Ronja Lindholm

Kognitiivinen kuorma ohjelmoinnin oppimisessa

Tietotekniikan kandidaatintutkielma

9. kesäkuuta 2016

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Ronja Lindholm

Yhteystiedot: ronja.lindholm@gmail.com

Ohjaaja: Hannakaisa Isomäki

Työn nimi: Kognitiivinen kuorma ohjelmoinnin oppimisessa

Title in English: Cognitive load in learning programming

Työ: Kandidaatintutkielma

Sivumäärä: 31+0

Tiivistelmä: Ohjelmoinnin oppiminen on kognitiivisesti kuormittavaa. Kuorman hallintaan on löydetty mahdollisia hallintakeinoja, kuten tehtävätyyppien sovittaminen oppilaan taitotasolle, opetusmateriaalin esittäminen kuormaa vähentävästi ja tyypillisten vaikeuksien huomioiminen. Tässä kirjallisuuskatsauksessa eritellään näitä keinoja ja sovelletaan tietoa käytännön ohjelmointikursseihin.

Avainsanat: Ohjelmointi, oppiminen, kognitiivinen kuorma

Abstract: Learning programming happens to have high cognitive load on the learner. Studies have found multiple ways to manage this cognitive load. These ways are: providing right kind of task based on learners level, presenting study material in a load-reducing way and taking into consideration typical difficulties that students have while learning programming. This literature review defines these ways and applies this information to programming courses.

Keywords: Programming, learning, cognitive load

Kuviot

Kuvio 1. Malliesimerkki 11
Kuvio 2. Häiventyvä esimerkk 14
Kuvio 3. Tehtavan jakaminen osiin 18
Kuvio 4. Malliesimerkin osittaminen ja nimeäminen 19
Kuvio 5. Koodin mahdollistettu piilottaminen 21

Sisältö

1 JOHDANTO 1

2 OHJELMOINNIN OPPIMINEN JA KOGNITIIVINEN KUORMA 3
2.1 Ohjelmointi 3
2.2 Ohjelmoinnin oppiminen 5
2.2.1 Elementti 6
2.2.2 Skeema 6
2.3 Kognitiivinen kuorma 7

3 KOGNITIIVISEN KUORMAN SÄÄTELYKEINOT 9
3.1 Tehtävätyypit 10
3.1.1 Pelkät esimerkit 10
3.1.2 Häiventyvät esimerkit 13
3.1.3 Ongelmanratkaisutehtävät 15
3.2 Tehtävien esitystapa 17
3.2.1 Tehtävän jakaminen osiin 17
3.2.2 Osatavoitteiden jakaminen ja merkitseminen 19
3.3 Informaation esitystapa 20
3.3.1 Tarpeettoman koodin piilotus 21
3.3.2 Tarkkaavaisuuden jakautuminen 21
3.3.3 Toiston poisto 22
3.3.4 Tiedon jakaminen vaiheittain 23

4 YHTEENVETO 24

KIRJALLISUUTTA 25

1 Johdanto

Peruskoulun opetusohjelma tulee muuttumaan syksyllä 2016, kun uusi, vuonna 2014 laadittu opetussuunnitelma astuu voimaan (Opetushallitus 2016). Erityisen mielenkiintoista tässä muutoksessa on ohjelmoinnin opetuksen mukaantulo; kaikki peruskoululaiset aloittavat kyseisen aineen opiskelun porrastetusti muutaman vuoden kuluessa. Tämä lisää ohjelmoinnin opiskelijoita, opettajia ja ihmisten yleistietoa ohjelmoinnista. Onkin tärkeää, että opiskelijat saavat mielekkään ja yhtenäisen opetuksen maanlaajuisesti. Jotta näin voi tapahtua, on ohjelmoinnin opettelemisen laaja ymmärtäminen paikallaan.

Uuden opetussuunnitelman innoittamana tutkielma keskittyy ohjelmoinnin oppimiseen. Yhtenä tärkeänä osana oppimista on kognitiivinen kuorma ja sen hallinta. (Sweller 1988). Oikeanlainen kuorma auttaa opiskelijaa asiasisällön omaksumisessa (Sweller 2010). Tässä tutkielmassa selvitetään, mitkä tekijät vaikuttavat kognitiiviseen kuormaan juuri ohjelmoinnin opettelemisen kannalta. Tutkimuksessa pyritään myös antamaan keinoja näiden tekijöiden muokkaamiseen, jotta kognitiivinen kuorma voidaan saada optimaaliselle tasolle. Metodin tarkoituksena on saada monipuolinen, tietoa yhdistelevä ja kartoittava vastaus tarkasti rajattuun tutkimuskysymykseen. Tällöin metodina on narratiivinen yleiskatsaus aiheeseen. (Salminen 2011.) Varsinainen tutkimuskysymys on: miten kognitiivinen kuorma voidaan huomioida ohjelmoinnin opettelun kannalta?

Tutkielma on toteutettu kuvailevana kirjallisuustutkimuksena (Salminen 2011). Hakupalveluina käytettiin Google Scholar ja ACM digital library -palveluita. Haku aloitettiin termeillä "Cognitive load programming", kielenä haettiin pääsääntöisesti englanninkielistä materiaalia eikä muita hakurajoituksia tehty. Lisäksi haettiin erillistä materiaalia hakusanoilla "worked examples" ja "worked examples not working", "faded examples" ja "what is programming". Useita henkilöitä kuten "John Sweller" haettiin erikseen. Monet lähteet löytyivät myös jo löydettyjen artikkeleiden lähteistä. Tutkimuksen esimerkkjä etsittiin Jyväskylän yliopiston ohjelmoinnin noviisikurssien materiaaleista.

Tutkielma lähtee liikkeelle tärkeimpien käsitteiden määrittelystä, joita ovat ohjelmointi, ohjelmoinnin oppiminen ja kognitiivinen kuorma. Tämän jälkeen, kappaleessa 3, käydään läpi kognitiiviseen kuormaan vaikuttavia tekijöitä tehtävätyyppien ja informaation esitystavan kannalta. Luku 4 sisältää yhteenvedon ja pohdintaa teorian käytännöllisyydestä.

2 Ohjelmoinnin oppiminen ja kognitiivinen kuorma

Tässä osiossa kartoitetaan olennaisimmat käsitteet, joita ovat: ohjelmointi, ohjelmoinnin oppiminen ja kognitiivinen kuorma. Ohjelmointia käsitellään eritoten kognitiotieteen ja psykologian näkökannalta ja samalla selvennetään myös ohjelmointikielten merkitystä ohjelmoinnissa. Oppimisen käsitettä on rajattu siten, että se nojautuu kognitiivisen kuorman teoriaan (Sweller 1988). Kognitiivinen kuorma on selitetty omana osionaan, myöskin kognitiivisen kuorman teorian pohjalta.

2.1 Ohjelmointi

Ohjelmoinnin ulkoisesti näkyvä toiminta on ohjelmakoodin tuottamista, muokkaamista ja testaamista. Emme kuitenkaan voi sanoa ohjelmoinniksi esimerkiksi sitä, että valmiiksi kirjoitettua koodia kirjoitetaan uuteen kohteeseen tarkasti annettujen ohjeiden mukaisesti. Ohjelmointiin vaaditaan siis myös ymmärrys siitä, mikä kirjoitetun koodin merkitys on. Tarvitaan koodin syntaksista että semanttista ymmärrystä. (Hoc ym. 1990, 10).

Ohjelmointikielet eroavat puhutuista kielistä siten, että niistä puuttuu monia elementtejä, joita puhutussa kielissä on. Tällaiset tekijät yleensä rikastavat puhuttua kieltä, tekevät siitä monitulkintaista ja antaa lisää vaihtoehtoja asian ilmaisemiseen. Ohjelmointikieli ei sisällä kuvainnollisia elementtejä, ei synonyymeja tai homonyymeja. Asioiden ei kuulu olla tulkinnanvaraisia. Puhutuissa kielissä esimerkiksi lauseen voi rakentaa monessa erilaisessa järjestyksessä. Ohjelmointikielissä lauseiden rakenne on hyvin rajattu. Alla oleva esimerkki osoittaa, kuinka kielissä lauseenjärjestystä voi muuttaa, mutta ohjelmointikielen esimerkeissä vain ensimmäinen on syntaktisesti laillista C#-kielessä, muut vaihtoehdot aiheuttavat virheen.

int-muuttujaan sijoitetaan arvo 3.	int a = 3;
Arvo 3 sijoitetaan int-muuttujaan.	3 = int a;
Sijoitetaan arvo 3 int-muuttujaan.	= 3 int a;

Ohjelmointikielen yhtenä tärkeänä ominaisuutena onkin yksinkertaisuus (Hoare 1989, ks. Kaijanaho 2010, 17-18). Myös Guy L. Steele (1999) puhui konferenssipuheessaan siitä, kuinka ohjelmointikielen tulisi olla pieni, mutta käyttäjän kasvatettavissa oleva. Pienen kielen etuna on se, että se ei kuormita tarpeettomilla ominaisuuksilla niitä, jotka kyseisiä ominaisuuksia eivät tarvitse. Toisaalta kasvatettavissa oleva kieli mahdollistaa näiden ominaisuuksien hankinnan niitä tarvitseville.

Suhteellisesti pieni ohjelmointikieli ei kuitenkaan tarkoita välttämättä helppoutta sen käytössä. Yksi haaste ohjelmoinnissa onkin puhutun kielen ja ajatellun algoritmin kääntäminen ohjelmointikielelle. Tämä on myös yksi ohjelmoinnin perustehtävistä: *"Programming ... is basically a process of translating from the language convenient to human beings to the language convenient to the computer"* (McCracken 1957, ks. Blackwell 2002). Jotta tämä käännöstyö on mahdollista, ohjelmoijan tulee tietää, milloin käytetty ohjelmointikieli hoitaa asian ja milloin ohjelmoijan itse on suunniteltava ja toteutettava ratkaisu. Tämä riippuu pitkälti kielen ominaisuuksista (Kaijanaho 2010, 9.) Esimerkiksi, onko kielessä sisäänrakennettu roskienkeruu vai täytyykö ohjelmoijan itse hoitaa muistintyhjennystä. Tämä ohjelmointikielen käytön haaste onkin yksi isoimmista hankaluuksista ohjelmoinnissa, johtuen siitä että ohjelmointikielen käyttö on abstraktia toimintaa (Blackwell 2002.) Ohjelmointikielen mukana tulee myös sille sopiva ohjelmointiympäristö. Ohjelmoijan tulee huomioida ohjelmointiympäristön ominaisuudet ja täten esimerkiksi se, miten ja minkälaisista virheistä se ilmoittaa.

Vaikka ohjelmoinnissa käytetään ohjelmointikieltä, ohjelmointiympäristöä ja muuta teknologiaa apuna, on se ihmisen mielensisäinen prosessi, kuten J-M Hoc ym. kertovat kirjassaan Psychology of programming: *"Programming is a human activity that is a great challenge, involving the design of machine behavior.. ".* Tähän mielen sisäiseen prosessiin viittaa myös seuraavat määritelmät ohjelmoinnista: ohjelmointi suunnitelmallista ohjeiden antamista (Lappalainen ym. 2013) ja *"ohjelmointi on ongelmien ratkaisemista"* (Kaijanaho 2010). Ongelmanratkaisu, suunnittelu ja ohjeiden antaminen ovat monimutkaisia kognitiivisia prosesseja. Näistä edellisistä lainauksista saadaan aikaiseksi seuraavanlainen määritelmä: Ohjelmointi on kognitiivinen

prosessi, jossa ratkaistaan jokin ongelma toimintaohjeiden avulla, käyttäen ohjelmointikieltä ja koneelle tyypillistä tapaa toimia. (Lappalainen ym. 2013; Kaijanaho 2010; McCracken 1957; Hoc ym. 1990).

Ohjelmoinnissa käytettäviä kognitiivisia toimintoja on tutkittu jonkin verran. Vuonna 2014 tehdyssä kokeessa tutkittiin aivojen verenkierron happipitoisuuksia fMRI-aivokuvantamislaitteella, kun koehenkilöt saivat tehtäväkseen lukea ja ymmärtää ohjelmointikoodia. (Siegmund ym. 2014). Kokeen tuloksena löydettiin viisi aktiivista aivoaluetta, jotka käsittelevät seuraavien toimintojen käsittelyä: työmuisti, keskittymiskyky ja kielelliset toiminnot. Koska koehenkilöiden varmistettiin kokeen jälkeen todella ymmärtävän kyseistä koodia, voidaan katsoa että ohjelmakoodin ymmärtämiseen vaaditaan näitä edellemainittuja kognitiivisia prosesseja. Tätä mieltä on myös Hoc ym. (1990) selventäessään, että ohjelmointiin vaaditaan monentasoisia kognitiivisia taitoja, kuten työmuistia, havaintokykyä, tarkkaavaisuutta ja työ- sekä pitkäkestoisen muistin käyttöä.

Ohjelmoijan tulee ottaa huomioon paljon erilaisia tekijöitä ohjelmoinnin aikana. Hänen tulee ymmärtää ratkaistava ongelma, suunnitella sille ratkaisu, toteuttaa se valitsemansa ohjelmakoodin mukaisella notaatiolla ja ymmärtää kielen ominaisuudet, ajatella ohjelman ylläpitoa tulevaisuudessa, ottaen huomioon kollegat ja loppukäyttäjät (Blackwell 2002). Tästä muodostuu lopullinen määritelmä: Ohjelmointi on työmuistia, keskittymiskykyä ja kielellisiä toimintoja vaativaa aktiivista toimintaa, jossa suunnitelmallisesti pyritään ratkaisemaan jokin ongelma käyttäen ohjelmointikieltä ja koneelle tyypillistä tapaa toimia (Kaijanaho 2010; Lappalainen ym 2013; McCracken 1957; Hoc ym. 1990; Siegmund ym. 2014).

2.2 Ohjelmoinnin oppiminen

Oppimisen päämääränä on asian ymmärtäminen. Jotta ymmärtäminen voidaan saavuttaa, tulee elementeistä muodostaa kokonainen skeema. (Sweller 1988.) Seuraavaksi käsitellään, mitä elementti ja skeema tarkoittavat Swellerin kognitiivisen kuorman teorian ja Piagetin kognitiivisen kehityksen teorian mukaisesti.

2.2.1 Elementti

Elementti on yksittäinen, vuorovaikutukseton asia (Sweller 1988). Ohjelmoinnissa elementeiksi voitaisiin laskea noviisien kohdalla esimerkiksi int-muuttujalle tyypilliset ominaisuudet C#-kielessä:

- int on C#-kielen avainsana
- int on alkeistietotyyppi
- int alustaa kokonaislukutyypin
- int vie 32 bittiä muistia
- int-tyyppiin voi tallentaa arvoja välillä -2,147,483,648 - 2,147,483,647

Näiden yksittäisten elementtien muistaminen ei vielä vaadi niiden keskinäisen vuorovaikutuksen ymmärtämistä. Kognitiivisen kuorman teorian (Sweller 1988) mukaan vuorovaikutuksen ymmärtäminen on kuitenkin oleellinen osa skeeman rakentamista. Teorian mukaisesti, kun elementtien vuorovaikutus on kokonaisuudessaan ymmärretty, on skeema rakennettu. Skeeman ymmärtäminen on siten verrattavissa asian oppimiseen. Tästä syystä ymmärtämättömyys johtuu siitä, että jotain skeemaan kuuluvaa elementtiä tai sen vuorovaikutusta muihin elementteihin ei ole opittu. (Sweller 1988.) Ohjelmoinnissa tällainen tilanne voi tulla vastaan, kun int-muuttujaan sijoitetaan liian suuri tai pieni arvo, eikä ymmärretä, miksi ohjelma ei toimi halutulla tavalla. Tällöin oppilaalle tulee tunne, ettei hän ymmärrä ohjelman toimimattomuutta, eikä siten osaa korjata virhettä.

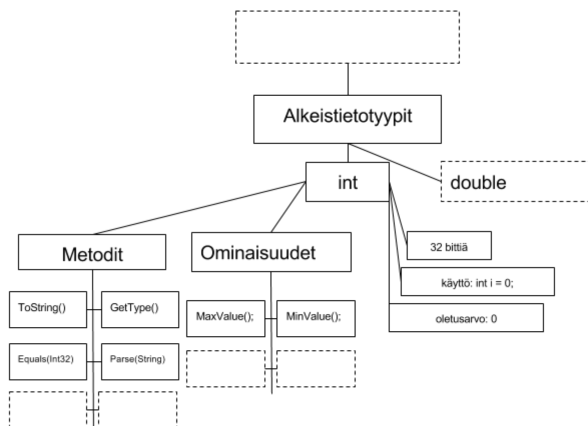
2.2.2 Skeema

Piagetin kognitiivisen kehityksen teorian mukaisesti skeemoja on jo vastasyntyneellä. Tällöin puhutaan kuitenkin vasta reflekseistä, jotka ovat sisäsyntyisiä, mutta ympäristön kanssa vuorovaikuttavia skeemoja. Nämä ensimmäiset skeemat ja niiden ympärille rakentuvat uudet skeemat ovat kaiken ajattelun pohja. (Piaget 1952.) Skeemat tallentuvat semanttiseen muistiin, jonne tallentuvat muutkin yleistiedot maailmasta. (Hoc ym. 1990.)

Skeema on elementeistä rakennettu kokonaisuus (Sweller 1988). Skeeman oppimi-

sen vaikeuteen vaikuttaa elementtien keskinäisen vuorovaikutuksen monipuolisuus. Mitä monipuolisempi ja vaikeasti ymmärrettävää vuorovaikutus on, sitä korkeampi kognitiivinen kuorma kyseisellä asialla on. Edellämainitun int-muuttujaan liittyvistä elementeistä tulisi muodostaa kokonaisuus, jolloin se vie vähemmän työmuistia ja vähentää kognitiivista kuormaa. (Sweller ym. 1998). Tästä syystä skeemat ovat hyödyllisiä ja niiden automatisoituminen on toivottavaa. Kun oppilas on saanut rakennettua toimivan skeeman int-muuttujasta, hän voi käyttää sitä lähes automaattisesti, ikään kuin ajattelematta ja keskittyä uuden skeeman rakentamiseen. Hän ei enää tässä vaiheessa pidä työmuistissaan tallella kaikkia int-muuttujalle tyypillisiä ominaisuuksia, joten työmuistia vapautuu muuhun käyttöön.

Esimerkki yksinkertaisesta skeemasta. Nyt useista int-muuttujaan liittyvistä elementeistä on rakennettu skeema.



2.3 Kognitiivinen kuorma

Kognitiivista kuormaa on kognitiivisen kuorman teorian mukaan kolmenlaista: Luontainen (engl. intrinsic), ulkoinen (engl. extraneous) ja hyödyllinen (engl. germane) (Sweller 1988). Luontainen kuorma on asiasisällön oma kuorma. Ohjelmoinnissa tällainen asia voisi olla esimerkiksi ehtolauseiden muodostaminen tai funktioiden perusrakenne. Luontaiselle kuormalle on ominaista se, että sitä ei juurikaan voi vähentää (Sweller 1988). Tämä johtuu siitä, että luontainen kuorma koostuu elemen-

teistä ja niiden vuorovaikutuksesta. Vähentäminen tarkoittaisi jonkin elementin tai sen vuorovaikutuksen poistamista, mikä muuttaisi itse opittavan asian sisältöä.

Ulkoinen kuorma taas koostuu asiasisällön ulkopuolisista tekijöistä. Ulkoiset tekijät on helppo tunnistaa siitä, että niitä voi vähentää tai poistaa, ilman että opittava asia kärsii. (Sweller 1988.) Tällaisia asioita on esimerkiksi ohjemateriaali tai oppimisympäristön hallintaan kuuluva oppiminen. Luontainen ja ulkoinen kuorma ovat kumulatiivisia ja vievät siten yhteisesti työmuistin resursseja (Sweller 1988). Tämä on merkittävää siksi, että ihmisten työmuisti on hyvin rajallinen verrattuna säilömuistiin. Työmuistissa voidaan käsitellä noin seitsemää (+-2) asiaa kerrallaan. (Miller 1956). Mikäli tarkastelun kohteena olevista asioista suurin osa koskee jatkuvasti esimerkiksi oppimisympäristöön kuuluvien asioiden tarkastelua, ei varsinaista oppimista juurikaan tapahdu. Todellisen ja ulkoisen kuorman kumulatiivisuuden takia toisen vähentäminen vähentää kognitiivista kokonaiskuormaa (Sweller 1988). Koska todellista kuormaa ei voi oikeastaan vähentää (Sweller 2010), useat keinot keskittyvät ulkoisen kuorman vähentämiseen.

Hyödyllinen kognitiivinen kuorma on se kuorman alue, joka keskittyy todellisen kuorman käsittelyyn. (Sweller 1998). Mikäli opittava skeema on funktioiden perusrakenne ja oppilas keskittyy funktioiden tekemisen oppimiseen, voidaan ajatella että oppilaan hyödyllinen kuorma on korkea. Mikäli opittavana asiana on edelleen funktiot, mutta henkilö keskittyy oppimateriaalin suomentamiseen, koska se on englanniksi, eikä oppilas ymmärrä englantia, on hyödyllinen kuorma lähes olematon. Yleisesti ottaen voidaan ajatella, että oppimisen kannalta on mielekästä, jos hyödyllisen kuorman alue on mahdollisimman suuri ja ulkoisen kuorman alue mahdollisimman pieni. Tällöin ihmisen työmuisti keskittyy olennaiseen asiaan ja ihminen kykenee rakentamaan tarvittavaa skeemaa. Pitää kuitenkin huomioida, että pelkkä ulkoisen kuorman vähäisyys ei ole päämäärä. Ulkoista kuormaa voidaan helposti minimoida poistamalla kaikki opetusmateriaali tai yrittämällä kiteyttää materiaali mahdollisimman kompaktiin muotoon. Tämä ei kuitenkaan myöskään opeta mitään, mikä ei tietenkään ole toivottavaa. Tehtävänä on saada aikaan mahdollisimman vähäinen kuorma siten, että luontainen kuorma ei kärsi.

3 Kognitiivisen kuorman säätelykeinot

Ohjelmoinnilla on korkea kognitiivinen luontainen kuorma (Paas, & Van Merriënboer 1994, 351-371) ja ohjelmoinnin alkeiskursseilla on suuri keskeytysprosentti (Kinnunen & Malmi 2006; Lakanen & Lappalainen 2014). Keskeytyksien syitä on tutkittu Kinnusen ja Malmin vuonna 2006 teettämässä tutkimuksessa, missä selvisi että aika, motivaation puute ja ohjelmoinnin vaikeus olivat suurimpia keskeyttämistä selittäviä tekijöitä ja että vaikeus oli selittävä tekijä myös ajan ja motivaation kanssa. (Kinnunen & Malmi 2006.) Ohjelmointikurssien vaikeuteen vaikuttaa osaltaan ohjelmoinnille ominainen suuri kognitiivinen kuorma. Luontaisen kuorman jo ollessa suuri, olisi tärkeää että ulkoinen kuorma saataisiin mahdollisimman pieneksi. Tämä ei tee ohjelmoinnista helppoa, mutta ulkoisen kuorman vähentäminen lisää hyödyllisen kuorman osuutta, joka on oleellinen asia skeeman rakentamisessa ja asian ymmärtämisessä (Sweller 1998.), sillä nimenomaan ymmärtämättömyys tekee ohjelmoinnista vaikeaa (Kinnunen & Malmi 2006). Kinnusen ja Malmin tutkimuksessa todettiin, että oppilaat mainitsivat ymmärtämättömyyden jopa yhdeksi keskeyttämisen syyksi. Esimerkkinä yhden oppilaan lainaus paljasti, että jatkuvat ongelmat virheiden tulkitsemisessa tekivät ohjelmoinnista vaikeaa.

Kognitiivista kuormaa voi säädellä keskittymällä niihin kahteen osa-alueeseen, joita opetuksessa yleensä on: opetusmateriaaliin ja tehtäviin. Seuraavaksi käydään läpi kognitiivisen kuorman teoriaan pohjautuvia keinoja, joiden avulla kognitiivista kuormaa voi hallita. Keinot pohjautuvat aikaisempaan tutkimustietoon ja niitä on havainnollistettu mielekkäin esimerkein, jotka on saatu Jyväskylän yliopiston ohjelmointikurssien oppimateriaaleista.

Tehtävätyypit, jotka tässä käydään läpi ovat: malliesimerkit, häiventyvät esimerkit ja ongelmanratkaisutehtävät. Tehtävätyyppien esitystapoja käsitellään lisäksi erikseen. Tehtävätyyppien lisäksi käydään läpi opetusmateriaalin esitystapaan liittyviä hallintakeinot, joita on (1) kuinka tarkkaavaisuus jakautuu materiaalia opetellessa, (2) onko materiaalissa turhaa toistoa (3) tiedon jakaminen vaiheittain.

3.1 Tehtävätyypit

Malliesimerkit (engl. worked examples) sisältävät toimintaohjeet ongelman ratkaisemiseen. Skudder & Luxton (2014) esittävät neljä erilaista variaatiota malliesimerkeistä joko yhdistettynä tehtäviin tai ilman niitä. Nämä variaatiot ovat: Pelkät esimerkit (engl. examples only), häiventyvät esimerkit (engl. faded examples), mallitehtäväparit (engl. example-problem pairs) ja malli-tehtäväryhmät (engl. example-problem blocks). Seuraavaksi käydään tarkemmin läpi mallesimerkkien eri variaatiot ja niiden vaikutukset ohjelmoinnin opetuksessa, erityisesti keskittyen kognitiiviseen kuormaan.

3.1.1 Pelkät esimerkit

Pelkissä esimerkeissä esitellään opeteltava kokonaisuus, esimerkiksi ratkaisumalli ongelmanratkaisutehtävään. Ohjelmoinnissa tällainen malli voisi olla ohjeistus funktion rakentamisesta. (Kuvio 1). Ohjelmoinnin malliesimerkki sisältää tyypillisesti mallikoodia sekä metakielellä kirjoitettua ohjeistusta, sisältäen ongelman ja askeleittain eteneviä vaiheita lopullista ratkaisua kohti. Mikäli ohjeessa on kysymyksiä, niihin myös vastataan samassa esimerkissä. Pelkissä esimerkeissä ei ole erillistä tehtävää mallin opettelun lisäksi.

Kysymykseen, kannattaako pelkkiä esimerkkejä käyttää ohjelmoinnin opetuksessa, ei ole yksiselitteistä vastausta. Sweller & Cooper (1985) esittävät hypoteesin siitä, että pelkät esimerkit voisivat toimia noviisien kohdalla paremmin skeeman rakentamisessa verrattuna ongelmanratkaisutehtäviin. Tämä johtuu heidän mukaansa siitä, että ongelmanratkaisutehtävät eivät välttämättä kohdista tarkkaavaisuutta skeeman rakentamiseen aivan yhtä hyvin kuin malliesimerkit. Ongelmanratkaisutehtävissä ongelman vaiheittainen ratkaiseminen vie niin ison osan henkisestä ponnistelusta (engl. mental effort) että skeeman rakentaminen noviisien kohdalla on hitaampaa kuin malliesimerkkien avulla. Pelkkien esimerkkien kanssa vastaavaa ongelmaa ei ole. (Sweller & Cooper 1985.)

Eksperttien kanssa pelkät esimerkit eivät kuitenkaan välttämättä nopeuta skeeman

How to actually go about writing simple functions?
1. Types, define what is the input for the function and what is the output.
Write this down as a type declaration.
2. Example applications, write some examples of how your function could be used and what the result should be.
3. The function definition, or how the function actually and exactly operates. If you can't do it yet, then Look at your examples
4. The tests, You're not done until you've made sure that your function actually works.
Worked example on defining a simple function
Suppose we need to calculate the number of words in a piece of text
1. Types
Firstly, the type of the function is probably String -> Int, since it takes a string of letters and returns a number, so
numberOfWords :: String -> Int
numberOfWords str = undefined
2. Example applications
Secondly, we will write some examples:
numberOfWords :: String -> Int
numberOfWords str = undefined
– Examples:
– numberOfWords "I am Groot-= 3
Definition
– | Number of words counts the number of *sequences of
– characters* not containing whitespace but which are
– separated by arbitrary amounts of whitespace.
numberOfWords :: String -> Int
numberOfWords str = length (words str)
Tests
Next we test our definition with an Haskell interpreter

Kuvio 1: Malliesimerkki Jyväskylän yliopiston Funktio-ohjelmoinnin noviisikurssilta (Tirronen 2015). Lyhennetty versio.

oppimista. Eksperttien tapauksessa malliesimerkit voivat olla turhaa toistoa ja niillä voi olla jopa negatiivinen vaikutus. (Kalyuga ym 2003.) Kalyuga ym käyttävät tästä ominaisuudesta termiä eksperttien päinvastaisuuden efektiksi (engl. the expertise reversal effect). Toistosta puhutaan lisää osiossa 3.3.2 toiston poisto.

Malliesimerkkien avulla ei kuitenkaan voida varmistaa, että oppilas opettelee välttämättä mitään (Sweller & Cooper 1985), koska varsinaista tehtävää ei ole. Pelkän esimerkin opetteleminen on täysin oppilaan vastuulla ja suurelta osalta kiinni hänen motivaatiostaan. Silloinkin, kun oppilas keskittyy täysin ja hyvällä motivaatiolla esimerkin opettelemiseen, ei voida varmistaa, että tarkkaavaisuus kohdistuu skeeman kannalta olennaisiin asioihin.

Pelkät esimerkit voisivat kuitenkin auttaa nimenomaan noviiseja toimimaan eksperttien tavoin. Ekspertit osaavat jakaa ongelmanratkaisutehtävän osiin ja ratkoa osat yksi kerrallaan, edeten takaperin kohti alkua. Noviisit sen sijaan ratkovat ongelmia mahdollisesti yksi koodirivi kerrallaan, aloittaen alusta ja edeten kohti ratkaisua. (Anderson 1985, ks. Hoc 1990, 72.) Noviisit voivat opetella pelkän esimerkin avulla, kuinka ongelman voi jakaa osiin ja käyttää vastaavaa tapaa tulevissa ongelmanratkaisutehtävissä.

Ohjelmointia ei voi kuitenkaan kokonaan opetella pelkkien esimerkkien avulla. Tämä johtuu siitä, että ohjelmoinnin oleellinen osa on ongelmanratkaisutaito (Kaijanaho 2010), eikä ongelmanratkaisua voi opettaa tai opetella vain esimerkkien avulla. Ensinnäkin, malliesimerkkejä ei voi esittää joka ikiselle ongelmalle, sillä niitä tulisi liian suuri määrä opeteltavaksi (Sweller & Cooper, 1985) ja koska emme tunne vielä kaikkia ongelmia ratkaisumalleineen. Pelkkien esimerkkien ideana on opetella skeema, jota voi soveltaa tarvittaviin ongelmanratkaisutehtäviin. Sweller & Cooper esittävätkin, että tuntemattomia onglemanratkaisutehtäviä on tarjottava opetustavasta riippumatta. Pelkät esimerkit eivät siten voi ohjelmoinnin oppimisessa korvata täysin ongelmanratkaisutehtäviä, vaikka ne veisivätkin vähemmän kognitiivista kuormaa noviisien kohdalla.

On kuitenkin mahdollista käyttää malliesimerkkejä yhdistettynä ongelmanratkai-

sutehtäviin. Skudder & Luxton (2014) esittelivät tästä kaksi erilaista tapaa, jossa ratkaisumalli ja ongelmanratkaisutehtävä esiintyvät pareittain tai ryhmittäin. Mallitehtäväparissa esitetään yksi malliesimerkki, jota seuraa ongelmanratkaisutehtävä. Kun tehtävä on ratkaistu, annetaan uusi esimerkki ja uusi tehtävä. Malli-esimerkkiryhmässä esitetään useita malliesimerkkejä, joita seuraa isompi määrä ongelmanratkaisutehtäviä. Näiden malliesimerkkien ja ongelmanratkaisutehtävien yhdistelmien voi olettaa soveltuvan ohjelmoinnin opiskeluun pelkkiä esimerkkejä paremmin, sillä niissä yhdistyy pelkkien esimerkkien ohjeistus ja vähäisempi kognitiivinen kuorma, sekä ongelmanratkaisutehtävien mahdollistama itse tekeminen ja sen motivoiva vaikutus. Pelkät esimerkit yksinään eivät kuitenkaan riitä opettamaan ohjelmointia ja pelkät ongelmanratkaisutehtävät voivat olla kognitiiviselta kuormaltaan liian haastavia. Yhtenä vaihtoehtona on myös seuraavaksi käsiteltävä tehtävämalli, häiventyvät esimerkit.

3.1.2 Häiventyvät esimerkit

Häiventyvissä esimerkeissä (engl. faded examples) on kyse siitä, että malliesimerkeistä jätetään oppilaan ratkaistavaksi jokin osa-alue. Tätä tapaa kutsutaan myös opastuksen hälventymiseksi (engl. guidance fading effect) (Sweller 2010). Ohjelmoinnin osalta tällainen tehtävä voisi olla funktioiden esittelyrivin hahmottaminen. Ensin oppilaalle tarjotaan malliesimerkki funktiosta. Tämän jälkeen osa funktiosta jätetään tyhjäksi ja oppilaan tulee selventää, mikä osa funktiosta puuttuu. Kuvio 2 esittää häiventyvien esimerkkien yhtä mahdollista toteutustapaa käyttäen C#-kieltä. Siinä annetaan dokumentaatio, funktion nimi, parametrien nimet ja sulut valmiina. Oppilaan tehtäväksi jää täydentää saantimääre, staattisuus, palautustyyppi sekä parametrien tyypit.

Häiventyviä esimerkkejä on ehdotettu pelkkien malliesimerkkien ja ongelmanratkaisutehtävien siirtymävaiheeseen. Täysin valmiiden malliesimerkkien jälkeen esiteltäisiin aina vain keskeneräisempiä esimerkkejä, kunnes lopulta siirryttäisiin puhtaisiin ongelmanratkaisutehtäviin (Renkl ym. 2002). Renkl ym. osoittivat kokeellisesti, kuinka häiventyvät esimerkit paransivat oppimistuloksia läheisessä siirtovai-

```
/// <summary>
/// Lasketaan kolmion pinta-ala
/// </summary>
/// <param name="luku1»kanta</param>
/// <param name="luku2»korkeus</param>
/// <returns>pinta-ala</returns>
XXX YYY ZZZ KolmionAla (??? luku1, IIII luku2)
{
}
```

Kuvio 2: Esimerkki Jyväskylän yliopiston Ohjelmointi 1-kurssilta (Lappalainen ym. 2016): Tehtävässä oppilaan tulee täydentää kohtien XXX YYY ZZZ ??? ja IIII tilalle oikeat vaihtoehdot.

kutuksessa verrattuna malliesimerkki-tehtäväryhmään. (Renkl ym. 2002). Tulokset eivät kuitenkaan ulottuneet kaukaiseen siirtovaikutukseen, minkä syynä ehdotettiin ratkaisun yleistymättömyyttä oppilaan mielessä. Tähän voisi heidän mukaansa auttaa, jos oppilas samanaikaisesti opiskelisi sekä häiventyvää esimerkkiä, että selittäisi itselleen sen rakenteita. Tälle efektille on myös nimi *itse selityksen efekti* (engl. self-explanation effect) (Skudder & Luxton, 2014; Sweller 2010). Tämä voisi auttaa yleistämään ohjetta laajemmalla tasolla ja parantaa myös kaukaista siirtovaikutusta (Renkl ym. 2002.) Käytännössä tämä tarkoittaisi sitä, että oppilas kykenisi soveltamaan skeemaa myös tehtäviin, jotka ovat hyvin erilaisia verrattuna alkuperäiseen skeemaan.

Häiventyvien esimerkkien etu verrattuna pelkkiin esimerkkeihin piilee sen kyvyssä kohdistaa oppilaan tarkkaavaisuus vain tiettyyn kohtaan koko esimerkin sijasta. Pelkissä esimerkeissä tulee opetella kokonaisuus, mutta häiventyvissä esimerkeissä voidaan osoittaa ja testata nimenomaan tiettyä osaa esimerkistä.Paitsi että häiventyvissä esimerkeissä on vähemmän ulkoista kognitiivista kuormaa kuin ongelmanratkaisutehtävissä, se myös kohdentaa oppilaan tarkkaavaisuutta oikeaan kohtaan ja estää näin epätuottavia oppimistapahtumia (Renkl ym. 2004). Tällöin varmistetaan että oppilas todella keskittyy juuri skeeman kannalta olennaisiin elementteihin

ja niiden vuorovaikutukseen. Koska häiventyvissä tehtävissä täytyy myös täydentää oikea vastaus, voidaan pelkkiä esimerkkejä paremmin varmistaa, että oppilas todella opettelee asiaa.Häiventyvien esimerkkien etuna taas verrattuna ongelmanratkaisutehtäviin voi nähdä sen kuormaa vähentävän vaikutuksen noviisien kohdalla. Oppilaan ei tarvitse vielä keskittyä ongelman kaikkiin osa-alueisiin, minkä voi olettaa vähentävän kognitiivista kokonaiskuormaa.

Mikä osa sitten pitäisi hälventää eli jättää tyhjäksi? Renkl ym. (2002) osoittivat tekemässään kokeessa, että virheiden kannalta ei ole merkitsevää, aloitetaanko häiventäminen alusta vaiko lopusta. (engl. backward fading, forward fading). Kuitenkin oppimiseen käytetty aika oli pienempi lopusta häivennetyissä tehtävissä (Renkl ym. 2002). Vaikuttaa kuitenkin siltä, että oppilas muistaa häivennetyn osa-alueen muita paremmin (Renkl ym. 2004; Fleischman & Jones 2002). Ohjelmoinnissa voisi pohtia, tulisiko ensin häiventää sellaiset alueet, joita muut elementit eivät lainkaan tarvitse, vaiko nimenomaan ne, joita kaikki muut osa-alueet tarvitsevat. Esimerkiksi yksinkertaisissa funktioissa voisi häiventää ensin toteutuksen, antaen valmiiksi esittelyrivin, dokumentaation, testit, sekä palautustyypin. Tällaisessä muodossaan ohjelma on ajettavissa ja virheetön. Toisaalta taas voitaisiin keskittyä nimenomaan niihin pakollisiin osa-alueisiin ja käskeä oppilasta täydentämään funktion esittelyrivi tai palautustyyppi. Lopputuloksena päästään siihen, että häivennetään se alue funktiosta, johon pitää keskittyä ja joka pitää opetella.

3.1.3 Ongelmanratkaisutehtävät

Ongelmanratkaisutehtävissä on kyse tehtävistä, jossa oppilaan tehtävänä on ratkaista jokin ongelma.

Esimerkki Jyväskylän yliopiston Ohjelmointi 1-kurssilta (2016):

```
Kirjoita funktio nimeltä ‘MailitKilometreiksi‘, joka palauttaa parametrina
viedyn mailit kilometreinä.
```

Ongelmanratkaisutehtävät ovat kognitiivisesti kuormittavia noviiseille (Sweller 1988; Sweller 2010). Kuormittavuus ei itsessään ole huono asia, mikäli se on hyödyllistä

kuormaa ja tukee näin oppimista. Sweller kuitenkin osoittaa, että näin ei välttämättä ole. Ongelmanratkaisutehtävien avulla oppiminen tapahtuu suhteellisen hitaasti, johtuen kenties siitä, että ongelmanratkaisutehtävissä tarkkaavaisuus ei kohdistu skeeman kannalta olennaisiin asioihin (Sweller & Cooper 1985.) Skudder & Luxton (2014) sanovat, että ohjaamattomissa ongelmanratkaisutehtävissä oppilaan toiminta kohdistuu ratkaisun etsimiseen skeeman rakentamisen sijasta. Sweller & Cooper (1985) esittävätkin hypoteesin, jonka mukaan ongelmanratkaisu vaatii oppijalta monia skeemoja ja niiden toteuttamista yhtäaikaisesti. Tämän voi olettaa olevan kognitiivisesti noviiseille hyvin kuormittavaa, koska he rakentavat näitä skeemoja samaan aikaan kun käyttävät niitä. Suuri kognitiivinen kuorma vaikuttaa haitallisesti oppimiseen. (Sweller & Cooper 1985.)

Vaikka ongelmanratkaisutehtävät ovat noviiseille mahdollisesti kuormittavia, voi niiden käyttö silti olla ohjelmoinnin oppimisessa järkevää. Syynä on muun muassa se, että ohjelmointi on osaltaan ongelmanratkaisua (Kaijanaho 2013), jolloin pelkkä tiedon omaksuminen ei riitä. Oleellista ohjelmoinnissa on se, että miten sitä tietoa osataan käyttää (Hoc, 1990, 231-132). Kohdassa 3.1.1 pelkät esimerkit jo todettiin, kuinka pelkät esimerkit eivät riitä kokonaisvaltaiseen ohjelmoinnin opettelemiseen. Häilyvien esimerkkien ongelman taas on niiden huono kaukainen siirtovaikutus. Ongelmanratkaisutehtävät ovatkin näille kahdelle oiva jatkumo, kunhan jotain oleellisia skeemoja on saatu rakennettua pelkkien esimerkkien ja häilyvien esimerkkien avulla. Skudder ym. (2014) esittävätkin, että juuri tämä järjestys on oppimisen kannalta suotuisa. Mikäli oppilaan annetaan ensin opetella skeema, sitten esitellään ongelma, hän saavuttaa parempia tuloksia kuin jos ensin annettaisiin tehtävä ja vasta sen jälkeen malliesimerkki tarvittavasta skeemasta. Tämä vaikuttaa järkevältä kognitiivisen kuorman kannalta, sillä jos oppilas saa samanaikaisesti tehdä tehtävää ja opiskella malliesimerkkiä, osa henkisestä ponnistelusta menee ongelmaavaruudessa olemiseen ja vastauksen etsimiseen ja osa taas yrittää samanaikaisesti opetella skeemaa. Näiden molempien tuottama kognitiiviivinen kuorma on kumulatiivista ja voi vaikuttaa oppimiseen.

Eksperttien kohdalla voi olla mielekästä käyttää ongelmanratkaisutehtäviä, ainakin

verrattuna malliesimerkkeihin, sillä ekspertit eivät hyödy malliesimerkeistä samalla tavalla kuin noviisit. Tämä johtuu eksperttien päinvastaisuuden efektistä (Kalyuga ym. 2003.) Kalyuga ym. esittävätkin, että malliesimerkeistä ainakin osa on todennäköisesti heille vain turhaa toistoa. Toiston takia heille voi olla järkevintä käyttää ongelmanratkaisutehtäviä pelkkien- tai häiventyvien esimerkkien sijasta. Lisäksi ongelmanratkaisutehtävät saattavat varmistaa edes jonkinasteista skeeman omaksumista, sillä vaikka niiden avulla skeema opittaisiin hitaammin, jää siitä väistämättä aina jotain muistiin.

3.2 Tehtävien esitystapa

Kognitiiviseen kuormaan voi vaikuttaa erilaisten tehtävätyyppien lisäksi myös tehtävien esitystapaan liittyvillä tekijöillä. Seuraavaksi käydään läpi kaksi tälläistä tekijää.

3.2.1 Tehtävän jakaminen osiin

Kohdassa 2.3 selitettiin, kuinka luontaista kuormaa ei juurikaan voi vähentää. Tämä on totta, kokonaiskuorma ei voida vähentää, mutta voimme jakaa kokonaiskuorman osiin ja pienentää siten kerralla käsiteltävää kuormaa. Tämä on erityisen hyödyllistä ongelmanratkaisutehtävissä, joissa noviisien on opeteltava yhtä aikaa monta erilaista skeemaa ja lisäksi keskittyä oikeaan ratkaisuun. Noviisit ja ekspertit ratkaisevat ongelmanratkaisutehtäviä erilailla (Hoc 1990). Jakamalla tehtävä osiin saatetaan auttaa noviiseja siinä, että näytetään miten tehtävää kannattaa alkaa ratkoa ja minkälaisissa vaiheissa edetä kohti ratkaisua. Kuvio 3 näyttää, kuinka tehtävän jakamista osiin toteutettiin Ohjelmoinnin noviisikurssilla Jyväskylän yliopistossa. Kuvio sisältää myös kuvitteellisen tehtävän, ilman osiinjakamista.

Tällainen osittaminen saattaa oikeastaan lisätä, ainakin näennäisesti, ulkoista kuormaa. Ohjeita ja tehtäviä tulee oppilaalle moninkertainen määrä. Jakaminen kuitenkin mahdollistaa pienemmät ongelma-avaruudet ja helpottaa noviiseja siten pääsemään oikeaan ratkaisuun. Liian laaja ongelma-avaruus, ilman malliesimerkkiä

voi osoittautua noviiseille liian hankalaksi. Tehtävien osittaminen toimii näin myös mallina siitä, kuinka tällaistä laajaa tehtävää voi rakentaa osista.

Tehtävä 1

M: 15. Taulukot. Tee uusi Jypeli-peli (Fysiikkapeli), ja kopioi Begin-aliohjelmaan alla oleva koodi.

```
PhysicsObject[] pallot = new PhysicsObject[] {
  LuoPallo(10, 0, 0, Color.White),
  LuoPallo(10, 20, 20, Color.White),
  LuoPallo(10, 40, 40, Color.White),
  LuoPallo(10, 60, 60, Color.White),
  LuoPallo(10, 80, 80, Color.White)
};
```

Tehtävä 2

(Tässä tehtävässä jatketaan tehtävää 1.)

a) Tee `Begin`-aliohjelmaan, ekan tehtävän koodin perään, silmukka, jolla laitat `pallot`-taulukossa olevat pallot satunnaisesti ympäri kenttää. Huom! Et saa olettaa, että taulukossa on viisi palloa, eli koodisi pitää toimia muunkin kokoiselle taulukolle.

Tehtävä 3

(Tässä tehtävässä jatketaan tehtävää 2.)

Tee funktio `TeeSatunnaisetPallot`, jolle annat parametrina luvun, kuinka monta palloa arvotaan. Tähän perustuen funktio luo taulukollisen palloja satunnaisesti ympäri pelikenttää, ja palauttaa tuon taulukon, eli paluuarvon tyyppi on `PhysicsObject[]`. Käytä funktioita, joita teit tehtävissä 1-2.

Tehtävä 4

(Tässä tehtävässä jatketaan tehtäviä 1-3. Voit tehdä tämän samaan kooditiedostoon edellisten tehtävien kanssa.)

M: 15. Taulukot, 16. Silmukat.

Tee funktio

```
public PhysicsObject LahinPallo(PhysicsObject[] pallot, Vector piste)
{
 // täydennä ...
}
```

joka etsii ja palauttaa annettua pistettä lähimmän taulukon alkion `PhysicsObject`-taulukosta.

Tehtävä 5

Tässä tehtävässä voit hyödyntää tekemiäsi tehtäviä 1-4.

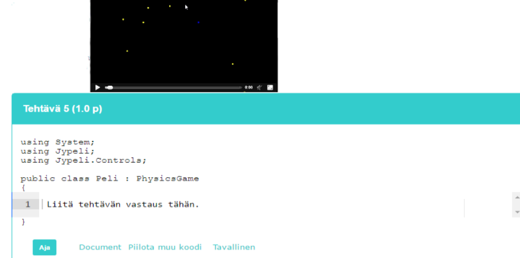
1. Lisää peliin pieni sininen pallo. Tee aliohjelma, joka siirtää pienen sinisen pallon siihen kohtaan, mihin klikkasit hiirellä. Hiiren paikan ruudulla saat tätä ohjetta käyttäen.
2. Tehtäviä 1-4 (ja 5 a) hyväksi käyttäen tee Jypeli-peli, missä arvotaan ruudulle 15 kappaletta palloja. Kun hiirellä klikkaa ruudulle, ilmestyy siihen kohtaan sininen pallo. Samalla lähimmän pallon kohdalle ilmestyy punainen pallo. Sama toistuu kun hiirellä klikataan uudestaan jonnekin muualle, niin sinisen kuin punaisen pallon paikka päivittyy.

Katso video

Tehtävä 1

1. Lisää peliin pieni sininen pallo. Tee aliohjelma, joka siirtää pienen sinisen pallon siihen kohtaan, mihin klikkasit hiirellä. Hiiren paikan ruudulla saat tätä ohjetta käyttäen.
2. Tee Jypeli-peli, missä arvotaan ruudulle 15 kappaletta palloja. Kun hiirellä klikkaa ruudulle, ilmestyy siihen kohtaan sininen pallo. Samalla lähimmän pallon kohdalle ilmestyy punainen pallo. Sama toistuu kun hiirellä klikataan uudestaan jonnekin muualle, niin sinisen kuin punaisen pallon paikka päivittyy.

Katso video



Kuvio 3: Tehtavan jakaminen osiin

Jyväskylän yliopiston Ohjelmointi 1-kurssilta (Lappalainen ym. 2016). Oikeanpuoleinen on kuvitteellinen kuva siitä, minkälainen tehtävä olisi, jos se olisi annettu kerralla. Vasemmalla on oikea tehtävänanto, mutta välivaiheita ja ohjeita on karsittu yksinkertaistuksen vuoksi. Lisäksi jokainen tehtävävaihe sisälti oman vastauslaatikkonsa, tässä nekin on poistettu kaaviosta.

Alunperinkin on hyvä pohtia tehtävään liittyviä vaiheita ja ovatko ne oppilaan taitotasolla. Mikäli ei, voi tehtävän jakaa osiin. Toisaalta tämä myös tarkoittaa sitä, että tehtävien ei välttämättä tarvitse olla itsenäisiä kokonaisuuksia, vaan ne voivat tukeutua toisiinsa. Lisäksi tämä voi tarkoittaa sitä, että ongelmanratkaisutehtäviä voi myös käyttää opetusmenetelmänä, jos sen avulla osoitetaan isomman ongelman ratkaisuvaiheita.

3.2.2 Osatavoitteiden jakaminen ja merkitseminen

Kuvio 4 osoittaa kaksi erilaista tapaa tehdä malliesimerkki funktion rakentamisesta Haskel-ohjelmointikielellä. Kuvion vasemmalla puolella on sama kuvio kuin kuvio 1:ssä. Oikealla taas on kuvitteellinen esitystapa, jossa osatavotteita ei ole esitelty tai nimetty.

How to actually go about writing simple functions?
1. Types, define what is the input for the function and what is the output.
Write this down as a type declaration.
2. Example applications, write some examples of how your function could be used and what the result should be.
3. The function definition, or how the function actually and exactly operates. If you can't do it yet, then Look at your examples
4. The tests, You're not done until you've made sure that your function actually works.
Worked example on defining a simple function
Suppose we need to calculate the number of words in a piece of text
1. Types
Firstly, the type of the function is probably String -> Int, since it takes a string of letters and returns a number, so
numberOfWords :: String -> Int
numberOfWords str = undefined
2. Example applications
Secondly, we will write some examples:
numberOfWords :: String -> Int
numberOfWords str = undefined
– Examples:
– numberOfWords "I am Groot-= 3
Definition
– | Number of words counts the number of *sequences of
– characters* not containing whitespace but which are
– separated by arbitrary amounts of whitespace.
numberOfWords :: String -> Int
numberOfWords str = length (words str)
Tests
Next we test our definition with an Haskell interpreter

– | Number of words counts the number of *sequences of
– characters* not containing whitespace but which are
– separated by arbitrary amounts of whitespace.
1. numberOfWords :: String -> Int
2. numberOfWords str = length (words str)
3. – Examples:
4. – numberOfWords "I am Groot-= 3
How to actually go about writing simple functions?
First, we should define what is the input for the function and what is the output. You need to write down the type declaration (see line 1). Then, write some examples of how your function could be used and what the result ahould be (see line 3 and 4). Write down how the function actually and exatly operators. If you can't do it yet, then look at your examples. You're not done until you've made sure that your function actually works.

Kuvio 4: Malliesimerkin osittaminen ja nimeäminen. Jyväskylän yliopiston Funktio-ohjelmoinnin kurssilta (Tirronen 2016). Oikeanpuoleinen on kuvitteellinen kuva siitä, minkälainen tehtävä olisi, jos se olisi annettu kerralla. Vasemmalla on oikea tehtävänanto, mutta välivaiheita ja ohjeita on karsittu yksinkertaistuksen vuoksi.

Malliesimerkit voidaan esittää usealla erilaisella tavalla. Kuviossa 4 esitetty vasemmanpuoleinen tapa sisältää pelkän koodin sijasta useita muitakin elementtejä ja se on jaettu osiin, joista jokaisella osalla on oma nimensä. Lisäksi varsinaiset ratkai-

suvaiheet on käyty läpi kahteen kertaan. Ensimmäisellä kerralla ratkaisusta käytiin läpi vain sen ulkoinen rakenne, ottamatta kantaa kielen syntaksiin. Tässä vaiheessa jokaiselle vaiheelle annettiin nimi ja selitettiin, mitä vaiheessa oli tarkoitus saada aikaiseksi. Toisella läpikäyntikerralla jokaista nimettyä osiota vastaan esitettiin mallikoodia käyttäen kurssin Haskel-ohjelmointikieltä. Tälläinen osittaminen ja nimeäminen auttaa kognitiivisen kuorman vähentämisessä ja ongelmanratkaisemisessa (Margulieux ym. 2012). Vaikka vasemmalla oleva esimerkki on paljon pidempi, se myös kohdentaa tarkkaavaisuuden yleisten rakenteiden suuntaan nimeämällä, osittamalla ja numeroimalla erinäiset ongelmanratkaisuun liittyvät vaiheet. Oikella puolella oleva koodi on huomattavasti lyhyempi, mutta ei sisällä vastaavaa jaottelua. Siinä toteutettu numerointi vastaa ohjelmakoodin rivejä. Margulieux ym. (2012) esittämän teidon mukaan voidaan tehdä oletus, että vasemmalla oleva malliesimerkki auttaa skeeman rakentamisessa oikeanpuoleista paremmin, vaikka oikeanpuolimmainen on lyhyempi ja siten sen ulkoinen kuorma on vähäisempi.

Yhtenä mahdollisuutena on myös antaa oppilaille mahdollisuus nimetä osa-alueet itse. Tämä pakottaa oppilaat tukimaan osion rakennetta ja hakemaan syytä sen olemassaololle, vastaten kysymykseen: mitä tämän on tarkoitus tehdä? Kognitiivisen kuorman teoriassa tätä nimitetään *itse selityksen efektiksi* ja sen hyöty perustuu tarkkaavaisuuden kohdentamiseen oikeaan paikaan ja pintaa syvemmältä, mikä edesauttaa oppimista. Mielenkiintoista tässä teoriassa on, ettei se oikeastaan vähennä ylimääräistä kuormaa, vaan lisää sitä. Se kuitenkin auttaa oppilaita lisäämään hyödyllisen kuorman osuutta. (Sweller 2010.) Morrison ym. (2015) huomasivat kokeessaan, että välivaiheiden tekeminen tai oppiminen valmiista materiaalista, vei kauemmin aikaa, mutta auttoi oppimisessa.

3.3 Informaation esitystapa

Tässä käydään läpi kolme keinoa, joiden on todettu vähentävän ulkoista kognitiivista kuormaa tai hallitsevan luontaista kuormaa opetusmateriaalin esitystavassa.

3.3.1 Tarpeettoman koodin piilotus

Tehtävissä itsessään saattaa olla ylimääräistä kuormaa. Ohjelmoinnissa tälläinen tilanne tulee vastaan, jos tehtävä vaatii skeeman kannalta ylimääräistä ohjelmakoodia toimiakseen. Tästä esimerkki löytyy kuviosta 5, jossa opiskelijalle on annettu tehtäväksi tietynlaisen funtion toteuttaminen. Valmiiksi hänelle on annettu luokka, pääohjelma, funktion testit ja kirjastojen kutsut. Oppilas saa halutessaan nähdä koko ohjelmakoodin tai piilottaa sen. Tämä mahdollisuus auttaa kognitiivisen kuorman hallinnassa.

Kuvio 5: Jyväskylän yliopiston Ohjelmointi 1-kurssilta (Lappalainen ym. 2016). Molemmissa kuvissa on sama tehtävänanto. Vasemmalla on näkyvillä kaikki valmiiksi annettu ohjelmakoodi. Oikealla näkyy piilotettu versio ohjelmasta. Oppilas kykenee itse säätelemään näkymää. Kuvaa muokattu lisäämällä nappia havainnollistava punainen kehä.

Sama ideologia löytyy myös esimerkiksi Visual Studio ja Eclipse -ohjelmointiympäristöistä, joissa erilaiset osiot saavat omat “sulje ja aukaise” avaimensa. Tällöin ohjelmoija voi valintansa mukaan asettaa tarkastelun kohteeksi vain oleelliset elementit ja piilottaa epäolennaiset.

3.3.2 Tarkkaavaisuuden jakautuminen

Toinen keino on tarkkaavaisuuden jakautumisen efekti (engl. the split-attention effect), joka on osa kognitiivisen kuorman teoriaa (Chandler & Sweller 1991). Teorian mukaan diagrammi ja sitä selittävät tiedot tulisi pitää yhdessä, esimerkiksi siten,

että teksti on osa diagrammia. Tietoa ei siis saisi jakaa kahtia ja pitää osia erillään, sillä niiden yhdistämiseen kuuluu turhaa kognitiivista kuormaa. (Chandler & Sweller 1991.) Ohjelmointiin sovellettuna voitaisiin olettaa, että kun oppilaalle esitetään samalla ohjelmointikoodia ja sitä tarkastellaan metakielellä, tulisi ne asettaa mahdollisimman lähelle toisiaan. Ohjelmoinnissa tämä voitaisiin toteuttaa laittamalla metakieli kommentteihin koodin perään tai seuraavalle riville.

Hyvä tapa vähentää tarkkaavaisuuden jakautumista:

```
int a;     // määritellään muuttuja a
a = 3;    // alustetaan muuttujalle a arvoksi 3;
```

Ylläoleva ohjeistus on teorian mukaisesti parempi tapa kuin se, että koodiin lisätään rivinumerot ja koodia selittävä teksti on esitetty erillään koodista. Tämä korostuu, mitä enemmän koodia ja selitystä vaaditaan. Jos rivinumerointia kuitenkin käytetään, tulisi koodi jakaa pienempiin osioihin, joita seuraa heti selittävä tekstiosa.

Huono tapa esitellä ohjelmointikoodia:

```
1. int a;
2. a = 3;
Rivillä 1 määritellään muuttuja a. Sitten, rivillä kaksi alustetaan muuttujalle a arvoksi 3.;
```

Tarkkaavaisuuden jakautumisen efekti koskettaa kuitenkin vain sellaisia elementtejä, jotka kuuluvat yhteen, eikä kumpaakaan voi ymmärtää ilman toista elementtiä. Mikäli osat ovat ymmärrettävissä erikseen, ei tarkkaavaisuuden jakautumisella ole merkitystä. (Chandler & Sweller 1991.) Ideana ei siis suinkaan ole mahduttaa kaikkea tietoa tiiviiseen pakettiin, mahdollisimman lähelle toisiaan, vaan antaa opiskelijalle mahdollisuus yhdistää saumattomasti yhteenkuuluvat elementit, ilman että ne täytyy erikseen yhdistää eri osista tekstiä. Tämä yhdistäminen lisää kognitiivista kuormaa (Chandler & Sweller 1991).

3.3.3 Toiston poisto

Kolmas keino vähentää turhaa kognitiivista kuormaa on toiston poisto (engl. the redundancy effect) (Chandler & Sweller 1991). Tutkimusten mukaan oppilaat eivät

hyödy suoranaisesta toistosta, vaan se on parhaimmillaankin neutraalia kognitiivisen kuorman kannalta. Tämä tarkoittaa eritoten eksperttejä siinä mielessä, että heille ei kannata lisätä samoja ohjeistuksia kuin noviiseille. Työmuistia kuluu jo pelkästään siihen tarkasteluun, onko tieto oppilaalle tuttua vaiko ei. (Chandler & Sweller 1991.)

3.3.4 Tiedon jakaminen vaiheittain

Riippuen ohjelmointikielestä, noviisien kohdalla voi tulla vastaan tilanne, jossa pienenkin asian tekeminen vaatii paljon koodia. Esimerkiksi Java-kielessä tulostamiseen vaaditaan seuraava koodi:

```
public class Hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello World");
    }
}
```

Tällöin voi olla mielekästä selittää osa koodista myöhemmin. Koko ohjelmakoodi on itssesään luontaista kuormaa ja siten se tulee ymmärtää kokonaisuudessaan aikanaan, mutta ei välttämättä kerralla. Kuten Pollock ym. selittävät: *"understanding must be one of the ultimate goals of learning, it may not be necessary or indeed, possible, as an intermediate goal"* (Pollock ym. 2002). Ylläolevista esimerkeistä voidaan lähteä liikkeelle tulostuslauseen opettamisella. Luokka, saantimääreet, staattisuus, system-kirjasto, syntaksi, pääohjelma, paluuarvo, merkkijono ja muut elementit jäävät myöhempää selitystä varten.

Kognitiivisen kuorman kannalta tämä vähentää kuormaa huomattavasti. Ensinnäkin, sen hetkistä luontaista kuormaa rajoitetaan rajaamalla alue vain tulostuslauseeseen. Lisäksi, selkeällä skeemalla autetaan oppilasta keskittymään juuri tarvittavaan kohtaan sanomalla että mikä rivi on oleellinen, mitä se tekee ja miten.

4 Yhteenveto

Tutkimuskatsauksessa käytiin läpi kognitiivisen kuorman huomioimista ohjelmoinnin opetuksessa. Kuorman huomioiminen ohjelmoinnin oppimateriaalissa ei tee ohjelmoinnista helppoa, eikä se tarkoita, etteikö opetuksessa tulisi huomioida myös muut vaikeuksia tuottavat asiat, kuten ajan ja motivaation puutetta. Se saattaa kuitenkin edesauttaa oppimista vähentämällä ylimääräistä kognitiivista kuormaa ja vähentämällä henkisen ponnistelun määrää. Tämä auttaa eritoten niitä, jotka vaikuttavat jo tekevän kaikkensa ja silti saavat ohjelmoinnin kursseilla huonoja tuloksia (katso. Mason & Cooper 2012).

Käytännössä kuorman huomioiminen voi kuitenkin osoittautua haastavaksi, johtuen osaltaan siitä, että tarkkoja määrityksiä ei ole olemassa. Minkälainen tarkalleen ottaen on hyvä malliesimerkki mihinkin ongelmaan? Kuinka monta malliesimerkkiä tulisi esitellä, jotta niiden variaatio olisi tarpeeksi laaja, mutta jotta kuorma ja muistin rajat eivät ylity? Kuinka paljon esimerkkien tulisi erota toisistaan, jotta oppilas vielä osaa yhdistää ne samaan kategoriaan? Entä minkälainen on hyvä malliesimerkkiä seuraava ongelmanratkaisutehtävä? Nämä ovat hyviä kysymyksiä, sillä väärin sovellettuna kognitiivisen kuorman teoriasta voi olla jopa haittaa opiskelussa. Kognitiivisen teorian mukaan oppilaan taitotaso on tärkeä tekijä ja tulee ottaa huomioon materiaalissa. Käytännössä tämä on lähes mahdotonta, sillä opetukseen osallistuvilla on lähes väistämättä jonkinlaisia eroja aikaisemmissa tiedoissaan.

Opetusmateriaalia ei välttämättä pysty viilaamaan täysin optimaaliseksi, jotta se sisältää juuri oikean määrän ulkoista kuormaa jokaiselle yksilölle. Teoriaa voi kuitenkin pitää enemmänkin yleisenä ohjenuorana. Mason & Cooper (2012) ehdottavatkin, että skeeman rakentamista voi edesauttaa sillä yksinkertaisella asialla, että määrittelee opetettavat asiat selkeästi ja keskittyy niiden opettamiseen. Opetusmateriaalin tekijä voi pohtia, mikä tässä tehtävässä on tärkeintä oppia ja onko se sama asia mihin oppilas tulee keskittymään tehtävää tehdessään. Tätä yleistä ohjetta voi soveltaa kaikkiin opetusasteisiin. Tarvitaan kuitenkin lisätietoa enemmän siitä, kuinka kognitiivinen kuorma voidaan huomioida paremmin erilaisilla luokka-asteilla.

Kirjallisuutta

Blackwell, A. F. (2002, June). *What is programming*. In 14th workshop of the Psychology of Programming Interest Group (pp. 204-218).

Chandler, Paul, and John Sweller. *Cognitive load theory and the format of instruction*. Cognition and instruction 8.4 (1991): 293-332.

Fleischman, E. S., & Jones, R. M. (2002). *Why example fading works: A qualitative analysis using Cascade.* In Proceedings of the 24th Annual Conference of the Cognitive Science Society (pp. 298-303).

Hoc, J.-M., Green, T.R.G., Samurcay, R. and Gilmore, D.J, (1990) *Psychology of programming*. London : Academic Press cop.

Kaijanaho, A-J (2010). *Ohjelmointikielten periaatteet*, Luentomoniste. Jyväskylän yliopisto, Tietotekniikan laitos.

Kalyuga, S., Ayres, P., Chandler, P.,& Sweller, J. (2003). *The expertise reversal effect*. Educational psychologist, 38(1), 23-31.

Kinnunen, P., & Malmi, L. (2006). *Why students drop out CS1 course?*. In Proceedings of the second international workshop on Computing education research (pp. 97-108). ACM.

Lakanen, A.-J., & Lappalainen, V. (2014). *What Students Think About Game-Themed CS1*. In M. Koskela,& K. Heikkinen (Eds.), Yhdistetyt tietojenkäsittelypäivät 2014 F̄ederated Computer Science Event 2014 : 3.-4.6.2014 Lappeenranta, Finland (pp. 19-22). LUT Scientific and Expertise publications. Tutkimusraportit (24). Lappeenranta: Lappeenrannan teknillinen yliopisto.

Lappalainen ym. Jyväskylän yliopisto (2016) *Ohjelmointi 1*. Luentomoniste. Nähtävillä: `<https://tim.jyu.fi/view/1>` Haettu 30.4.2016.

Margulieux, L. E., Guzdial, M., & Catrambone, R. (2012, September). *Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications.* In Proceedings of the ninth annual international conference on International computing education research (pp. 71-78). ACM.

Mason & Cooper. 2012.*Why the bottom 10% just can't do it: mental effort measures and implication for introductory programming courses.* Proceedings of the Fourteenth

Australasian Computing Education

Subgoals, Context, and Worked Examples in Learning Computing Problem Solving Morrison, B. B., Margulieux, L. E., & Guzdial, M. (2015). Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In Proceedings of the eleventh annual International Conference on International Computing Education Research (pp. 21-29). ACM.

Opetushallitus. (2016). Ops 1026. Viitattu. 23.03.2016. Näkyvillä verkossa os *http://www.oph.fi/ops2016*

Paas, F. G., & Van Merriënboer, J. J. (1994). *Instructional control of cognitive load in the training of complex cognitive tasks*. Educational psychology review, 6(4), 351-371.

Paas, F., Renkl, A., & Sweller, J. (2004). *Cognitive load theory: Instructional implications of the interaction between information structures and cognitive architecture.* Instructional science, 32(1), 1-8.

Piaget, J. (1952). *Origins of intelligence in children* (Vol. 8, No. 5, pp. 18-1952). New York: International Universities Press.

Pollock, E., Chandler, P., & Sweller, J. (2002). *Assimilating complex information*. Learning and instruction, 12(1), 61-86.

Renkl, A., Atkinson, R. K., Maier, U. H., & Staley, R. (2002). *From example study to problem solving: Smooth transitions help learning.* The Journal of Experimental Education, 70(4), 293-315.

Renkl, A., Atkinson, R. K., & Große, C. S. 2004.*How fading worked solution steps works a –cognitive load perspective.* Instructional Science, 32(1-2), 59-82.

Salminen, A. (2011). *Mikä kirjallisuuskatsaus?* Johdatus kirjallisuuskatsauksen tyyppeihin ja hallintotieteellisiin sovelluksiin. Vaasan yliopiston julkaisuja. Opetusjulkaisuja 62, Julkisjohtaminen 4.

Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., ... & Brechmann, A. (2014, May). *Understanding understanding source code with functional magnetic resonance imaging*. In Proceedings of the 36th International Conference on Software Engineering (pp. 378-389). ACM.

Skudder, B., & Luxton-Reilly, A. (2014, January). *Worked examples in computer science.* In Proceedings of the Sixteenth Australasian Computing Education Conference-

Volume 148. Australian Computer Society.

Sweller, J., & Cooper, G. A. (1985). *The use of worked examples as a substitute for problem solving in learning algebra*. Cognition and Instruction, 2(1), 59-89.

Sweller, J. (1988). *Cognitive load during problem solving: Effects on learning.* Cognitive science, 12(2), 257-285.

Sweller, J. 2010. *Element interactivity and intrinsic, extraneous, and germane cognitive load. Educational psychology review*, 22(2), 123-138.

Steele, G. L. (1999). *Growing a language*. Higher-Order and Symbolic Computation, 12(3), 221-236.

Tirronen, V. Jyväskylän yliopisto *Funktio-ohjelmointi -kurssimateriaali, 2016*. Nähtävillä osoitteessa: `<http://functional-programming.it.jyu.fi/pages/Functions.md#WhichAreFunctions>` Haettu 15.4.2016.