiin. Laadullisessa menetelmässä tuloksista kerätään dataa, jota ei voi merkityksellisesti kuvailla määrinä. Tutkimus on pääasiassa määrällinen, mutta myös laadullista menetelmää käytetään: opiskelijoiden vastauksia arvioidaan määrällisesti, mutta tarvittaessa turvaudutaan myös vastausten laadulliseen tarkasteluun määrällisten muuttujien arvojen syiden selvittämisessä.

Hanenberg (2015) luokittelee myös kokeita. Tutkimuksessa tehtävässä kokeessa on yksi riippumaton muuttuja, joka on ongelmanratkaisussa käytetty tekniikka. Sen käsittelyjä (engl. *treatment*) on kaksi: silmukat ja lambda-lausekkeet, ja yksi osallistuja ratkaisee tehtävän näistä vain yhdellä tavalla. Siten koe on Hanenbergin (2015) luokitteluun perustuen *One-Factor Design with Two Alternatives (AB-Between-Subject)*.

Ko, Latoza ja Burnett (2015), joiden artikkeliin myös Hanenberg viittaa, käsittelevät ohjelmointityökaluja (*software engineering tools*) tutkivien kokeiden suunnittelua. Vaikka ohje on nimensä mukaisesti ohjelmointityökaluille, se soveltuu sekä tutkijan että Hanenbergin (2015) mukaan hyvin kontrolloitujen kokeiden tekemiseen yleisestikin ohjelmistotieteessä (*software science*). Siten se soveltuu myös ohjelmointikielten tutkimukseen. Ko, Latoza ja Burnett (2015) ovat jakaneet kokeen suunnittelun seuraavaan kronologiseen järjestykseen:

1. Osallistujien rekrytointi
2. Osallistujien valinta
3. Suostumuksen hankkiminen osallistujilta
4. Kokeen suorittamisprosessin suunnittelu
5. Demografinen mittaaminen
6. Osallistujien jakaminen ryhmiin
7. Osallistujien kouluttaminen
8. Tehtävien tekeminen ja jakaminen osallistujille
9. Lopputuloksen mittaaminen
10. Raportointi ja osallistujien palkitseminen

Tutkimuksessa osallistujat olivat ohjelmointikurssin opiskelijoita. Rekrytoinnissa opiskelijoita kannustettiin osallistumaan tutkimukseen ohjelmointikurssin arvosanaan positiivisesti vaikuttavan pistemäärän kasvattamisella ja päivittäistavarakaupan lahjakorttiarvonnalla. Osallistujia ei siivilöity, vaan kaikki halukkaat saivat osallistua. Opiskelijoilta kysyttiin suostumus tutkimukseen osallistumisesta: opiskelijat saivat vapaasti valita, tekivätkö he tutkimusta varten laaditut tehtävät (jatkossa koetehtävät), vai vaihtoehtoiset, kurssilla yleensä käytetyt tehtävät, joista ei kerätty dataa. Valittuaan koetehtävät opiskelijoilta kysyttiin koetehtävät paljastaneella sivulla vielä erikseen lupa heidän vastaustensa käyttämiseen tutkimustarkoituksessa. Opiskelijoita myös tiedotettiin tutkimuksesta sekä sen tarkoituksesta ja toteutuksesta. Lisäksi opiskelijoilla oli pääsy koetta varten laadittuun tietosuojaselosteeseen. Tietosuojaseloste on luettavissa liitteessä A.

Osallistuneet opiskelijat jaettiin kahteen ryhmään satunnaisesti arpomalla. Yksi ryh-

mä (lambdaryhmä) teki heille annetut tehtävät lambda-lausekkeita hyödyntäen, ja toinen ryhmä (silmukkaryhmä) teki tehtävät silmukoilla. Lambdaryhmän osallistujat koulutettiin tehtäviä varten ohjelmointikurssin luentomonisteeseen kirjoitetun lisämateriaalin avulla. Silmukkaryhmän osallistujat olivat koulutettu tehtäviä varten jo valmiiksi kurssin aikana. Tehtävät suunniteltiin ja toteutettiin lambdaryhmälle tehtyyn materiaaliin nojautuen.

Opiskelijat tekivät tehtävät osana kurssin tavanomaisia viikottaisia harjoitustehtäviä. Opiskelijoita kehoitettiin itsenäisyyteen tehtäviä tehdessä, mutta tarvittaessa he saivat kysyä apua kurssin ohjaajilta, jotka oltiin myös koulutettu tutkimusta varten materiaalilla ja tiedotuksella. Ohjaajia kehotettiin myös ottamaan yhteyttä tutkijaan, mikäli tehtävissä tulisi vastaan epäselvyyksiä tai teknisiä ongelmia. Lopputulosta mitattiin opiskelijakohtaisesti käännöskertojen määrällä sekä pistemäärällä, jonka määritti jokaiseen tehtävään erikseen rakennettu automaattinen tarkistin. Lisäksi lopputulosta mitattiin opiskelijoiden itse antamalla aika-arviolla ja vaikeustasoarviolla. Opiskelijoilla oli viikko aikaa tehtävien tekemiseen ja tehtäviä sai yrittää sekä vastauksia palauttaa niin monta kertaa kuin halusi. Mitattaessa lopputulosta käännöskerrat sekä opiskelijoiden antamat arviot olivat painavampia kuin pistemäärät, sillä kun tehtävät tehtiin osana harjoituksia ja tehtäviin sai tarvittaessa apua, oli odotettavissa, että opiskelijat yleisesti saavat tehtävistä täydet pisteet. Tehtävien yhteydessä opiskelijoilla oli myös kenttä, jonka avulla he pystyivät antamaan avointa palautetta tehtävistä. Raportoinnin osalta koetehtävien mallivastaukset paljastettiin opiskelijoille palautusajan umpeuduttua, ja osallistujat palkittiin kurssin lisäpisteellä ja lahjakorttiarvonnalla.

Koska RCT-tutkimuksia tehdään tietotekniikassa vähän, on syytä katsoa tutkimusta myös yleisempien lääketieteellisten RCT-tutkimuksen suunnitteluohjeiden näkökulmasta. Stanley (2007) nostaa esille useita lääketieteellisen RCT-tutkimuksen suunnitteluun liittyviä kohtia. Näistä ison osan voi nähdä pätevän myös ohjelmointikielten tutkimuksessa. Tutkimuksella tulisi olla yksittäinen selkeä tavoite, sillä liian monta tavoitetta voi haitata tiedon keruuta ja käsittelyä siten, että yhtäkään tiettyä tutkimuskysymystä ei saada kunnollisesti ratkaistua. Tämän tutkimuksen osal-

ta tarkoitus on selvittää lambda-lausekkeiden hyödyllisyyttä tietorakenteiden käsittelyssä ohjelmointi- ja opetuskäytössä verrattuna silmukoiden käyttämiseen, joka on yksittäinen tutkimuskysymys. Mahdollisen tuloksen mielivaltaisen tulkitsemisen välttämiseksi tavoitteiden mittaustavat täytyy olla selvillä jo ennen koetta (Stanley 2007). Tämän tutkimuksen osalta mittauskeinot olivat tiedossa jo ennen koetta, sillä ryhmien suoriutumista mitattiin osallistujien pistemäärillä ja tehtävien yrityskerroilla sekä osallistujien antamilla arvioilla tehtäviin kuluneesta ajasta ja tehtävien vaikeustasosta.

Stanley (2007) toteaa, että eettisistä syistä RCT-tutkimuksen tulee jatkua vain niin kauan, kuin epävarmuutta tutkittavasta asiasta on, ja lisäksi yhden tutkimukseen osallistuvan ryhmän tulee saada standardikäsittely. Lisäksi standardikäsittelyä heikompaa käsittelyä ei ole syytä tutkia. Tämän tutkimuksen tapauksessa lambda-lausekkeiden käyttö tietorakenteiden käsittelyssä on epävarma kysymys, ja silmukoiden käyttäminen toimii standardikäsittelynä.

Kokeessa kahden osallistujaryhmän on syytä olla keskenään mahdollisimman samankaltaisia, jotta erot ryhmien yksilöiden välillä eivät vääristä tuloksia. Tämä saadaan aikaan ryhmien satunnaistamisella, eli osallistujien jakamisella ryhmiin satunnaisesti. Satunnaistaminen poistaa vaikutukset tutkijan sekä tiedostetuilta että tiedostamattomilta asenteilta, jotka voisivat vaikuttaa tutkimuksen tuloksiin. Satunnaistaminen ei itsessään varmista, että ryhmät ovat keskenään samankaltaisia, mutta Stanleyn (2007) mukaan sillä on taipumus tuottaa ryhmiä, jotka ovat keskimäärin lähellä toisiaan. Yksi keino varmistaa satunnaistamisen tuloksia on jakaa osallistujat ryhmiin heidän koetta edeltävän tasonsa (ohjelmoinnissa osaamistason, lääketieteessä terveyden) mukaan, ja satunnaistaa heidät osallistujaryhmiin erikseen näissä pienemmissä ryhmissä. Tässä tutkimuksessa näin ei tehty. Toisaalta osallistujien ennakkotasosta oli saatavilla tietoa rajallisesti, ja verraten suuri osallistujamäärä tekee tavallisen satunnaistamisen tuloksista luotettavamman.

Stanleyn (2007) mukaan kolmannen vaiheen RCT:ssä, jossa kahta eri käsittelyä verrataan toisiinsa, tulisi olla osallistujia sadasta tuhanteen. Tässä tutkimuksessa osallistujia oli 187, mikä osuu vaaditulle välille.

Lääketieteessä panostetaan paljon myös osallistujien turvallisuuteen seuraamalla koetta. Ohjelmointikielten tutkimuksessa fyysistä turvallisuusriskiä ei ole, mutta on riski, että yhden ryhmän opiskelijat oppisivat heiltä vaaditut asiat heikommin kuin toisen ryhmän opiskelijat. Koetehtävät tosin kattoivat hieman alle puolet kurssin yhden viikon tehtävistä, kun kurssi kesti 11 viikkoa. Siten riski merkittävästä haitasta oppimiselle oli pieni. Lisää tietoa kurssin rakenteesta kerrotaan luvussa 4.2.

4.2 Konteksti

Tutkimus toteutettiin Jyväskylän yliopiston (JYU) Ohjelmointi 1 (CS1) -kurssin yhteydessä. Kurssin laajuus on 6 opintopistettä (ECTS), ja sen suoritus perustuu luentoihin, viikoittaisiin tehtäviin, harjoitustyöhön ja kurssitenttiin. Viikoittaisista tehtävistä täytyy kurssin suorittamiseksi tehdä vähintään 40 %, mutta vähimmäismäärää enemmän tekemällä saa lisäpisteitä kurssitenttiä varten. Tehtävät tehdään itsenäisesti, mutta opiskelijat voivat halutessaan tulla tekemään tehtävät yliopiston mikroluokissa, joissa he voivat pyytää apua kurssille ohjaajiksi (tuntiopettajiksi) palkatuilta toisilta opiskelijoilta. Tehtävien lisäksi ohjaajilta saa apua myös harjoitustyön tekemiseen ja ohjaajat hoitavat myös harjoitustyön arvioinnin.

Kurssilla käytetään imperatiivista paradigmaa ja C#-kieltä. Opiskelijoiden motivoimiseksi kurssilla on osittain käytössä peliteema. Osalla kurssin tehtävistä hyödynnetään JYU:ssa kehitettyä, alkujaan Microsoftin XNA:n ja nykyisin MonoGamen päälle rakennettua opetustarkoitukseen tehtyä Jypeli-peliohjelmointikirjastoa, jonka avulla opiskelijat tekevät pieniä graafisia sovelluksia ja pelejä. Esimerkiksi yksi kurssin ensimmäisistä ohjelmointitehtävistä on lumiukon piirtäminen ruudulle kirjastoa käyttäen. Myös kurssin harjoitustyö on lähtökohtaisesti peliaiheinen, joskin harjoitustyönä saa myös tehdä toisenlaisen sovelluksen. Vuosittain kurssin sadoista opiskelijoista yleensä kuitenkin vain muutama tekee ei-peliaiheisen harjoitustyön.

Kurssia on nykyisessä muodossaan järjestetty syksystä 2010 alkaen. Ennen syksyn 2010 kurssia järjestettiin vastaava kurssi, jolla käytettiin Javaa. Kokonaisuutena kurssi nykyisellään on siten pitkän ajan kehitystyön tulos. Vuodesta 2010 lähtien

kurssi on järjestetty kahdesti vuodessa, keväällä ja syksyllä.

Kevään 2020 kurssitoteutus kesti 11 viikon ajan. Kurssin eri viikkojen luennoilla käsitellyt aiheet esitetään taulukossa 1.

Viikko	**Sisältö**
1	Kurssiin tutustuminen, yksinkertaisten ohjelmien tekeminen (Hello World) ja kääntäminen komentorivillä
2	Aliohjelmat, muuttujat, kokonaislukujen toiminta tietokoneissa, dokumentointi ja Visual Studion käyttö
3	Muuttujat, operaattorit ja funktiot eli arvon palauttavat aliohjelmat
4	Funktiot, vakiot, ehtolauseet, merkkijonot
5	Ehtolauseet, toistorakenteet, taulukot, muokattavat merkkijonot (`StringBuilder`)
6	Lyhyt katsaus olio-ohjelmointiin
7	Moniulotteiset taulukot ja matriisit
8	Listat ja toistorakenteet
9	Algoritminen ajattelu ja ongelmanratkaisu, assosiaatiotaulukko (`Dictionary`)
10	Rekursio, poikkeukset
11	Kertaus, merkkijonojen paloittelu, liukulukujen toiminta tietokoneissa

Taulukko 1. Jyväskylän yliopiston kevään 2020 Ohjelmointi 1 -kurssin luentojen sisältö viikoittain

Jypeli-kirjastoa kurssilla käytetään erityisesti viikosta 7 eteenpäin. Pelillisyys ja visuaalisuus on kuitenkin vain rakennettuna mukaan algoritmisiin haasteisiin, eikä pelien tekeminen yleensä ole tehtävissä itse tarkoituksena.

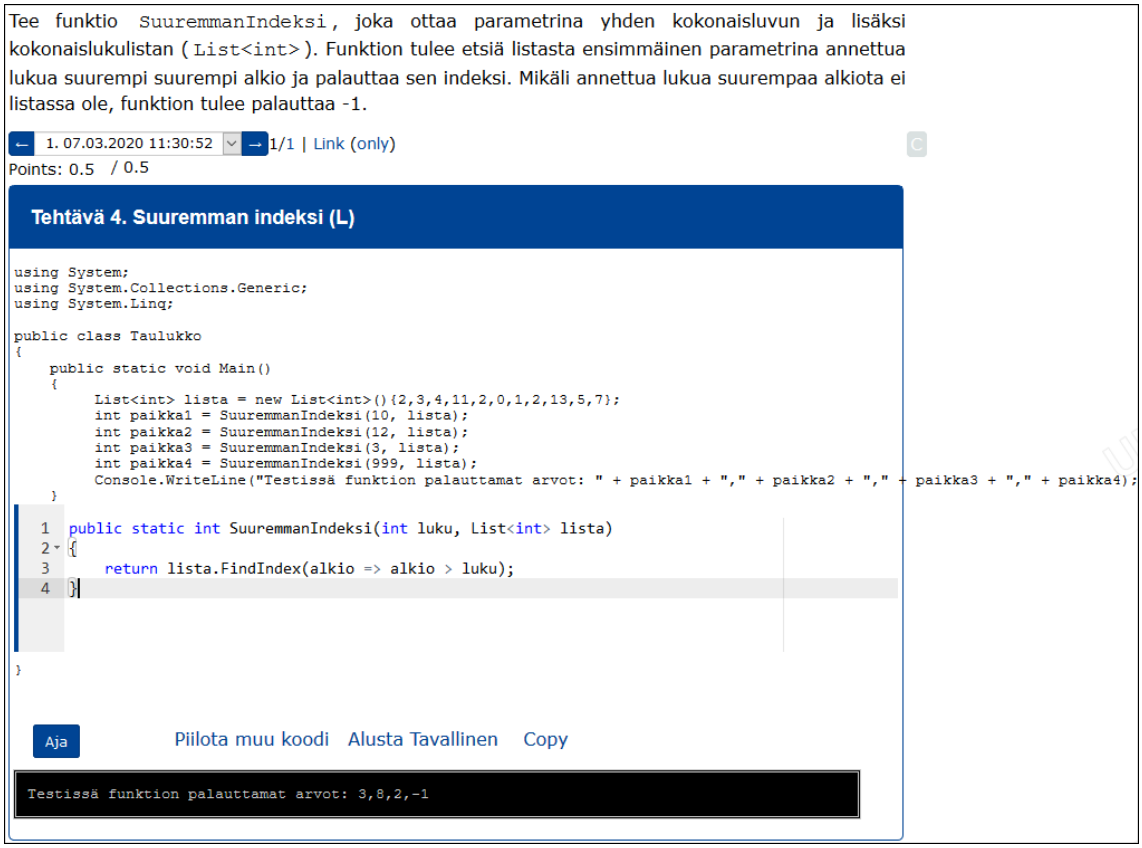
Koe suoritettiin kurssiviikon 9 tehtävien yhteydessä. Opiskelijat olivat siten opiskelleet listat ja toistorakenteet juuri edellisellä viikolla. Lambda-lausekkeita ei luennoilla opetettu, mutta niistä kirjoitettiin materiaali, joka annettiin lambda-tehtävät tehneelle opiskelijaryhmälle linkkinä tehtävien alussa.

Teknisenä alustana tehtävien jakamiselle ja palauttamiselle kurssilla käytetään Jy-

väskylän yliopistossa kehitettyä The Interactive Material (TIM) -järjestelmää. Järjestelmä on alkujaan suunniteltu korvaamaan staattiset luentomonisteet siten, että sivuille saa myös interaktiivista materiaalia. Ajan myötä TIM on otettu käyttöön luentomonisteen lisäksi kaikelle kurssimateriaalille, mukaan lukien viikoittaisille tehtäville.

Kurssin TIM-materiaalissa on eri aiheiden yhteyteen upotettu interaktiivisia koodikomponentteja, joihin käyttäjä voi kirjoittaa koodia, jonka voi ajaa komponentin yhteydessä olevaa painiketta painamalla (ks. kuvio 1). Ajamisen yhteydessä koodi lähetetään palvelimelle, jossa se ajetaan, ja mahdolliset tulosteet palautetaan käyttäjän selaimelle näytettäväksi. Komponenteissa voi myös olla koodia, jota ei näytetä käyttäjälle tai jota käyttäjä ei pääse muokkaamaan. Tämä mahdollistaa esimerkiksi tarkistimien tai debug-koodin upottamisen tällaisena komponenttina toteutetun tehtävän koodiin.

Tehtävissä on usein piilotettu pääohjelma (`Main`), joka kutsuu jotain funktiota ja tekee funktion paluuarvoille tarkistuksia. Opiskelijan täytyy näissä tehtävissä toteuttaa pääohjelman kutsuma funktio siten, että se toimii loogisesti oikein. Palvelin tarkistaa funktion toiminnan ja mahdolliset tehtävän muut vaatimukset ja antaa toteutuksesta pisteitä vastauksen oikeellisuudesta riippuen. Koetehtävät toteutettiin tällaisina komponentteina. Tehtäviin lisättiin syntaktinen tarkistin, joka varmisti, että tehtävät tehtiin tarkoitetulla tavalla. Esimerkiksi lambdaryhmään kuuluneet opiskelijat eivät voineet tehdä tehtäviään silmukoiden avulla, ja silmukkaryhmään kuuluneet opiskelijat eivät voineet hyödyntää C#-kielen listaluokan (`List<T>`) valmiita funktioita.



Kuvio 1. Tehtävä TIM-järjestelmässä koodikomponenttina

Vaikka opiskelijat voivat kirjoittaa tehtävien vastaukset suoraan TIM-järjestelmässä, järjestelmä ei tarjoa koodin kirjoittamiseen vastaavaa apua kuin Visual Studio. Siten opiskelijoita kehotettiin kurssilla kirjoittamaan tehtävien koodi Visual Studiossa ja kopioimaan valmis vastaus TIM-järjestelmän komponenttiin, jonka avulla vastauksen voi palauttaa ja tarkistaa.

4.3 Aineiston keruu

Tutkimus suunniteltiin loppuvuoden 2019 ja alkuvuoden 2020 aikana. Koe suoritettiin ja aineisto kerättiin maaliskuussa 2020. Tutkimuksessa laadittiin tehtäviä, joissa käsiteltiin listoja. Osallistuneet opiskelijat jaettiin arpomalla kahteen ryhmään, joista ensimmäisen tuli tehdä tehtävät silmukoilla, ja toisen ryhmän lambda-lausekkeilla hyödyntäen C#-kielen standardikirjaston listatoteutuksen `List<T>` metodeja. Teh-

tävien vaikeustasoa verrattiin Uesbeckin ym. (2016) C++-kielessä lambda-lausekkeita tutkineen tutkimuksen tehtävien vaikeustasoon. Koska kyseisessä tutkimuksessa osallistujat olivat monentasoisia ohjelmoijia, ja tässä tutkimuksessa osallistujat olivat aloittelijatason ohjelmointikurssin (CS1) opiskelijoita, tehtävistä pyrittiin tekemään hieman yksinkertaisempia kuin kyseisessä tutkimuksessa.

Aineisto kerättiin opiskelijoiden vastauksista, joihin sisältyi jokaisen palautuskerran aikaleima, opiskelijan jokaisella palautuskerralla palauttama lähdekoodi sekä opiskelijan saama pistemäärä. Opiskelijat antoivat myös oman arvionsa tehtävien haastavuudesta ja työmäärästä. Aineistoa analysoitiin tilastollisesti tutkijan itse erikseen aineiston analyysia varten toteuttamalla C#-sovelluksella, Microsoftin Excelillä ja IBM:n SPSS-ohjelmistolla. Laadullista analyysia varten opiskelijoiden viimeisimmät vastaukset sekä avoin palaute katsottiin läpi myös silmin.

4.4 Materiaali ja tehtävät

Ennen tutkimusta ohjelmointikurssin materiaalissa oli vain kaksi lyhyttä mainintaa ja esimerkkiä lambda-lausekkeista. Siten varsinainen opetusmateriaali täytyi erikseen tehdä tutkimuksen yhteydessä. Tehty materiaali lisättiin alalukuna kurssin luentomonisteen dynaamisia tietorakenteita eli käytännössä listoja käsittelevään lukuun, ja lambdaryhmään kuuluneita kehotettiin lukemaan materiaali ennen tehtävien tekemistä. Materiaalin selkeys tarkistettiin koeluetuttamalla se kahdella kurssin jo käyneellä opiskelijalla. Esimerkki materiaalista on nähtävissä kuviossa 2.

23.3 Anonyymit funktiot (lambda-lausekkeet)

Listoja voidaan käsitellä silmukoiden lisäksi myös `List<T>`-luokan metodeilla. Monet näistä metodeista ottavat parametrina aliohjelman, jonka avulla listaa käsitellään. Tällaisia metodeja ovat esimerkiksi `Find`, `FindIndex`, `Exists`, `FindAll` ja `ForEach`. Näille metodeille voidaan antaa argumenttina aliohjelma metodin kutsun kaarisulkeiden sisään. Valmiilla metodeilla käsittely silmukoiden sijaan mahdollistaa usein vastaavan asian tekemisen huomattavasti pienemmällä määrällä koodia.

Usein helpoin keino antaa näille metodeille parametreja on tehdä aliohjelmat *lambda-lausekkeina*, jossa parametrina annettava aliohjelma määritellään nimettömästi suoraan parametrilausekkeen sisään. Esimerkiksi jos meillä on lista peliolioita (`List<GameObject>`), jonka nimi on `lista`, seuraava kutsu etsisi listasta ensimmäisen olion, joka on ympyrän muotoinen, ja asettaisi sen muuttujaan `pallo`:

```
GameObject pallo = lista.Find(olio => olio.Shape ==
Shape.Circle);
```

Kuvio 2. Esimerkki lambda-lausekkeen opettamista varten tehdystä materiaalista. Koko materiaali on sisällytetty liitteeseen B

Silmukoita sen sijaan kurssilla oli käsitelty runsaasti jo ennen tutkimusta, eikä uutta materiaalia tarvinnut tehdä. Esimerkki silmukoita koskevasta materiaalista on nähtävissä kuviossa 3.

16.2 while-silmukka

while-silmukka on yleisessä muodossa seuraava:

```
while (ehto) lause;
```

Kuten ehtolauseissa, täytyy ehdon taas olla jokin lauseke, joka saa joko arvon true tai false. Ehdon jälkeen voi yksittäisen lauseen sijaan olla myös lohko.

```
while (ehto)
{
  lause1;
  lause2;
  lauseX;
}
```

Silmukan lauseita toistetaan niin kauan kuin ehto on voimassa, eli sen arvo on true. Ehto tarkastetaan aina ennen kuin siirrytään seuraavalle kierrokselle. Jos ehto saa siis heti alussa arvon false, ei lauseita suoriteta kertaakaan.

Kuvio 3. Esimerkki luentomonisteen silmukoita koskevasta osiosta

Opetusmateriaalin lisäksi tutkimusta varten luotiin tehtävät. Kokeen toteutusviikolla eli kurssin viikolla 9 opiskelijat tekivät koetehtävien lisäksi myös muita kurssin kyseisen viikon tehtäviä. Viikoittaisissa tehtävissä koetehtävät olivatkin numeroitu 4–7, mutta raportointia varten tehtävät on numeroitu uudelleen välille 1–4. Kaikki koetehtävät olivat opiskelijoille pakollisia riippumatta heidän tutkimusryhmästään. Alla on tehtävänannot lainauksina kevään 2020 Ohjelmointi 1 -kurssin materiaalista (Jyväskylän yliopisto 2020). Lisäksi esitellään jokaisen tehtävän mallivastaus niin lambda-lausekkeilla kuin myös silmukoilla toteutettuna.

T1 (0.5 p):

Tee funktio `SuuremmanIndeksi`, joka ottaa parametrina yhden kokonaisluvun ja lisäksi kokonaislukulistan (`List<int>`). Funktion tulee etsiä listasta ensimmäinen parametrina annettua lukua suurempi alkio ja palauttaa sen indeksi. Mikäli annettua lukua suurempaa alkiota ei listas-

sa ole, funktion tulee palauttaa -1.

Lambdatoteutuksen mallivastaus:

```
public static int SuuremmanIndeksi(int luku, List<int> lista)
{
    return lista.FindIndex(alkio => alkio > luku);
}
```

Silmukkatoteutuksen mallivastaus:

```
public static int SuuremmanIndeksi(int luku, List<int> lista)
{
    for (int i = 0; i < lista.Count; i++)
    {
        if (lista[i] > luku)
            return i;
    }

    return −1;
}
```

T2 (0.5 p):

Tee funktio `KerroKaikki`. Funktio ottaa parametrina kokonaislukulistan ja palauttaa luvun, joka saadaan, kun kaikki listan luvut kerrotaan keskenään. Esimerkiksi jos listassa on luvut 2, 3 ja 4, niin tulos olisi 2 * 3 * 4 = 24. Mikäli lista on tyhjä, niin tulee palauttaa luku 1.

Lambdatoteutuksen mallivastaus:

```
public static int KerroKaikki(List<int> lista)
{
    int tulos = 1;
    lista.ForEach(luku => tulos = tulos * luku); // tai tulos *= luku
    return tulos;
}
```

Silmukkatoteutuksen mallivastaus:

```
public static int KerroKaikki(List<int> lista)
{
    int tulos = 1;
    foreach (int i in lista)
    {
        tulos = tulos * i;
    }
    return tulos;
}
```

T3 (1 p):

Tee aliohjelma `TuhoaSuuret`, joka ottaa parametrina listan fysiikkaolioista (`List<PhysicsObject>`) ja tuhoaa niistä kaikki, joiden leveys (Width) on yli 50.

Lambdatoteutuksen mallivastaus:

```
public static void TuhoaSuuret(List<PhysicsObject> oliot)
{
    oliot.ForEach(olio => {
        if (olio.Width > 50.0)
            olio.Destroy();
    });
}
```

Silmukkatoteutuksen mallivastaus:

```
public static void TuhoaSuuret(List<PhysicsObject> oliot)
{
    foreach (PhysicsObject olio in oliot)
    {
        if (olio.Width > 50.0)
            olio.Destroy();
    }
}
```

T4 (1 p):

Tee funktio `VaihdaVari`. Funktio ottaa parametrina listan fysiikkaolioita (`List<PhysicsObject>`), etsii listasta kaikki punaiset (`Color.Red`) suorakulmiot, ja vaihtaa niiden värin siniseksi (`Color.Blue`). Funktio palauttaa listan kaikista olioista, joiden väriä se muutti.

Lambdatoteutuksen mallivastaus:

```
1  public static List<PhysicsObject> VaihdaVari(List<PhysicsObject> oliot)
2  {
3      List<PhysicsObject> varjattavat = oliot.FindAll(
4          olio => olio.Shape == Shape.Rectangle && olio.Color == Color.Red);
5
6      varjattavat.ForEach(olio => olio.Color = Color.Blue);
7      return varjattavat;
8  }
```

Silmukkatoteutuksen mallivastaus:

```
1  public static List<PhysicsObject> VaihdaVari(List<PhysicsObject> oliot)
2  {
3      List<PhysicsObject> varitetyt = new List<PhysicsObject>();
4      foreach (PhysicsObject olio in oliot)
5      {
6          if (olio.Shape == Shape.Rectangle && olio.Color == Color.Red)
7          {
8              olio.Color = Color.Blue;
9              varitetyt.Add(olio);
10         }
11     }
12     return varitetyt;
13 }
```

Tehtävän 4 toteutusten mallivastaukset eivät ole toiminnaltaan täysin yhteneviä, mutta antavat saman lopputuloksen. Molemmat ratkaisut käyvät `oliot`-listan läpi kerran, mutta lambda-ratkaisu käy vielä `varjattavat`-listan läpi erikseen. Suu-

rimmassa osassa kurssilla esiintyvistä käyttötarkoituksista tämä ei kuitenkaan muodosta merkittävää eroa suorituskykyyn.

5 Tulokset

Tässä luvussa esitellään tutkimuksen tulokset ja otetaan niistä olennaiset poiminnot lähempään tarkasteluun. Tulosten pääpaino on tilastollisessa analyysissa, mutta myös laadullisen analyysin tulokset käydään läpi.

5.1 Tilastollinen analyysi

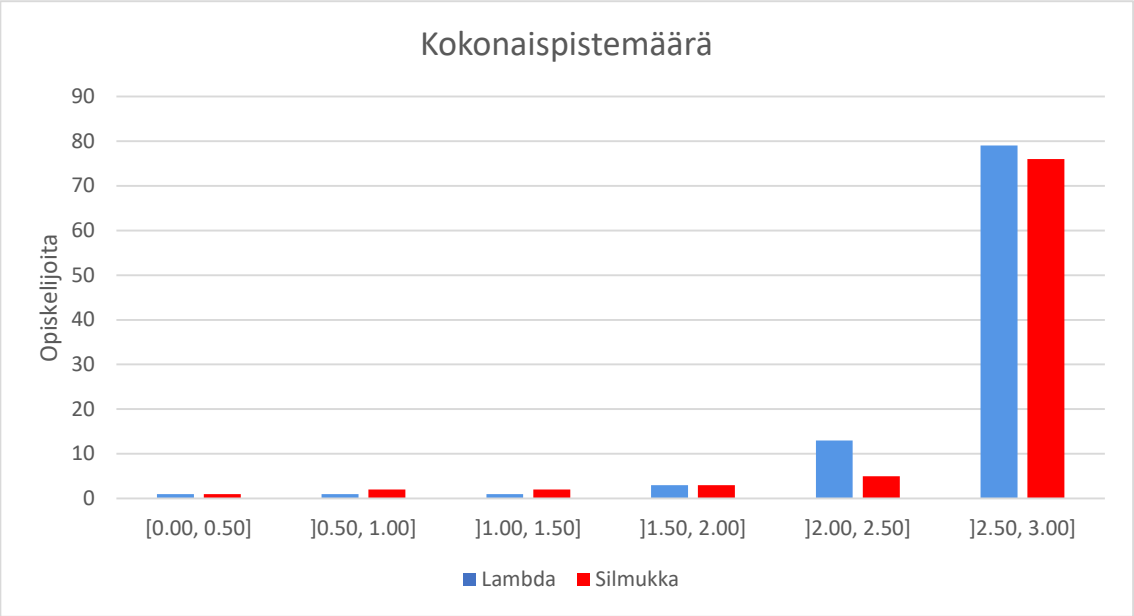
	Lambda			Silmukka		
	N	Keskiarvo	Keskihajonta	N	Keskiarvo	Keskihajonta
	98			**89**		
T1 pisteet	98	0.49	0.07	88	0.47	0.12
T2 pisteet	97	0.50	0.00	88	0.49	0.07
T3 pisteet	97	0.97	0.16	87	0.96	0.17
T4 pisteet	97	0.90	0.24	87	0.93	0.22
Yhteispisteet		2.83	0.45		2.80	0.55
T1 yritykset		4.00	6.27		5.04	9.85
T2 yritykset		3.39	4.89		3.13	3.40
T3 yritykset		3.49	6.53		2.96	4.10
T4 yritykset		5.96	8.11		3.47	3.87
Aika-arvio	95	2.71	2.46	86	2.75	2.40
Vaikeusarvio	95	5.12	2.11	86	5.01	2.36

Taulukko 2. Tutkimusaineiston keskeiset muuttujat sekä niiden keskiarvot ja keskihajonnat. Tehtävien maksimipistemäärät löytyvät luvun 4.4 tehtäväkuvauksista

Tutkimusaineiston keskeiset muuttujat esitellään taulukossa 2. Opiskelijat antoivat ensin vaikeusarvion asteikolla, jossa 1 tarkoitti erittäin vaikeaa ja 10 erittäin helppoa. Tulosten tulkitsemisen selkeyttämiseksi asteikko käännettiin toisin päin analyysin yhteydessä.

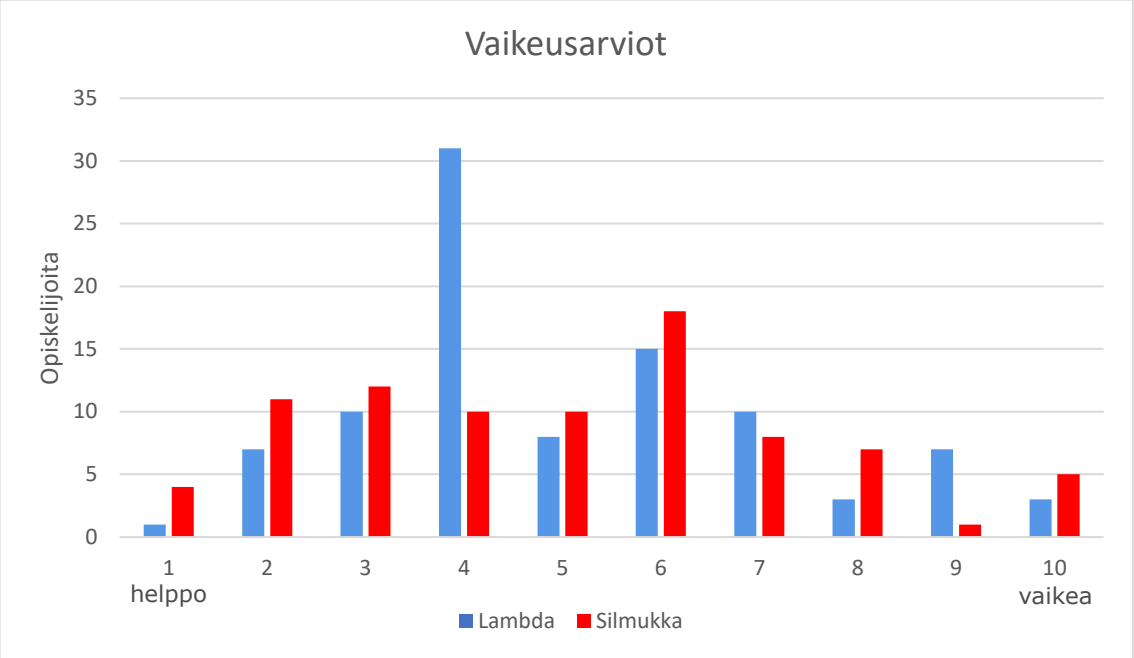
Tehtävien maksimipistemäärä oli 3,0. Selvä enemmistö molempien ryhmien opiskelijoista sai tehtävistä täydet pisteet (ks. kuvio 4), eikä ryhmien pistemäärissä ole tilastollisesti merkitsevää eroa (Mann-Whitneyn U-testi, Sig = 0,275). Tämä oli osin odotettua, sillä koetehtävät olivat pakollisia viikoittaisia tehtäviä, jotka opiskelijat yleensä tekevät täysin valmiiksi. Tarvittaessa opiskelijat saivat myös apua kurssin

ohjaajilta tehtävien tekemistä varten.



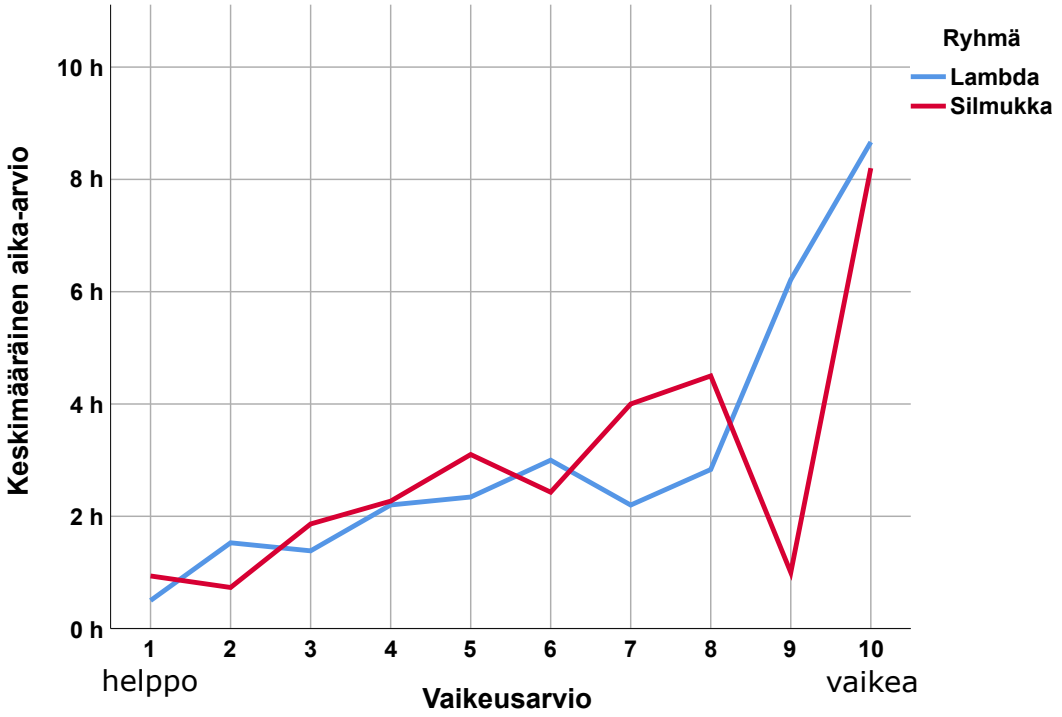
Kuvio 4. Opiskelijoiden koetehtävistä saamien pisteiden jakauma

Koska korkeat pisteet olivat odotettavissa, pistemäärien sijaan tärkeämpää on keskittyä opiskelijoiden antamiin aika- ja vaikeustasoarvioihin. Vaikeustasoarvioinnissa opiskelijoita pyydettiin arvioimaan tehtävien vaikeustaso välillä 1–10. Kuvio 5 esittää molempien ryhmien vaikeustasoarvioiden jakauman. Lambdaryhmässä selvästi yleisin vaikeustasoarvio oli 4, kun silmukkaryhmässä se oli 6. Keskimääräinen vaikeustasoarvio ryhmien kesken oli kuitenkin lähellä toisiaan (ks. kuvio 2). Aikaarviot painottuvat molemmissa ryhmissä alle kolmeen tuntiin (ks. kuvio 2).

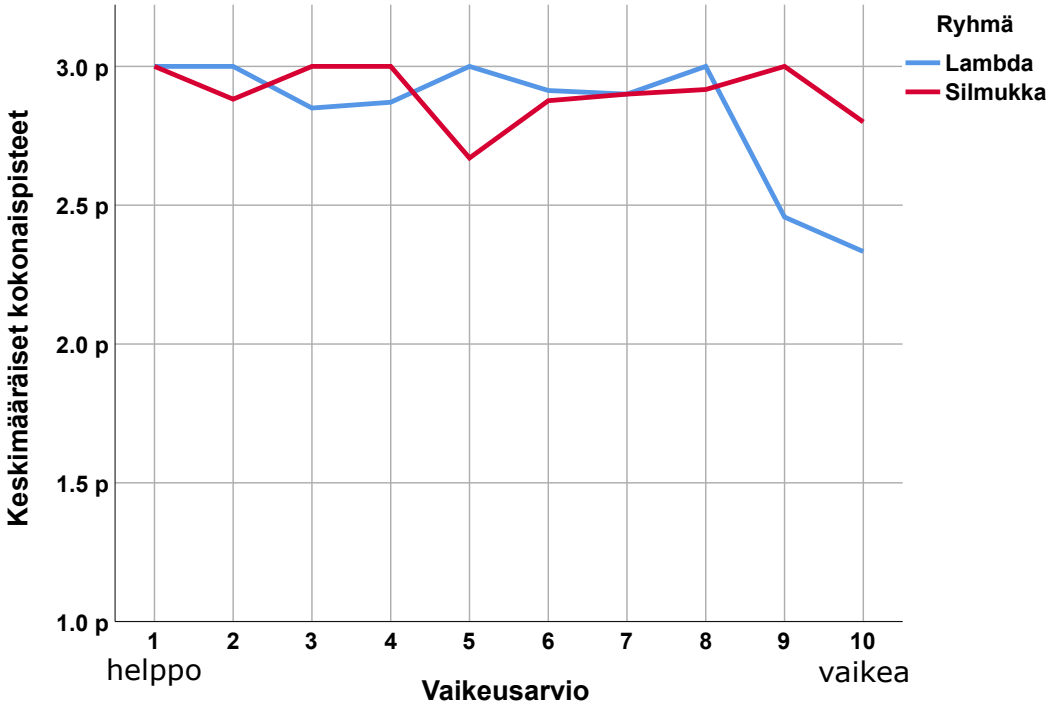


Kuvio 5. Opiskelijoiden tehtävistä antama vaikeustasoarvio

Aika-arviossa pyydettiin arvioimaan koetehtäviin kulunut aika tunteina. Kuten odotettua, kuviosta 6 on nähtävissä, että opiskelijat, jotka kokivat tehtävät vaikeiksi, antoivat myös suuremman arvion tehtävien tekoajalle kuin opiskelijat, jotka kokivat tehtävät helpoiksi. Silmukkaryhmässä vaikeustasoarvion yhdeksän kohdalla olevan notkahduksen aika-arviossa selittää se, että vain yksi silmukkaryhmän opiskelija antoi tehtävien vaikeustasoarvioksi yhdeksän (ks. kuvio 5). Tehtävien koetun vaikeuden ja opiskelijan saaman kokonaispistemäärän välillä ei ole graafisesti havaittavissa selkeää korrelaatiota (ks. kuvio 7).



Kuvio 6. Keskimääräinen opiskelijoiden antama aika-arvio vaikeusarvion funktiona. Käyrä näyttää aika-arvion keskiarvon eri vaikeusarvion antaneiden kesken. Mitä vaikeammaksi opiskelijat kokivat tehtävät, sitä enemmän aikaa heillä kului tehtävien tekemiseen



Kuvio 7. Keskimääräinen opiskelijoiden koetehtävistä saama kokonaispistemäärä vaikeusarvion funktiona. Käyrä näyttää pistemäärän keskiarvon eri vaikeusarvion antaneiden kesken

Aika- ja vaikeustasoarvioiden normaalijakautuneisuutta tutkittiin Kolmogorov-Smirnov- ja Shapiro-Wilk -testeillä. Kuten kuviosta 8 on nähtävissä, kummankaan ryhmän arviot aika- ja vaikeustasosta ei noudata normaalijakaumaa. Koska arviot ei noudata normaalijakaumaa, tutkittiin ryhmien välisten erojen tilastollista merkitsevyyttä Mann-Whitneyn U -testillä sekä Kruskal-Wallis-testillä, jotka eivät oleta aineiston normaalijakautuneisuutta. Testien mukaan erot eivät ole tilastollisesti merkitseviä (ks. kuvio 9).

Tests of Normality

		Kolmogorov-Smirnov [a]			Shapiro-Wilk		
	Ryhmä	Statistic	df	Sig.	Statistic	df	Sig.
Vaikeusarvio	Lambda	.217	95	.000	.932	95	.000
	Silmukka	.113	83	.010	.954	83	.005
Aika-arvio	Lambda	.287	95	.000	.746	95	.000
	Silmukka	.234	83	.000	.795	83	.000

a. Lilliefors Significance Correction

Kuvio 8. Aika- ja vaikeusarvioille tehty normaalijakautuvuustesti

Hypothesis Test Summary

	Null Hypothesis	Test	Sig.	Decision
1	The distribution of Aika-arvio is the same across categories of Ryhmä.	Independent-Samples Mann-Whitney U Test	.854	Retain the null hypothesis.
2	The distribution of Aika-arvio is the same across categories of Ryhmä.	Independent-Samples Kruskal-Wallis Test	.854	Retain the null hypothesis.
3	The distribution of Vaikeusarvio is the same across categories of Ryhmä.	Independent-Samples Mann-Whitney U Test	.704	Retain the null hypothesis.
4	The distribution of Vaikeusarvio is the same across categories of Ryhmä.	Independent-Samples Kruskal-Wallis Test	.704	Retain the null hypothesis.

Asymptotic significances are displayed. The significance level is ,050.

Kuvio 9. Aika- ja vaikeustasoarvioiden erojen tilastolliselle merkitsevyydelle tehdyt testit

Pistemäärien ja arvioiden lisäksi mitattiin myös tehtäväkohtaista yrityskertojen määrää. Yrityskerrat eivät ole tarkkoja yksittäisen opiskelijan kohdalla, sillä monet opiskelijat tekivät tehtävät ensin Visual Studiossa, ja palauttivat TIM-järjestelmään vain valmiit vastaukset, eikä tutkijalla ollut keinoa seurata opiskelijoiden tehtävien tekoa Visual Studiossa. Toiset opiskelijat taas tekivät tehtävät suoraan TIM-järjestel-

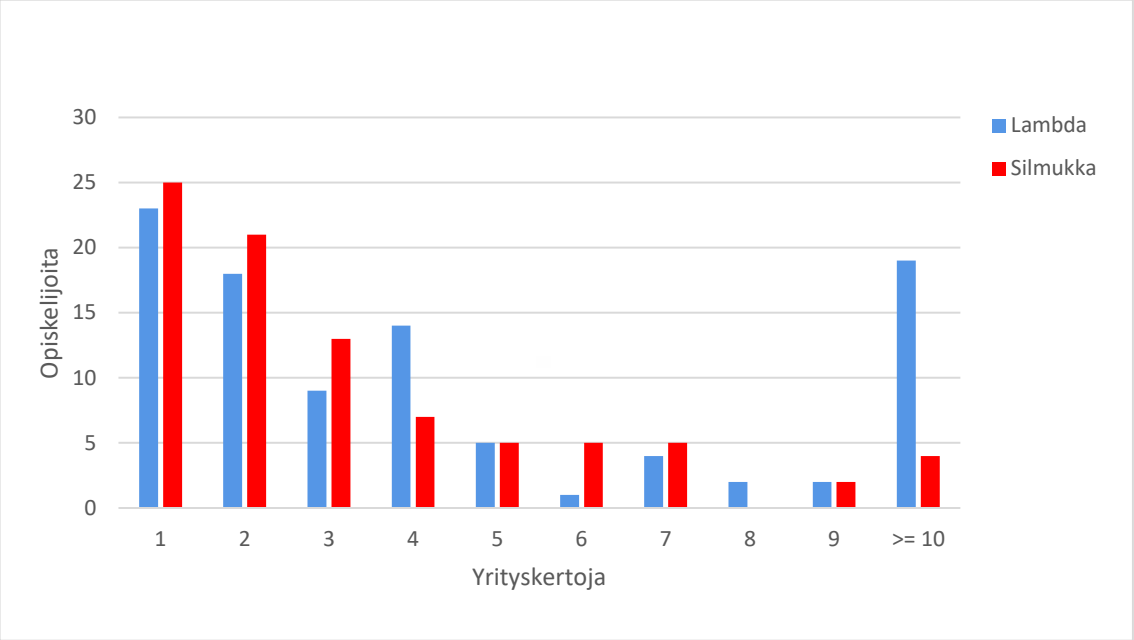
män tekstieditorilla. Kun yrityskertoja seurataan TIM-järjestelmään tehtyjen palautusten määrällä, jälkimmäiselle opiskelijaryhmälle tulee yleensä enemmän yrityskertoja kuin Visual Studiota käyttäville opiskelijoille. Osallistujien korkean määrän ja satunnaisuuteen perustuvan ryhmävalinnan vuoksi voi tosin olettaa, että eri työkaluja käyttävät osallistujat jakautuvat tasaisesti molempiin ryhmiin. Tehtävien yrityskerrat eivät noudata normaalijakaumaa. Analysoidessa yksittäisten tehtävien sekä kaikkien tehtävien yhteenlaskettua yrityskertojen määrää ryhmien välillä Mann-Whitneyn U-testillä, vain tehtävässä 4 tulee ilmi tilastollisesti merkitsevä ero (ks. kuvio 10). Tarkastelemalla kuviota 11 nähdään, että lambdaryhmän opiskelijat käyttivät kyseiseen tehtävään enemmän yrityskertoja kuin silmukkaryhmän opiskelijat.

Hypothesis Test Summary

	Null Hypothesis	Test	Sig.	Decision
1	The distribution of T1 yrityskerrat is the same across categories of Ryhmä.	Independent-Samples Mann-Whitney U Test	.378	Retain the null hypothesis.
2	The distribution of T2 yrityskerrat is the same across categories of Ryhmä.	Independent-Samples Mann-Whitney U Test	.439	Retain the null hypothesis.
3	The distribution of T3 yrityskerrat is the same across categories of Ryhmä.	Independent-Samples Mann-Whitney U Test	.211	Retain the null hypothesis.
4	The distribution of T4 yrityskerrat is the same across categories of Ryhmä.	Independent-Samples Mann-Whitney U Test	.033	Reject the null hypothesis.
5	The distribution of Kokonaisyrityskerrat is the same across categories of Ryhmä.	Independent-Samples Mann-Whitney U Test	.782	Retain the null hypothesis.

Asymptotic significances are displayed. The significance level is .050.

Kuvio 10. Mann-Whitneyn U-testi ryhmien tehtävien yrityskertojen erojen tilastollisesta merkitsevyydestä. Tehtävässä 4 (värin vaihto) on ainoa tilastollisesti merkitsevä ero ryhmien välillä (sig. < 0.05)



Kuvio 11. Osallistujien tehtävän 4 yrityskerrat. Lambdaryhmässä oli suuri määrä opiskelijoita, joilla meni tehtävään yli 9 yrityskertaa

Otettaessa ryhmät, vaikeusarviot, aika-arviot ja tehtävien yrityskerrat tarkasteluun korrelaatioanalyysissa, voidaan aineistosta tehdä useita havaintoja. Lambdaryhmällä meni tehtävään 4 enemmän yrityskertoja. Korrelaatioanalyysin perusteella ne opiskelijat, jotka kokivat tehtävät helpoiksi, käyttivät tehtäviin vähemmän aikaa, käyttivät tehtävää 4 lukuun ottamatta tehtäviin vähemmän yrityskertoja ja myös saivat tehtävistä korkeammat pisteet. Edelleen, tehtäviin enemmän aikaa käyttäneet opiskelijat joutuivat yrittämään tehtäviä 1 ja 2 useammin. Vaikeustasoarvion korreloiminen ajan, yrityskertojen ja pisteiden kanssa on odotettu tulos, joka vahvistaa tulosten luotettavuutta. Sekä kuviossa 12 esitetty Spearmanin rho että kuviossa 13 esitetty Kendallin tau -menetelmä antavat vastaavat tulokset. Pearsonin menetelmä ei sovellu aineiston analyysiin yhtä hyvin, koska aineisto ei noudata normaalijakaumaa.

Correlations

			Ryhmä	Vaikeusarvio	Aika-arvio	Kokonaispisteet	T1 yrityskerrat	T2 yrityskerrat	T3 yrityskerrat	T4 yrityskerrat
Spearman's rho	Ryhmä	Correlation Coefficient	1.000	.028	.014	.081	.065	.057	-.092	-.156 *
		Sig. (2-tailed)	.	.705	.855	.276	.379	.440	.212	.033
		N	187	180	180	183	187	187	187	187
	Vaikeusarvio	Correlation Coefficient	.028	1.000	-.550 **	.255 **	-.230 **	-.223 **	-.194 **	-.119
		Sig. (2-tailed)	.705	.	.000	.001	.002	.003	.009	.113
		N	180	180	178	178	180	180	180	180
	Aika-arvio	Correlation Coefficient	.014	-.550 **	1.000	-.192 *	.185 *	.232 **	.049	.010
		Sig. (2-tailed)	.855	.000	.	.010	.013	.002	.517	.896
		N	180	178	180	178	180	180	180	180
	Kokonaispisteet	Correlation Coefficient	.081	.255 **	-.192 *	1.000	-.084	-.176 *	-.129	-.237 **
		Sig. (2-tailed)	.276	.001	.010	.	.257	.017	.081	.001
		N	183	178	178	183	183	183	183	183
	T1 yrityskerrat	Correlation Coefficient	.065	-.230 **	.185 *	-.084	1.000	.564 **	.346 **	.372 **
		Sig. (2-tailed)	.379	.002	.013	.257	.	.000	.000	.000
		N	187	180	180	183	187	187	187	187
	T2 yrityskerrat	Correlation Coefficient	.057	-.223 **	.232 **	-.176 *	.564 **	1.000	.348 **	.316 **
		Sig. (2-tailed)	.440	.003	.002	.017	.000	.	.000	.000
		N	187	180	180	183	187	187	187	187
	T3 yrityskerrat	Correlation Coefficient	-.092	-.194 **	.049	-.129	.346 **	.348 **	1.000	.409 **
		Sig. (2-tailed)	.212	.009	.517	.081	.000	.000	.	.000
		N	187	180	180	183	187	187	187	187
	T4 yrityskerrat	Correlation Coefficient	-.156 *	-.119	.010	-.237 **	.372 **	.316 **	.409 **	1.000
		Sig. (2-tailed)	.033	.113	.896	.001	.000	.000	.000	.
		N	187	180	180	183	187	187	187	187

*. Correlation is significant at the 0.05 level (2-tailed).

**. Correlation is significant at the 0.01 level (2-tailed).

Kuvio 12. Speamanin rho -menetelmällä tehty korrelaatioanalyysi

Correlations

			Ryhmä	Vaikeusarvio	Aika-arvio	Kokonaispisteet	T1 yrityskerrat	T2 yrityskerrat	T3 yrityskerrat	T4 yrityskerrat
Kendall's tau_b	Ryhmä	Correlation Coefficient	1.000	.025	.012	.079	.057	.051	-.083	-.136 *
		Sig. (2-tailed)	.	.704	.854	.275	.378	.439	.211	.033
		N	187	180	180	183	187	187	187	187
	Vaikeusarvio	Correlation Coefficient	.025	1.000	-.440 **	.218 **	-.180 **	-.176 **	-.156 **	-.088
		Sig. (2-tailed)	.704	.	.000	.001	.002	.003	.009	.120
		N	180	180	178	178	180	180	180	180
	Aika-arvio	Correlation Coefficient	.012	-.440 **	1.000	-.159 *	.140 *	.179 **	.038	.010
		Sig. (2-tailed)	.854	.000	.	.011	.013	.002	.511	.858
		N	180	178	180	178	180	180	180	180
	Kokonaispisteet	Correlation Coefficient	.079	.218 **	-.159 *	1.000	-.073	-.156 *	-.114	-.200 **
		Sig. (2-tailed)	.275	.001	.011	.	.252	.016	.083	.001
		N	183	178	178	183	183	183	183	183
	T1 yrityskerrat	Correlation Coefficient	.057	-.180 **	.140 *	-.073	1.000	.465 **	.280 **	.294 **
		Sig. (2-tailed)	.378	.002	.013	.252	.	.000	.000	.000
		N	187	180	180	183	187	187	187	187
	T2 yrityskerrat	Correlation Coefficient	.051	-.176 **	.179 **	-.156 *	.465 **	1.000	.284 **	.254 **
		Sig. (2-tailed)	.439	.003	.002	.016	.000	.	.000	.000
		N	187	180	180	183	187	187	187	187
	T3 yrityskerrat	Correlation Coefficient	-.083	-.156 **	.038	-.114	.280 **	.284 **	1.000	.334 **
		Sig. (2-tailed)	.211	.009	.511	.083	.000	.000	.	.000
		N	187	180	180	183	187	187	187	187
	T4 yrityskerrat	Correlation Coefficient	-.136 *	-.088	.010	-.200 **	.294 **	.254 **	.334 **	1.000
		Sig. (2-tailed)	.033	.120	.858	.001	.000	.000	.000	.
		N	187	180	180	183	187	187	187	187

*. Correlation is significant at the 0.05 level (2-tailed).

**. Correlation is significant at the 0.01 level (2-tailed).

Kuvio 13. Kendallin tau -menetelmällä tehty korrelaatioanalyysi

5.2 Laadullinen analyysi

Laadullisessa analyysissa kaikkien opiskelijoiden viimeiset palautukset tehtäviin tutkittiin läpi ja niistä etsittiin toistuvia teemoja. Silmukkaryhmän osalta mallivastauksen mukaiseksi vastaukseksi tulkittiin millä tahansa silmukkatyypillä tehty vastaus, joka muilta osin täsmäsi mallivastaukseen. Koodin tyyliä, esimerkiksi rivinvaihtojen käyttöä, ei huomioitu verratessa vastauksia mallivastauksiin.

Tehtävän 1 osalta sekä silmukka- että varsinkin lambdaryhmässä vastaukset olivat yleisesti lähellä mallivastausta. Silmukkaryhmässä variaatiota vastausten välillä esiintyi enemmän. Silmukkaryhmässä yleisin koodia monimutkaistanut rakenne oli ylimääräinen `break`-lause silmukassa ja apumuuttuja, kun mallivastauksessa silmukan sisällä palautetaan funktion tulos suoraan. Neljä opiskelijaa käytti silmukkaratkaisussa `foreach`-lausetta, joka oli tehtävässä epäoptimaalinen ratkaisu. Lambdaryhmässä yleisin virhe oli viiden opiskelijan tekemä. Näiltä opiskelijoilta oli jäänyt huomaamatta, että C#:n `FindIndex`-metodi palauttaa -1, jos ehdon täyttävää alkiota ei löydy. Tällöinkin ohjelma toimi, mutta siinä oli ylimääräinen tarkistus, että onko listassa jo olemassa sopiva alkio. Näiden lisäksi kolme opiskelijaa käytti listaolion funktioita monimutkaisesti, esimerkiksi samassa ratkaisussa `Exists`, `Find` ja `IndexOf` -funktioita.

Tehtävässä 2 oli havaittavissa vastaava teema kuin ensimmäisessä tehtävässä. Mallivastauksen kaltaisia vastauksia oli molemmissa ryhmissä paljon, mutta silmukkaryhmän ratkaisuissa oli enemmän eroja. Lambdaryhmässä 10 opiskelijaa ratkaisi tehtävän `Aggregate`-funktiolla, kun loput käyttivät mallivastauksen mukaisesti `ForEach`ia. Yksitoista lambdaryhmän vastausta sisälsi ylimääräisen tarkistuksen, että lukujen määrä listassa on suurempi kuin nolla. Silmukkaryhmässä 58 opiskelijaa suoritti tehtävän `for`-silmukalla, 26 `foreach`-silmukalla ja kolme `while`-silmukalla. 24 vastausta sisälsi ylimääräisen tarkistuksen lukujen määrälle, ja näistä vastauksista 18 käytti `for`-silmukkaa. Yleisesti oli nähtävissä, että tehtävän tapausta varten optimaalisen `foreach`-silmukan käyttäjillä koodin rakenne oli siistimpi ja luettavampi.

Tehtävä 3 meni molempien ryhmien opiskelijoilta vahvasti. Lambdaryhmän vastaajista 83 ja silmukkaryhmän vastaajista 77 palautti malliratkaisua vastaavan vastauksen. Erilaisia virheitä tekivät vain yksittäiset opiskelijat. Silmukkaryhmästä tehtävään optimaalisinta `foreach`-silmukkaa käytti 48 opiskelijaa, 36 `for`-silmukkaa ja kaksi `while`-silmukkaa.

Tehtävä 4 poikkesi muista tehtävistä sillä, että harvempi vastauksista täsmäsi mallivastaukseen. Lambdaryhmässä vain 14 opiskelijan ratkaisu täsmäsi mallivastaukseen, eli he käyttivät `FindAll`-metodia värjättävien olioiden löytämiseen ja `ForEach`-metodia olioiden värjäämiseen. Yleisin ratkaisu oli 38 opiskelijan `ForEach`in käyttö erillisen värjättyjen olioiden listan kanssa, jolloin tähän listaan lisättiin oliot samalla kun ne värjättiin. Tämä lista lopulta palautettiin. Tämän ratkaisun yleisyyttä voi osin selittää `ForEach`-metodin soveltuminen edelliseen tehtävään: tästä oli viitteitä muutamassa virheellisessä vastauksessa, joissa olioita tuhottiin kuin tehtävässä 3. Toinen malliratkaisua yleisempi ratkaisu, jonka 21 opiskelijaa palautti, oli etsiä värjättävät oliot `ForEach`-metodin yhteydessä ja sitten kaikkien värjäyksen jälkeen hakea nämä erilliseen palautettavaan listaan `FindAll`-metodia käyttämällä. Tämä johtaa loogiseen virheeseen, sillä tällöin palautettu lista sisältää mahdolliset listassa jo ennen värjäämistä olleet siniset suorakulmiot. Tehtävän automaattitarkistinta tehdessä tätä ei tosin huomioitu, minkä vuoksi nämäkin opiskelijat saivat ratkaisustaan täydet pisteet. Silmukkaryhmässä tehtävä 4 meni paremmin: yhteensä 74 opiskelijaa oli ratkaissut tehtävän malliratkaisun mukaisesti. Heistä 44 käytti `foreach`-silmukkaa ja 30 `for`-silmukkaa. Muita yleisiä havaintoja silmukkaryhmän vastauksista ei tehty. Tehtävä 4 oli ainoa, jossa ryhmien suoriutumisen välillä oli tilastollisesti merkitsevä ero, sillä lambdaryhmä käytti tehtävään silmukkaryhmää suuremman määrän yrityskertoja. Mikäli automaattitarkistin olisi huomannut lambdaryhmässä yleisen loogisen virheen, lambdaryhmä olisi saattanut vaatia tehtävään vielä enemmän yrityskertoja.

Opiskelijat pystyivät antamaan tehtävistä ja tutkimuksesta avointa palautetta. Avointa palautetta antoi 42 lambdaryhmän vastaajaa ja 29 silmukkaryhmän vastaajaa. Molemmissa ryhmissä neljä opiskelijaa kehui materiaalin laatua, ja molemmissa

ryhmissä moni koki tehtävät helpoksi. Toisaalta lambdaryhmässä, sekä etenkin silmukkaryhmässä, oli myös tehtäviä ja niiden vaikeustasoa kehuneita opiskelijoita. Silmukkaryhmästä kolme opiskelijaa kritisoi tehtävänantoja epäselviksi. Muuten silmukkaryhmässä ei ilmennyt toistuvaa palautetta.

Lambdaryhmässä yhdeksässä vastauksessa lambda-lausekkeita kuvailtiin käytännöllisiksi ja näiden lisäksi useassa vastauksessa myös muuten positiivisin adjektiivein, esimerkiksi sanoilla "helppo", "tehokas", ja "yksinkertainen". Viidessä vastauksessa kehuttiin lambda-ratkaisujen silmukoita lyhyempää syntaksia. Monessa vastauksessa, myös käytännöllisyyttä tai syntaksia kehuvissa, toisaalta todettiin lambda-lausekkeiden olevan vaikeita aloitteleville ohjelmoijille oppia. Osa opiskelijoista arvioi, ettei olisi selviytynyt tehtävistä, jos ne olisivat tulleet vastaan aiemmin kurssin aikana. Lambda-lausekkeet saivat kokonaisuutena moninkertaisesti enemmän positiivista kuin negatiivista palautetta. Kolme opiskelijaa totesi lambda-lausekkeiden opettamisen kurssilla hyväksi asiaksi.

5.3 Vääristävät tekijät

Tutkimukseen osallistuneilla opiskelijoilla oli kurssilta valmiiksi kokemusta silmukoista, mutta ei lambda-lausekkeista. Vaikka lambda-lausekkeiden opettamista varten tuotettiin materiaalia, valmis kokemus on voinut kallistaa silmukkaryhmän tuloksia paremmiksi suhteutettuna tilanteeseen, jossa sekä silmukka- että lambdaryhmän opiskelijat olisivat saaneet yhtä paljon opetusta vastaavista työkaluistaan. Pidempi tutkimus, jossa opetukseen olisi panostettu enemmän, olisi vaatinut enemmän opetusmateriaalin ja tehtävien tekemistä sekä vastausten käsittelyä kuin pro gradu -tutkimuksen yhteydessä oli mahdollista.

Tutkimuksessa luotettiin opiskelijoiden omiin aika-arvioihin, eikä opiskelijoiden tehtäviin käyttämää aikaa mitattu. Suuren vastaajajoukon vuoksi tämä tuskin on vaikuttanut merkittävästi, kun ryhmiä katsotaan kokonaisuutena. Koska tehtävät olivat kurssin pakollisia viikkotehtäviä, niihin sai myös pyytää apua, minkä takia analyysissa käytettiin vastaavasti opiskelijoiden itse antamaa arviota tehtävien vaikeus-

tasosta. Opiskelijoiden todellista pärjäämistä tehtävissä ei siten voitu kunnolla mitata. Opiskelijoiden TIM-järjestelmään palauttamia yrityskertoja mitattiin, mutta osa opiskelijoista on todennäköisesti tehnyt tehtävät Visual Studiolla, eikä heidän yrityskertojaan ole voitu mitata. Suuren vastaajajoukon ja arvontaan perustuneen ryhmävalinnan vuoksi on epätodennäköistä, että yhteen ryhmään olisi päätynyt toista ryhmää merkittävästi enemmän tehtävät Visual Studiolla tehneitä opiskelijoita. Jotta saatavilla ollut ohjaajien apu olisi vaikuttanut tutkimukseen ja erityisesti tehtävien yrityskertoihin mahdollisimman vähän, opiskelijoita kehotettiin yrittämään tehtävien tekoa ensin itsenäisesti, ja kysymään apua vasta, jos tehtävien tekeminen itsenäisesti ei onnistunut.

Tehtävässä 4 tapahtui lambdaryhmässä yleinen looginen virhe, jota automaattitarkistin ei havainnut (ks. luku 5.2). Tämä on voinut madaltaa lambdaryhmän opiskelijoiden yrityskertoja tehtävässä verrattuna tilanteeseen, jossa automaattitarkistin olisi tunnistanut tämän loogisen virheen.

Useampi opiskelija molemmista ryhmistä sanoi avoimessa palautteessa, että tehtävät olivat helppoja. Kun lisäksi ottaa huomioon opiskelijoiden hyvän suoriutumisen tehtävistä, on mahdollista, että tehtävät todella olivat liian helppoja tuodakseen esille mahdollisia eroja ryhmien välillä. Vaikka tämä saattaa haitata tutkimuksen hyödyllisyyttä kokonaisuutena, se ei kuitenkaan tee tuloksista tutkimuksessa käytettyjen tehtävien kanssa vähemmän luotettavia. Tutkimusta olisi ollut luontaista jatkaa pidemmälle vaikeammilla tehtävillä, mutta pro gradu -tutkimuksen asettamat rajoitteet työmäärälle eivät tätä mahdollistaneet.

6 Johtopäätökset ja pohdinta

Tutkimuksessa lambda- ja silmukkaryhmien suoriutumisen välillä ei yhtä tehtävää lukuun ottamatta havaittu tilastollisesti merkitseviä eroja. Uesbeckin ym. (2016) tutkimuksessa C++-kielen lambda-lausekkeita hyödyntäneet osallistujat suoriutuivat selvästi heikommin kuin iteraattoreita hyödyntäneet osallistujat. Erityisesti näin oli niiden osallistujien keskuudessa, joilla oli vähän kokemusta ohjelmoinnista. Tässä tutkimuksessa vastaavaa eroa ei ole havaittavissa. Moni opiskelija näki lambda-lausekkeissa hyötyjä, mikä tukee sekä lambda-lausekkeiden käyttämistä koodissa että lambda-lausekkeiden opettamista ohjelmointikursseilla. Yhtenä syynä tutkimusten tulosten eroavaisuuteen voi nähdä kielivalinnan. Lisäksi tässä tutkimuksessa käytetyt tehtävät olivat helpompia. Uesbeckin ym. (2016) tutkimuksessa lambda-lausekkeita hyödyntämätön ryhmä myös käytti iteraattoreita silmukoiden sijaan.

Laadullisen analyysin tuloksissa lambda-lausekkeet saivat opiskelijoilta kehuja käytännöllisyydestä ja ytimekkäästä syntaksistaan. Tämä tulos sopii yhteen Mazinanian ym. (2017) tekemän havainnon kanssa, että kehittäjät käyttävät lambda-lausekkeita koodinsa lyhentämiseen ja koodin syntaksin yksinkertaistamiseen. Lambda-lausekkeet on monissa kielissä osaltaan suunniteltu lyhentämään koodia (ks. luku 2.4). Opiskelijoiden antaman palautteen perusteella C#-kielen lambda-lausekkeet onnistuvat tässä tavoitteessa.

Sorva ja Vihavainen (2016) arvelevat anonyymien funktioiden olevan liian haastavia opiskelijoille CS1-kurssilla. Tätä näkemystä tukee opiskelijoiden antama palaute, jossa lambda-lausekkeiden todettiin olevan vaikeita oppia aloitteleville ohjelmoijille. Toisaalta lambda-lausekkeita käyttäneet opiskelijat suoriutuivat koetehtävistä hyvin. Tämän tutkimuksen perusteella anonyymit funktiot eivät siten ole opiskelijoille liian vaikeita oppia, jos ne opetetaan myöhäisessä vaiheessa CS1-kurssia.

Verrattuna erilaisia paradigmoja hyödyntäneisiin ja paradigmoja toisiaan verranneisiin tutkimuksiin, kuten Fisler (2014), Fisler, Krishnamurthi ja Siegmund (2016) ja Seppälä ym. (2015), tämän tutkimuksen tulokset ei tue näissä tutkimuksissa ha-

vaittuja eroja funktionaalisen ja imperatiivisen paradigman välillä. On toki huomattava, että lambda-lausekkeiden käytöstä huolimatta C#-kielen laskentamalli pohjautuu edelleen Turingin koneeseen Churchin lambda-laskennan sijaan. Siten lambda-lausekkeet eivät täysin vaihda C#-kielen paradigmaa.

Silmukoiden osalta laadullisen analyysin tulokset tukivat Qianin ja Lehmanin (2017) esille tuomaa huomiota, että opiskelijoiden on haastavaa valita eri tilanteisiin optimaalinen silmukkatyyppi. Eri silmukkatyyppien vahvuuksien opettamiseen panostaminen voisi siten auttaa opiskelijoita kirjoittamaan selkeämpää koodia.

Jatkotutkimuksessa voisi yrittää suorittaa tätä tutkimusta vastaavan kokeen vaikeammilla tehtävillä sekä kouluttamalla lambda-lausekkeita käyttävät osallistujat niin, että heidän kokemuksensa vastaisi silmukkaryhmän osallistujia. Tämä kertoisi, tulisiko C#-kielen kanssa lambda-lausekkeiden ja silmukoiden välillä eroja ilmi haastavammissa tehtävissä, sillä tämän tutkimuksen verrattain yksinkertaisissa tehtävissä niitä ei havaittu. Lisäksi tehtävien tarkistimia tulisi miettiä tarkemmin kaikkien loogisten virheiden havaitsemiseksi. Yksi vaihtoehto jatkotutkimuksessa on myös verrata C#-kielen lambda-lausekkeita silmukoiden sijasta iteraattoreihin, kuten Uesbeck ym. (2016) tekivät C++-kielen yhteydessä.

Tämä tutkimus osaltaan täyttää lambda-lausekkeiden käytön ja opetuksen tutkimukseen liittyvää aukkoa. Lambda-lausekkeet ovat tosin vain pieni osa nykyaikaisia ohjelmointikieliä. Kuten luvussa 2.2 todettiin, ohjelmointikieliin lisätään jatkuvasti erilaisia ominaisuuksia, joita voisi tutkia ihmislähtöisesti niin teollisesta hyötynäkökulmasta kuin myös opetuksellisesta näkökulmasta.

Lähteet

Altadmri, Amjad, ja Neil CC Brown. 2015. "37 million compilations: Investigating novice programming mistakes in large-scale student data". Teoksessa *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 522–527. ACM.

Backus, John. 1978. "The history of Fortran I, II, and III". *ACM Sigplan Notices* 13 (8): 165–180.

Barendregt, Henk P, ja Erik Barendsen. 1984. "Introduction to lambda calculus".

BlueJ. 2020. "Blackbox", 5. marraskuuta 2020. https://bluej.org/blackbox/.

Bruce, Kim B, Andrea Danyluk ja Thomas Murtagh. 2005. "Why structural recursion should be taught before arrays in CS 1". *ACM SIGCSE Bulletin* 37 (1): 246–250.

Chen, Tzu-Yi, Gary Lewandowski, Robert McCartney, Kate Sanders ja Beth Simon. 2007. "Commonsense computing: using student sorting abilities to improve instruction". Teoksessa *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, 276–280.

Church, Alonzo. 1932. "A set of postulates for the foundation of logic". *Annals of mathematics*, 346–366.

Davies, Stephen, Jennifer A Polack-Wahl ja Karen Anewalt. 2011. "A snapshot of current practices in teaching the introductory programming sequence". Teoksessa *Proceedings of the 42nd ACM technical symposium on Computer science education*, 625–630.

Deaton, Angus, ja Nancy Cartwright. 2018. "Understanding and misunderstanding randomized controlled trials". *Social Science & Medicine* 210:2–21.

Dougherty, John P, ja David G Wonnacott. 2005. "Use and assessment of a rigorous approach to CS1". *ACM SIGCSE Bulletin* 37 (1): 251–255.

Fisler, Kathi. 2014. "The recurring rainfall problem". Teoksessa *Proceedings of the tenth annual conference on International computing education research*, 35–42.

Fisler, Kathi, Shriram Krishnamurthi ja Janet Siegmund. 2016. “Modernizing plan-composition studies”. Teoksessa *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 211–216.

Floyd, Robert W. 2007. “The paradigms of programming”. Teoksessa *ACM Turing award lectures*, 1978.

Green, Thomas R. G., ja Marian Petre. 1996. “Usability analysis of visual programming environments: a ‘cognitive dimensions’ framework”. *Journal of Visual Languages & Computing* 7 (2): 131–174.

Hanenberg, Stefan. 2010. “Faith, hope, and love: an essay on software science’s neglect of human factors”. Teoksessa *ACM SIGPLAN Notices*, 45:933–946. 10. ACM.

———. 2015. “Empirical, Human-Centered Evaluation of Programming and Programming Language Constructs: Controlled Experiments”. Teoksessa *International Summer School on Generative and Transformational Techniques in Software Engineering*, 45–72. Springer.

Jyväskylän yliopisto. 2020. “Ohjelmointi 1 - Demo 9 (tutkimus)”, 9. maaliskuuta 2020. Viitattu 2. helmikuuta 2021. https://tim.jyu.fi/view/kurssit/tie/ohj1/2020k/demot/demo9-tutkimus.

Kaijanaho, Antti-Juhani. 2015. “Evidence-based programming language design: a philosophical and methodological exploration”. *Jyväskylä studies in computing*, numero 222.

Kereki, Inés Friss de, ja Alejandro Adorjan. 2020. “Flipped classroom in a CS1 course”. Teoksessa *2020 IEEE Global Engineering Education Conference (EDUCON)*, 110–114. IEEE.

Ko, Andrew J, Thomas D Latoza ja Margaret M Burnett. 2015. “A practical guide to controlled experiments of software engineering tools with human participants”. *Empirical Software Engineering* 20 (1): 110–141.

Krishnamurthi, Shriram, ja Kathi Fisler. 2019. “Programming paradigms and beyond”. *The Cambridge Handbook of Computing Education Research* 37.

Lakanen, Antti-Jussi, Vesa Lappalainen ja Ville Isomöttönen. 2015. “Revisiting rainfall to explore exam questions and performance on CS1”. Teoksessa *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, 40–49.

Lévénez, Éric. 2021. “Computer Languages History”, 1. tammikuuta 2021. https://www.levenez.com/lang/.

Luxton-Reilly, Andrew, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard ja Claudia Szabo. 2018. “Introductory programming: a systematic literature review”. Teoksessa *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, 55–106.

Markstrum, Shane. 2010. “Staking claims: a history of programming language design claims and evidence: a positional work in progress”. Teoksessa *Evaluation and Usability of Programming Languages and Tools*, 7. ACM.

Mason, Raina, Tom Crick, James H Davenport ja Ellen Murphy. 2018a. “Language choice in introductory programming courses at Australasian and UK universities”. Teoksessa *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 852–857.

———. 2018b. “Language choice in introductory programming courses at Australasian and UK universities”, 24. helmikuuta 2018. https://people.bath.ac.uk/masjhd/Slides/SIGCSE2018-Slides.pdf.

Mazinanian, Davood, Ameya Ketkar, Nikolaos Tsantalis ja Danny Dig. 2017. “Understanding the use of lambda expressions in Java”. *Proceedings of the ACM on Programming Languages* 1 (OOPSLA): 1–31.

McCarthy, John. 1978. “History of LISP”. Teoksessa *History of programming languages*, 173–185.

Microsoft. 2019a. “Lambda expressions - C# Programming Guide”, 29. heinäkuuta 2019. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions.

Microsoft. 2019b. “Lambda Expressions in C | Microsoft Docs”, 7. toukokuuta 2019. https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp.

———. 2019c. “The history of C#”, 18. lokakuuta 2019. https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history.

———. 2019d. “What’s new in C# 8.0”, 18. lokakuuta 2019. https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8.

———. 2020. “List<T>.ForEach(Action<T>) Method (System.Collections.Generic) | Microsoft Docs”, 4. marraskuuta 2020. https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.foreach.

Myers, Brad A, Andreas Stefik, Stefan Hanenberg, Antti-Juhani Kaijanaho, Margaret Burnett, Franklyn Turbak ja Philip Wadler. 2016. “Usability of programming languages: Special interest group (sig) meeting at CHI 2016”. Teoksessa *Proceedings of the 2016 chi conference extended abstracts on human factors in computing systems*, 1104–1107. ACM.

Nielebock, Sebastian, Robert Heumüller ja Frank Ortmeier. 2019. “Programmers do not favor lambda expressions for concurrent object-oriented code”. *Empirical Software Engineering* 24 (1): 103–138.

Oracle. 2014a. “Iterable (Java Platform SE 8)”, 18. maaliskuuta 2014. https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html.

———. 2014b. “Java SE 8: Lambda Quick Start”, 18. maaliskuuta 2014. https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html.

Python Software Foundation. 2020. “Classes - Python 3.9.0 documentation”. https://docs.python.org/3/tutorial/classes.html.

Qian, Yizhou, ja James Lehman. 2017. “Students’ misconceptions and other difficulties in introductory programming: A literature review”. *ACM Transactions on Computing Education (TOCE)* 18 (1): 1–24.

Roberts, Eric S. 1995. “Loop exits and structured programming: reopening the debate”. *ACM SIGCSE Bulletin* 27 (1): 268–272.

Seppälä, Otto, Petri Ihantola, Essi Isohanni, Juha Sorva ja Arto Vihavainen. 2015. “Do we know how difficult the rainfall problem is?” Teoksessa *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, 87–96.

Soloway, Elliot. 1986. “Learning to program = learning to construct mechanisms and explanations”. *Communications of the ACM* 29 (9): 850–858.

Soloway, Elliot, Jeffrey Bonar ja Kate Ehrlich. 1983. “Cognitive strategies and looping constructs: An empirical study”. *Communications of the ACM* 26 (11): 853–860.

Sorva, Juha, ja Arto Vihavainen. 2016. “Break statement considered”. *ACM Inroads* 7 (3): 36–41.

Stanley, Kenneth. 2007. “Design of randomized controlled trials”. *Circulation* 115 (9): 1164–1169.

Statista. 2019. “Software market revenue worldwide* from 2016 to 2021 (in billion U.S. Dollars)*”, 16. lokakuuta 2019. https://www.statista.com/statistics/963597/software-revenue-worldwide/.

Stefik, Andreas, ja Stefan Hanenberg. 2014. “The programming language wars: Questions and responsibilities for the programming language community”. Teoksessa *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 283–299. ACM.

———. 2017. “Methodological irregularities in programming-language research”. *Computer* 50 (8): 60–63.

Tsantalis, Nikolaos, Davood Mazinanian ja Shahriar Rostami. 2017. “Clone refactoring with lambda expressions”. Teoksessa *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 60–70. IEEE.

Turing, Alan Mathison. 1936. “On computable numbers, with an application to the Entscheidungsproblem”. *J. of Math* 58 (345-363): 5.

Turing, Alan Mathison. 1937. “Computability and λ-definability”. *The Journal of Symbolic Logic* 2 (4): 153–163.

Uesbeck, Phillip Merlin, Andreas Stefik, Stefan Hanenberg, Jan Pedersen ja Patrick Daleiden. 2016. “An empirical study on the impact of C++ lambdas and programmer experience”. Teoksessa *Proceedings of the 38th International Conference on Software Engineering*, 760–771. ACM.

Van Roy, Peter, ym. 2009. “Programming paradigms for dummies: What every programmer should know”. *New computational paradigms for computer music* 104:616–621.

Watt, David A, ja Steven Wong. 1990. “Programming Languages”. *Concepts and Paradigms Prentice Hall.*

———. 2012. “Programming Languages”. http://www.dcs.gla.ac.uk/~daw/teaching/PL3/Slides/00.Preamble.pdf.

Wegner, Peter, ja Dina Goldin. 2003. “Computation beyond Turing machines”. *Communications of the ACM* 46 (4): 100–102.

Wexelblat, Richard L. 2014. *History of programming languages.* Academic Press.

Zuhud, Daeng Ahmad Zuhri, Nurazzah Abdul Rahman ja Marina Ismail. 2013. “A preliminary analysis on the shift of programming paradigms”. Teoksessa *2013 5th International Conference on Information and Communication Technology for the Muslim World (ICT4M)*, 1–5. IEEE.

Liitteet

A Tietosuojaseloste

1. Tutkimuksessa "Lambda-lausekkeiden hyödyllisyys C#-kielessä" käsiteltävät henkilötiedot

Tutkimuksessa Sinusta kerätään seuraavia henkilötietoja: nimi, JYU-tunnus, tehtävien vastaukset jokaiselta yrityskerralta, yrityskertojen pistemäärät, yrityskertojen oikeellisuudet, antamasi aika-arvio, antamasi arvio tehtävien vaikeustasosta numeerisena arvona sekä avoimena vastauksena.

Tämä seloste on toimitettu tutkittavalle tutkimuksen verkkosivulla tutkimukseen liittyvien tehtävien yhteydessä. Tutkittavalle on annettu kyselylomakkeella suora linkki tähän selosteeseen.

2. Henkilötietojen käsittelyn oikeudellinen peruste tutkimuksessa

Tutkittavan suostumus (EU 679/2016 6.1 a)

Henkilötietojen siirto EU/ETA ulkopuolelle

Tutkimuksessa tietojasi ei siirretä EU/ETA -alueen ulkopuolelle.

Henkilötietojen suojaaminen

Henkilötietojen käsittely tässä tutkimuksessa perustuu asianmukaiseen tutkimussuunnitelmaan ja tutkimuksella on vastuuhenkilö. Henkilötietojasi käytetään ja luovutetaan vain historiallista/ tieteellistä tutkimusta taikka muuta yhteensopivaa tarkoitusta varten (tilastointi) sekä muutoinkin toimitaan niin, että Sinua koskevat tiedot eivät paljastu ulkopuolisille.

Tunnistettavuuden poistaminen

Suorat tunnistetiedot poistetaan suojatoimena aineiston perustamisvaiheessa (pseudonymisoitu aineisto, jolloin tunnistettavuuteen voidaan palata koodin tai vastaa-

van tiedon avulla ja aineistoon voidaan yhdistää uusia tietoja). Tunnistettavuuteen ei palata tutkimuksen yhteydessä, mutta TIM-järjestelmä säilyttää alkuperäiset vastaukset, joiden avulla tunnistettavuuteen olisi teknisesti mahdollista päästä.

Tutkimuksessa käsiteltävät henkilötiedot suojataan

TIM-järjestelmän yhteydessä käyttäjätunnuksella, salasanalla ja käytön rekisteröinnillä. Tutkimuksen analyysivaiheessa kulunvalvonnalla.

Henkilötietojen käsittely tutkimuksen *päättymisen jälkeen*

Tutkimusrekisteri anonymisoidaan eli kaikki tunnistetiedot poistetaan täydellisesti, jotta paluuta tunnisteelliseen henkilötietoon ei ole eikä aineistoon voida yhdistää uusia tietoja.

Rekisterinpitäjä(t) ja tutkimuksen tekijät

Rekisterinpitäjä on se, joka yksin tai yhdessä toisten kanssa määrittelee henkilötietojen käsittelyn tavoitteet ja keinot, organisaatio(t) tai henkilö(t) sekä vastaa henkilötietojen käsittelyn lainmukaisuudesta.

Tämän tutkimuksen rekisterinpitäjä ja tutkimuksen suorittaja on Rami Pasanen, rami.m.pasanen@student.jyu.fi

Henkilötietojen käsittelijänä tutkimuksessa on myös Jyväskylän yliopiston TIM-järjestelmä. Linkki TIM-järjestelmän rekisteriselosteeseen on sivun oikeassa alalaidassa.

Rekisteröidyn oikeudet

Suostumuksen peruuttaminen (tietosuoja-asetuksen 7 artikla)
Sinulla on oikeus peruuttaa antamasi suostumus ottamalla yhteytetä rekisterinpitäjään. Suostumuksen peruuttaminen ei vaikuta suostumuksen perusteella ennen sen peruuttamista suoritetun käsittelyn lainmukaisuuteen.

Oikeus saada pääsy tietoihin (tietosuoja-asetuksen 15 artikla) Sinulla on oikeus saada tieto siitä, käsitelläänkö henkilötietojasi ja mitä henkilötietojasi käsitellään. Voit myös halutessasi pyytää jäljennöksen käsiteltävistä henkilötiedoista.

Oikeus tietojen poistamiseen (tietosuoja-asetuksen 17 artikla) Sinulla on oikeus vaatia henkilötietojesi poistamista tietyissä tapauksissa. Oikeutta tietojen poistamiseen ei kuitenkaan ole, jos tietojen poistaminen estää tai vaikeuttaa suuresti käsittelyn tarkoituksen toteutumista tieteellisessä tutkimuksessa.

Oikeus käsittelyn rajoittamiseen (tietosuoja-asetuksen 18 artikla) Sinulla on oikeus henkilötietojesi käsittelyn rajoittamiseen tietyissä tilanteissa kuten, jos kiistät henkilötietojesi paikkansapitävyyden.

Oikeus siirtää tiedot järjestelmästä toiseen (tietosuoja-asetuksen 20 artikla) Sinulla on oikeus saada toimittamasi henkilötiedot jäsennellyssä, yleisesti käytetyssä ja koneellisesti luettavassa muodossa, ja oikeus siirtää kyseiset tiedot toiselle rekisterinpitäjälle, jos se on mahdollista ja käsittely suoritetaan automaattisesti.

Oikeuksista poikkeaminen Tässä kuvatuista oikeuksista saatetaan tietyissä yksittäistapauksissa poiketa tietosuoja-asetuksessa ja Suomen tietosuojalaissa säädetyillä perusteilla siltä osin, kuin oikeudet estävät tieteellisen tai historiallisen tutkimustarkoituksen tai tilastollisen tarkoituksen saavuttamisen tai vaikeuttavat sitä suuresti. Tarvetta poiketa oikeuksista arvioidaan aina tapauskohtaisesti.

Profilointi ja automatisoitu päätöksenteko Tutkimuksessa henkilötietojasi ei käytetä automaattiseen päätöksentekoon. Tutkimuksessa henkilötietojen käsittelyn tarkoituksena ei ole henkilökohtaisten ominaisuuksiesi arviointi, ts. profilointi vaan henkilötietojasi ja ominaisuuksia arvioidaan laajemman tieteellisen tutkimuksen näkökulmasta.

Rekisteröidyn oikeuksien toteuttaminen Jos sinulla on kysyttävää rekisteröidyn oikeuksista, voit olla yhteydessä yliopiston tietosuojavastaavaan. Kaikki oikeuksien toteuttamista koskevat pyynnöt toimitetaan Jyväskylän yliopiston kirjaamoon. Kirjaamo ja arkisto, PL 35 (C), 40014 Jyväskylän yliopisto, puh. 040 805 3472, e-mail: kirjaamo(at)jyu.fi. Käyntiosoite: Seminaarinkatu 15 C-rakennus (Yliopiston päärakennus, 1. krs), huone C 140.

Tietoturvaloukkauksesta tai sen epäilystä ilmoittaminen Jyväskylän yliopistolle

https://www.jyu.fi/fi/yliopisto/tietosuojailmoitus/ilmoita-tietoturvaloukkauksesta

Sinulla on oikeus tehdä valitus erityisesti vakinaisen asuin- tai työpaikkasi sijainnin mukaiselle valvontaviranomaiselle, mikäli katsot, että henkilötietojen käsittelyssä rikotaan EU:n yleistä tietosuoja-asetusta (EU) 2016/679. Suomessa valvontaviranomainen on tietosuojavaltuutettu.

Tietosuojavaltuutetun toimisto Ratapihantie 9, 6. krs, 00520 Helsinki, PL 800, 00521 Helsinki Puhelinvaihde: 029 566 6700 Sähköposti (kirjaamo): tietosuoja@om.fi

B Lambda-lausekkeiden opettamista varten tehty materiaali

23.3 Anonyymit funktiot (lambda-lausekkeet)

Listoja voidaan käsitellä silmukoiden lisäksi myös `List<T>`-luokan metodeilla. Monet näistä metodeista ottavat parametrina aliohjelman, jonka avulla listaa käsitellään. Tällaisia metodeja ovat esimerkiksi `Find`, `FindIndex`, `Exists`, `FindAll` ja `ForEach`. Näille metodeille voidaan antaa argumenttina aliohjelma metodin kutsun kaarisulkeiden sisään. Valmiilla metodeilla käsittely silmukoiden sijaan mahdollistaa usein vastaavan asian tekemisen huomattavasti pienemmällä määrällä koodia.

Usein helpoin keino antaa näille metodeille parametreja on tehdä aliohjelmat *lambda-lausekkeina*, jossa parametrina annettava aliohjelma määritellään nimettömästi suoraan parametrilausekkeen sisään. Esimerkiksi jos meillä on lista peliolioita (`List<GameObject>`), jonka nimi on `lista`, seuraava kutsu etsisi listasta ensimmäisen olion, joka on ympyrän muotoinen, ja asettaisi sen muuttujaan `pallo`:

```
GameObject pallo = lista.Find(
  olio => olio.Shape == Shape.Circle);
```

Ylläolevassa koodissa kohta `olio => olio.Shape == Shape.Circle` on lambda-lausekkeella toteutettu anonyymi funktio, joka ottaa yhden parametrin (`olio`). Lambda-lausekkeessa parametrit määritellään ennen nuolta `=>`. Funktio palauttaa lausekkeen `olio.Shape == Shape.Circle` arvon, eli `true` mikäli parametri-

na annetun olion muoto on ympyrä, ja muuten `false`. Listan `Find`-metodi suorittaa tämän sille lambda-lausekkeella parametrina annetun aliohjelman jokaiselle listan alkiolle, kunnes löytyy alkio, jolle lambda-lausekkeella määritelty aliohjelma palauttaa `true`.

Vastaava koodi silmukalla tehtynä olisi seuraavanlainen:

```
GameObject pallo;
foreach (GameObject olio in lista)
{
    if (olio.Shape == Shape.Circle)
    {
        pallo = olio;
        break;
    }
}
```

Lambda-lausekkeet käyttäytyvät kuin normaalit aliohjelmat ja funktiot, mutta niillä ei ole nimeä ja niihin ei voi viitata muualla koodissa. Tosin lambda-lausekkeen voi sijoittaa muuttujaan ja tätä kautta sitä voi tarvittaessa käyttää muualla koodissa.

Seuraavaksi esitellään olennaisia `List<T>` luokan metodeja listojen käsittelyyn esimerkkien kera.

23.3.1 `Find`

Kuten edellä mainittiin, listan `Find`-metodi ottaa parametrina aliohjelman, joka palauttaa `bool`-arvon. `Find` palauttaa listan ensimmäisen alkion, jolle annettu aliohjelma palauttaa `true`. Käytännössä siis `Find`-metodille annetaan parametriksi ehto, ja se etsii listasta ehdon täyttävän alkion. Mikäli ehdon täyttävää alkiota ei löydy, niin `Find` palauttaa arvon `null`. Esimerkki `Find`-metodin käytöstä annettiin edellisessä luvussa.

Find-metodin dokumentaatio MSDN:ssä

23.3.2 `FindIndex`

`FindIndex`-metodi toimii kuten edellä mainittu `Find`-metodi, mutta se palauttaa itse alkion sijaan indeksin. Esimerkiksi seuraava aliohjelma ottaa vastaan listan merkkijonoja, etsii listasta ensimmäisen sellaisen merkkijonon, jonka pituus on enemmän kuin 5 merkkiä, ja palauttaa sen indeksin.

```
public static int EtsiMerkkijononIndeksi(List<string> lista)
{
    return lista.FindIndex(jono => jono.Length > 5);
}
```

Vastaava aliohjelma toteutettuna silmukalla näyttäisi seuraavalta:

```
public static int EtsiMerkkijononIndeksi(List<string> lista)
{
    for (int i = 0; i < lista.Count; i++)
    {
        if (lista[i].Length > 5)
            return i;
    }

    return -1;
}
```

FindIndex-metodin dokumentaatio MSDN:ssä

23.3.3 `Exists`

`Exists`-metodi tarkistaa, löytyykö listasta tietyn ehdon täyttävää alkiota. Mikäli alkio löytyy, `Exists` palauttaa `true`, muuten `false`. Esimerkiksi seuraava aliohjelma tarkistaa, onko kokonaislukulistassa (`List<int>`) olemassa lukua, joka on suurempi kuin 10.

```
public static bool OnkoSuurempaaKuin10(List<int> lista)
{
    return lista.Exists(luku => luku > 10);
}
```

Vastaava aliohjelma toteutettuna silmukalla voisi näyttää esimerkiksi seuraavalta:

```
public static bool OnkoSuurempaaKuin10(List<int> lista)
{
    foreach (int luku in lista)
    {
        if (luku > 10)
            return true;
    }

    return false;
}
```

Exists-metodin dokumentaation MSDN:ssä

23.3.4 `FindAll`

`FindAll`-metodi toimii kuten `Find`-metodi, mutta palauttaa listan, joka sisältää kaikki ne alkiot, jotka täyttävät annetun ehdon, kun `Find` palauttaa alkioista vain ensimmäisen. `FindAll` sisällyttää tulokset uuteen listaan, jonka se palauttaa. Esimerkiksi seuraava aliohjelma etsii ja palauttaa pelioliolistasta (`List<GameObject>`) kaikki punaiset suorakulmiot. Esimerkki näyttää samalla, miten `Find`- ja `FindAll`-metodille annettava parametri voi tarkistaa useamman ehdon.

```
public static List<GameObject> HaePunaisetSuorakulmiot(
    List<GameObject> lista)
{
    return lista.FindAll(olio => olio.Shape == Shape.Rectangle
        && olio.Color == Color.Red);
```

```
}
```

Vastaava aliohjelma toteutettuna silmukalla näyttäisi seuraavalta:

```
public static List<GameObject> HaePunaisetSuorakulmiot(
    List<GameObject> lista)
{
    List<GameObject> tulokset = new List<GameObject>();

    foreach (GameObject olio in lista)
    {
        if (olio.Shape == Shape.Rectangle
            && olio.Color == Color.Red)
        {
            tulokset.Add(olio);
        }
    }

    return tulokset;
}
```

FindAll-metodin dokumentaatio MSDN:ssä

23.3.5 `ForEach` ja usean lauseen sisältävät anonyymit funktiot

Listan `ForEach`-metodilla (älä sekoita `foreach`-silmukkaan) pystyy suorittamaan jonkin aliohjelman listan jokaiselle alkiolle. Esimerkiksi seuraava aliohjelma vaihtaa kaikkien korkeudeltaan 100 ylittävien peliolioiden värin keltaiseksi ja lyö niitä ylöspäin.

```
public static void VaihdaVari(List<GameObject> lista)
{
    lista.ForEach(olio =>
    {
```

```
        if (olio.Height > 100.0)
        {
            olio.Color = Color.Yellow;
            olio.Hit(new Vector(0.0, 100.0));
        }
    });
}
```

Useimmat lambda-lausekkeilla tehdyt anonyymit funktiot ovat yksinkertaisia ja sisältävät vain yhden lauseen tai lausekkeen. Tällöin nuolen oikealle puolelle tuleva funktion toteutus ei tarvitse aaltosulkuja `{ }` koodinsa ympärille kuin tavalliset aliohjelmat. Yllä olevan esimerkin mukaisesti lambda-lausekkeilla tehdyt aliohjelmat voivat tosin sisältää useammankin lauseen. Tällöin koodin ympärille tulee aaltosulut vastaavasti kuin tavallistenkin aliohjelmien ympärille. Lambda-funktiossa voi myös käyttää kaikkia tavallisia C#-kielen ominaisuuksia, kuten ehtolauseita.

Vastaava esimerkki silmukoilla toteutettuna:

```
public static void VaihdaVari(List<GameObject> lista)
{
    foreach (GameObject olio in lista)
    {
        if (olio.Height > 100.0)
        {
            olio.Color = Color.Yellow;
            olio.Hit(new Vector(0.0, 100.0));
        }
    }
}
```

Lambda-funktiot voivat myös muokata itsensä ulkopuolella määriteltyjä paikallisia muuttujia. Esimerkiksi kokonaislukulistan alkioiden yhteenlasku toteutettaisiin seuraavasti:

```
public static int Summaa(List<int> luvut)
{
    int summa = 0;
    luvut.ForEach(luku => summa += luku);
    return summa;
}
```

ForEach-metodin dokumentaatio MSDN:ssä

Mikäli lambda-lausekkeet herättävät enemmänkin kiinnostusta, niihin voi tutustua syvemmin MSDN:ssä.