Muuttujien roolit ohjelmoinnin alkeiden opetuksessa

Pauli Harjumäki

31.8.2004

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

Tiivistelmä

Tässä työssä tarkastellaan ohjelmoinnin alkeiden oppimista ja opettamista ja esitellään uusi erityisesti ohjelmoinnin alkeiden opettamiseen soveltuva käsite eli muuttujan roolit, roolipohjainen animaattori PlanAni sekä näiden hyödyntämiseen ohjelmoinnin alkeiden opettamisessa liittyvä empiirinen koe.

Uusien asioiden oppiminen on mahdollista monimutkaisten kognitiivisten prosessien ansiosta. Tietämys jostakin uudesta asiasta karttuu vähitellen tiettyjen rajallisten resurssien ja lainalaisuuksien puitteissa. Näiden seikkojen esittely toimii pohjustuksena ohjelmoinnin oppimiseen liittyvälle tarkastelulle ja samalla perehdytään joihinkin kognitiivisen psykologian peruskäsitteisiin.

Ohjelmoinnin oppiminen on hyvä esimerkki monimutkaisen taidon oppimisesta. Tavoitteena on tuoda esille tämänhetkinen käsitys ohjelmoinnin eri osa-alueista ohjelmoinnin psykologiaan liittyviä teorioita läpikäymällä. Lisäksi selvitetään keskeisimmät noviisien ohjelmoinnin oppimiseen liittyvät ongelmat ja luodaan katsaus siihen, miten erilaisilla lähestymistavoilla ohjelmoinnin alkeita voidaan opettaa.

Muuttujien roolit on uusi ohjelmoinnin alkeiden opettamiseen soveltuva käsite, jonka avulla päästään käsiksi kokeneempien ohjelmoijien piilevässä muodossa olevaan tietämykseen. Roolipohjaista PlanAni-animaattoria käyttämällä voidaan visualisoida muuttujien rooleja. Muuttujien roolien ja PlanAnin käyttämisen vaikutuksia ohjelmoinnin alkeiden oppimiseen tutkittiin empiirisessä kokeessa. Kokeesta saatujen tulosten mukaan muuttujien roolien käyttäminen ohjelmoinnin alkeiden opetuksessa auttaa noviiseja ymmärtämään ohjelmien syvärakenteita. Tämä on merkittävää siksi, että ohjelman syvärakenteisiin liittyvä tietämys on merkki hyvästä lähdekoodin ymmärtämisestä. PlanAni-animaattorin käyttäminen näyttää lisäävän muuttujiin liittyvää tietämystä. Tämä näkyy käytännössä pienempänä virheiden määränä muuttujien käyttöön liittyen ohjelmien kirjoittamisen yhteydessä.

ACM-luokat (ACM Computing Classification System, 1998 version): A.m, K.3.2

Avainsanat: muuttujien roolit, noviisi, ekspertti, PlanAni, ohjelmoinnin alkeet, ohjelmoinnin opetus

Esipuhe

Monimutkaisten asioiden syvällinen ymmärtäminen vaatii aikaa ja tämän työn tekeminen on ollut siitä erittäin hyvä esimerkki. Aihetta valitessani en vielä ymmärtänyt, miten suurta pohjatietämystä ja paneutumista sen menestyksekäs käsittely vaatisi. Voin rehellisesti sanoa, että tämän työn loppuun saattaminen on vaatinut melkoisen määrän ponnistelua, mutta siinä samalla olen oppinut koko joukon asioita, joista on varmasti hyötyä muuallakin kuin tietojenkäsittelyn alalla.

Haluan kiittää ohjaajaani professori Marja Kuittista sekä professori Jorma Sajaniemeä kärsivällisyydestä ja hyvistä neuvoista. Vilpittömät ja valtaisan kokoiset kiitokset vanhemmilleni, ystävilleni ja tietysti Tinkalle, joka tuki minua henkisesti ja auttoi kuvien tekemisessä.

31.8. 2004 Joensuu

Pauli Harjumäki

Sisältö

1 Johdanto

Teknisen kehityksen myötä erilaisten ohjelmistojen tarve on kasvanut voimakkaasti. Emme useimmiten tule edes ajatelleeksi, miten laajalti tietokoneet pyörittävät jokapäiväistä elämäämme. Laajalti käytössä olevien ja laadukkaiden ohjelmistojen suunnittelu, toteutus ja ylläpito on vaativaa työtä, sillä ohjelmistojen koon ja monimutkaisuuden kasvun takia ihminen joutuu työskentelemään niiden parissa usein kykyjensä äärirajoilla.

Ohjelmoinnista on tullut sen merkityksen kasvun myötä laajan tutkimuksen kohde. Ohjelmoinnin psykologia on tutkimushaara, joka tutkii ohjelmointia ja siihen liittyviä asioita kognitiivisen psykologian näkökulmasta. Keskeisimmät tavoitteet ovat yleisellä tasolla ilmaistuna paitsi ohjelmistotyön tehostaminen, myös ihmisen kognition parempi ymmärtäminen. Ohjelmoinnin alkeisiin liittyvän tutkimuksen pyrkimyksenä on ohjelmoinnin ja sen oppimiseen liittyvän pitkän oppimiskaaren helpottaminen eri tavoin. Taitavinkin ohjelmoija aloittaa ohjelmoinnin opettelun alkeista, ja ohjelmoinnin alkeiden oppimiseen ja opettamiseen liittyy myös tämän työn aihe.

Tässä työssä tarkastellaan ohjelmoinnin alkeiden oppimista ja opettamista ja esitellään uusi erityisesti ohjelmoinnin alkeiden opettamiseen tarkoitettu käsite, muuttujan roolit sekä niiden hyödyntämiseen liittyvä empiirinen koe.

Luku kaksi toimii pohjustuksena ohjelmoinnin oppimiseen ja opettamiseen liittyville asioille. Tavoitteena on selvittää lyhyesti oppimiskäsityksiä, muistin toimintaa, miten oppiminen ja ongelmanratkaisu tapahtuu yleisellä tasolla ja miten tietämys jostakin asiasta kehittyy noviisin tasolta ekspertiksi sekä perehtyä samalla joihinkin kognitiivisen psykologian peruskäsitteisiin.

Seuraavassa luvussa tarkastelun kohteeksi otetaan varsinaiset ohjelmoinnin oppimiseen ja opettamiseen liittyvät seikat. Tässä yhteydessä luodaan yleiskatsaus siihen, mitä ohjelmoinnin eri osa-alueisiin liittyvistä kognitiivisista prosesseista ja noviisien vaikeuksista tiedetään ja miten eri tavoin ohjelmointia voidaan opettaa.

Luvussa neljä esitellään muuttujien roolit ja käydään läpi joitakin muuttujien roolien ominaisuuksia. Lisäksi tutustutaan roolipohjaiseen animaattoriin PlanAniin sekä perehdytään siihen, miten muuttujien rooleja voidaan opettaa ohjelmoinnin alkeiskurssin yhteydessä. Lopuksi viidennessä luvussa tarkastellaan empiiristä koetta, jossa tutkittiin

muuttujien roolien ja PlanAnin käyttämisen vaikutuksia ohjelmoinnin alkeiden oppimiseen. Läpi käydään paitsi kokeen suorittamiseen liittyviä seikkoja, myös siitä saadut tulokset ja niiden analysointi.

2 Oppimisprosessi yleisellä tasolla

Tämän luvun tarkoituksena on tarkastella yleisellä tasolla oppimisprosessia ja ekspertiyden kehittymistä, jotta saataisiin tarvittava pohjatietämys seuraavissa luvuissa olevaa ohjelmoinnin oppimisen ja opettamisen tarkastelua varten. Tarkastelu aloitetaan oppimiskäsityksistä, minkä jälkeen perehdytään mm. aistimusten prosessointiin, muistin toimintaan, tarkkaavaisuuteen sekä ongelmanratkaisuheuristiikkoihin. Luvun päätteeksi tarkastellaan taidon oppimisen vaiheita ja kootaan yhteen noviisin ja ekspertin välisiä eroavaisuuksia.

Ennen kuin tarkastellaan oppimiskäsityksiä sekä itse oppimisprosessia ja eksperttiyden kehittymistä, määritellään noviisi ja ekspertti. *Noviisi* on henkilö, jonka tiedot ja taidot jollakin tietyllä alalla eivät ole vielä täysin kehittyneet tai henkilö, joka on vasta opettelemassa oman alansa taitoja eikä omaa paljon alan työkokemusta. *Ekspertti* puolestaan on oman alansa ammattilainen, joka on osaava, taitava ja omaa pitkän työkokemuksen omalta erikoisalaltaan (Eteläpelto, 1996). Noviisin ja ekspertin välinen ero näyttää selkeältä, mutta millä tavoin ja mistä johtuen noviisi ja ekspertti eroavat toisistaan? Jotta näihin kysymyksiin voisi vastata syvällisemmällä tasolla, on syytä tarkastella lähemmin sitä, miten oppiminen tapahtuu eri oppimiskäsitysten mukaan sekä perehtyä muistiin ja ongelmanratkaisuun liittyviin peruskäsitteisiin. Ellei muuta ole tässä luvussa mainittu, on lähteenä ollut Anderson (1985).

2.1 Oppimiskäsityksistä

Oppimiseen liittyen on olemassa erilaisia käsityksiä, jotka tarkastelevat tietoa ja sitä miten oppiminen tapahtuu eri tavoin. Kaksi laajalle levinnyttä ja tämän työn kannalta olennaisinta oppimiskäsitystä ovat *behaviorismi* ja *konstruktivismi*. Seuraavassa on lyhyet kuvaukset kummankin oppimiskäsityksen olennaisimmista piirteistä. Lähteenä oppimiskäsityksissä kertovassa osuudessa on ollut Tynjälä (2000).

2.1.1 Behaviorismi

Behaviorismi perustuu objektivistiseen ja empiristiseen näkemykseen tiedosta. *Objektivistisen näkemyksen* mukaan ihminen voi saavuttaa maailmasta ja sen ilmiöistä ob-

jektiivista tietoa, joka on riippumatonta tietävästä subjektista, eli kyseisestä henkilöstä. Objektivismiin liittyy myös ajatus, että tietoa voidaan saavuttaa vain kokemuksen ja havaintojen kautta. *Empiristinen näkemys* puolestaan korostaa todellisuuden erillisyyttä havainnoitsijasta sekä kokemusta ja havaintoja ainoina tiedon lähteinä. Behavioristinen näkemys ei hyväksy mitään ihmisten mentaalisten prosessien tutkimista, vaan keskittyy tarkastelemaan oppimista ainoastaan ulkoisen käyttäytymisen perusteella. Behavioristinen näkemys oppimisesta vallitsi aina 1950- ja 1960-lukujen vaihteeseen asti.

Oppimisessa on behaviorismin mukaan kyse ärsyke-reaktiokytkentöjen muodostumisesta, jota voidaan säädellä vahvistamisella. Tällöin oletetaan, että opetuksen tavoitteena oleva reaktio (R) vakiintuu pysyväksi käyttäytymiseksi, kun se yhdistetään ympäristöstä tulevaan ärsykkeeseen (S). Käyttäytymistä säätelevä vahvistaminen suoritetaan oppijan ulkopuolelta. Jos tietynkaltaista reaktiota halutaan vahvistaa, annetaan siitä palkkio, päinvastaisesta tai muuten ei-toivotusta reaktiosta annetaan rangaistus, jolloin kyseinen reaktio heikkenee. Vahvistaminen on siis oppimisen kannalta keskeisessä asemassa.

Esimerkkinä behavioristisen oppimiskäsityksen soveltamisesta käy eläinten kouluttaminen: halutunlaisesta käyttäytymisestä eläimelle annetaan palkkio esimerkiksi ruoan muodossa. Vastaavalla tavalla myös ihmisen käyttäytyminen voidaan behaviorismin mukaan nähdä ympäristöärsykkeiden ja reaktioiden assosiaatioina. Behaviorismin mukaan opettamisessa on kyse ennen kaikkea valmiin tiedon siirtämisestä. Jakamalla tieto sopivan kokoisiin pienempiin elementteihin se voidaan ikään kuin siirtää opiskelijoiden päähän. Oppimistulosten arviointi tapahtuu tällöin määrällisesti, eli oppijan katsotaan oppineen asian sitä paremmin, mitä enemmän hän pystyy toistamaan opetettua tietoa kokeessa tai tentissä.

Valmiiksi määriteltyjen tietoelementtien ulkoa opettelu varsinaista sisältöä pohtimatta ei ole riittävä lähtökohta ohjelmoinnin opettamiselle ja oppimiselle. Ohjelmoinnissa on pitkälti kyse aktiivisesta ongelmanratkaisusta ja tiedon soveltamisesta uusiin tilanteisiin, ja tässä suhteessa behavioristisessa oppimiskäsityksessä on selkeitä puutteita. Käytännössä tämä tarkoittaa, että vaikkapa ohjelmointikurssilla pelkän ohjelmointikielen syntaksin ulkoa opettelu ei varmastikaan riitä käytännön ohjelmointitaidon saavuttamiseen. Luonnontieteellisten aineiden opettamiseen konstruktivismi näyttääkin tarjoavan sopivamman lähestymistavan.

2.1.2 Konstruktivismi

Konstruktivismi ei ole yhtenäinen teoria, vaan sillä on useita eri suuntauksia, jotka painottavat eri asioita. Konstruktivismi itsessään ei myöskään ole oppimisteoria, vaan tiedon olemusta käsittelevä paradigma, joka on laajalti käytössä yhteiskunta- ja ihmistieteissä. Konstruktivistinen oppimiskäsitys on tämän paradigman ilmenemismuoto oppimisen tutkimuksen sekä pedagogiikan aloilla. Konstruktivismin eri suuntauksille on yhteistä niiden käsitys tiedosta. Tieto ei koskaan voi olla riippumatonta objektiivista heijastumaa maailmasta, vaan se on aina yksilön tai yhteisöjen rakentamaa. Toisin sanoen konstruktivismi ei hyväksy behavioristista näkökantaa, jonka mukaan maailmasta voi saada objektiivista tietoa havaintojen välityksellä.

Konstruktivistisen oppimiskäsityksen mukaan oppimisessa on kyse oppijan aktiivisesta kognitiivisesta toiminnasta, jossa oppija tulkitsee havaintojaan ja uutta tietoa aikaisempaan tietoon ja kokemuksiin pohjautuen. Täten oppija rakentaa jatkuvasti käsitystään maailmasta ja siihen liittyvistä ilmiöistä.

Konstruktivismin tärkeimmät suuntaukset ovat karkealla tasolla ilmaistuna *sosiaalinen konstruktivismi* ja *yksilökonstruktivismi*. Tämän työn kannalta olennaisin on yksilökonstruktivismia edustava suuntaus eli *radikaali konstruktivismi*, jonka oppimispsykologista suuntausta kutsutaan *kognitiiviseksi konstruktivismiksi*. Radikaalin konstruktivismin mukaan tiedon todellisuus testataan käytännössä. Jos tieto osoittautuu käyttökelpoiseksi ja on avuksi toiminnassamme, sitä voidaan pitää totena. Mikäli tieto ei palvele käytäntöä, sen arvo on kyseenalainen. Kognitiivisen konstruktivismin keskeisimpiä käsitteitä ovat skeema, assimilaatio ja akkommodaatio. *Skeemoja* käsitellään tarkemmin myöhemmin tässä luvussa, mutta tässä yhteydessä mainittakoon lyhyesti skeemojen olevan muistissa olevia jäsentyneitä tietorakenteita, joihin ovat tallentuneet ihmisen muodostamat sisäiset mallit ympäröivästä todellisuudesta. *Assimilaatio* tarkoittaa kognitiivista mekanismia, jossa uusi havainto, tieto tai kokemus liitetään jo olemassa olevaan skeemaan. *Akkommodaation* tapauksessa tehdyt havainnot ja kokemukset eivät sovellu olemassa oleviin skeemoihin, vaan niitä on mukautettava ja muovattava siten, että syntyy uudenlainen tietorakenne. Akkommodaation merkitys oppimisessa onkin tärkeä, sillä sen kautta oma ajattelutapamme ja maailmankuvamme muuttuvat. Radikaalin konstruktivismin periaatteita sovelletaan nykyään laajalti etenkin matematiikan ja luonnontieteellisten aineiden opetuksessa. Myös tämän työn yhteydessä aihetta tarkastellaan konstruktivistisesta näkökulmasta.

2.2 Näkö- ja kuuloaistimusten alustava prosessointi

Ihmismieli jäsentelee ympäristöstä tekemänsä näköaistimukset mielekkäiksi yksiköiksi tiettyjen periaatteiden mukaisesti, joista tunnetuimpia ovat niin sanotut *hahmolait* (Gestalt principles of perceptual organization). Hahmolait ovat nimeltään *läheisyys* (proximity), *samanlaisuus* (similarity), *sulkeutuvuus* (closure), *jatkuvuus* (continuity), *alue* (area) ja *symmetria* (symmetry). Muun muassa näitä lakeja hyväksi käyttämällä näköaistimukset ryhmitellään alustavasti ja ne tallentuvat visuaaliseen sensoriseen muistiin odottamaan mahdollista jatkoprosessointia.

2.2.1 Sensoriset muistit: visuaalinen sensorinen muisti ja ekoinen muisti

Sensoriset muistit (sensory memory) ovat aistimusten tallentamiseen tarkoitettuja muistijärjestelmiä, joihin alustavasti prosessoidut aistimukset tallentuvat enintään muutaman sekunnin ajaksi odottamaan mahdollista jatkoprosessointia. Aistimukset tallentuvat sensorisiin muisteihin tiedostamattomasti. Kuuloaistiin liittyvästä sensorisesta muistista (auditory sensory memory) puhutaan monilla eri nimillä. Käytössä ovat ainakin *sensorinen kuulomuisti*, *kaikumuisti* ja *ekoinen muisti*, joista viimeksi mainittua käytetään tämän työn yhteydessä. Näköaistiin liittyvää sensorista muistia (visual sensory memory) eli *visuaalista sensorista muistia* kutsutaan myös *ikonimuistiksi*. Muillakin aisteilla kuin näöllä ja kuulolla on omat sensoriset muistinsa, mutta tässä yhteydessä tarkastellaan ainoastaan näkö- ja kuuloaistimusten prosessointia. Sensorisissa muisteissa oleva informaatio unohtuu nopeasti, koska ympäristöstä tulee jatkuvasti uutta tietoa, jolloin vanha tieto vaimenee ja unohtuu vähitellen. Jotta sensorisissa muisteissa olevasta tiedosta saisi jotakin irti, pitää siihen kiinnittää tarkkaavaisuutta, muuten tieto ehtii hävitä.

2.2.2 Tarkkaavaisuus aistimusten tunnistamisen yhteydessä

Sensorisissa muisteissa oleva informaatio täytyy prosessoida tietoisesti, jotta se pääsisi lyhytkestoiseen muistiin. Sensorisiin muisteihin tulevan informaation määrä on kuitenkin niin suuri, että on tietoisesti valittava, mihin informaatioon keskittyy. Informaation analysoimiseen vaaditaan *tarkkaavaisuutta* (attention). Se on rajallinen resurssi, jota eri asioihin kohdistamalla ihminen voi tehdä havaintoja ympäröivästä maailmasta,

käsitellä siihen liittyviä tietoja ja suorittaa erilaisia kognitiivisia prosesseja. Tarkkaavaisuuden kohteita voi olla samanaikaisesti useita, mutta useamman paljon tarkkaavaisuutta vaativan prosessin samanaikainen menestyksekäs suorittaminen on mahdotonta. Tarkkaavaisuutta tarvitaan paitsi aistimusten käsittelemisessä, niin myös uusien asioiden oppimisessa ja jokapäiväisessä elämässä. Käytössä olevaa tietoutta ja kykyjä eri tavoin yhdistelemällä ja käyttämällä pyritään pääsemään haluttuun lopputulokseen.

2.2.3 Aistimusten tunnistamismekanismeja

Tarkkaavaisuuden lisäksi aistimusten tunnistamisessa tarvitaan jokin keino tunnistaa aisteista tulevaa informaatiota. Näköaistimusten yhteydessä keinona on *hahmontunnistus* ja kuuloaistimusten yhteydessä esimerkiksi *puheentunnistus*. Visuaalisessa sensorisessa muistissa olevat havainnot, jotka on alustavasti käsitelty hahmolakeja käyttämällä, prosessoidaan tässä vaiheessa syvällisemmin lopullisen hahmontunnistuksen saavuttamiseksi. Tähän prosessointiin liittyen on laadittu useampiakin teorioita, joista tässä yhteydessä mainitaan lyhyesti mallikuvion sovitusteoria ja piirreanalyysiteoria.

Mallikuvion sovitusteoriassa (template matching) ajatuksena on, että aivoissa on joukko malleja, joihin kulloinkin tunnistettavaa hahmoa yritetään sovittaa. Valituksi tulee se malli, joka parhaiten vastaa sovitusta. Sovitus voi epäonnistua helposti, jos hahmo esimerkiksi poikkeaa aivoissa olevasta mallista eikä tunneta menetelmää sovituksen parantamiseksi.

Piirreanalyysiteoriassa (feature analysis) tunnistettavan hahmon ajatellaan koostuvan tietyistä piirteistä. Muistissa oleva malli koostuu näistä piirteistä ja kuvauksesta, joka kertoo, miten nämä piirteet on yhdistettävä sekä mitkä seikat ovat tärkeimpiä tässä yhdistämisessä. Myös puheentunnistus (speech recognition) voidaan selittää piirreanalyysin avulla. Yksittäisillä foneemeilla (pienimmillä merkityksiä erottavilla yksiköillä sanoja muodostettaessa) on joukko ominaisuuksia, joiden avulla ne voidaan tunnistaa. Näitä ovat esimerkiksi foneemin ääntämispaikka, onko kyseessä konsonantti vaiko vokaali jne. Mitä enemmän foneemeilla on toisistaan eroavia ominaisuuksia, sitä helpompi ne on erottaa toisistaan ja luonnollisesti myös päinvastoin.

Tutkimuksissa on todettu, että sekä näköaistimusten että kuuloaistimusten prosessoinnissa asiayhteydellä on suuri merkitys. Aivot pyrkivät automaattisesti ”korjaamaan” havaittuja aistimuksia muotoon, joka vaikuttaa kyseisessä asiayhteydessä järkevältä.

Kun hahmontunnistus tai esimerkiksi äänen- tai puheentunnistus on suoritettu, tunnistettu havainto on käytettävissä lyhytkestoisessa muistissa.

2.3 Havaintopohjainen ja merkityspohjainen tiedon esitys sekä erilaiset muistirakenteet

Keskeinen käsite tiedon käsittelyssä on *mieltämisyksikkö* (mental chunk). Yksittäinen mieltämisyksikkö voi olla melkein mitä tahansa: kirjain, sana tai joukko sanoja, lause, numero, numerosarja, mielikuva jne. On yritetty selvittää, mikä on suurin määrä mieltämisyksiköitä, joita voidaan aktiivisesti käsitellä muistissa ja päädytty noin 5-9 (Andersonin (1985) mukaan Miller (1956)) tai 3-4 (Andersonin (1985) mukaan Broadbent (1975); Simon (1974)) mieltämisyksikköön. Tarkka määrittäminen on kuitenkin osoittautunut ongelmalliseksi, koska mieltämisyksikölle on vaikea antaa yksiselitteistä määritelmää, ja koska ihmismuisti toimii siten, että määrää on vaikea selvittää. Joka tapauksessa on selvää, että aivot pyrkivät vähentämään muistin kuormitusta muuttamalla suuria tietomääriä yhdeksi kokonaiseksi mieltämisyksiköksi, jolloin on mahdollista käsitellä enemmän tietoa kerralla. Tämä saavutetaan opiskelun ja kokemuksen kautta, ja se liittyy olennaisella tavalla myös eksperttiyden kehittymiseen. Missä muodossa tehtyjä havaintoja ja tietoa käsitellään ihmisen aivoissa? Tiedosta on olemassa kahdentyyppisiä esityksiä: *havaintopohjaisia* (perception-based knowledge representations) ja *merkityspohjaisia* (meaning-based knowledge representations). Aistihavainnoista on aluksi olemassa sekä havaintopohjainen että merkityspohjainen esitys. Havaintopohjainen esitys unohtuu kuitenkin nopeasti ja jäljelle jää havainnoitsijan tulkinta siitä, mitä kyseiset tapahtumat hänelle merkitsevät eli merkityspohjainen esitys.

2.3.1 Havaintopohjainen tiedon esitys

Havaintopohjainen tiedon esitys voi esiintyä joko *mielikuvina* (images) tai *lineaarisina listoina* (linear orderings). Mielikuvien avulla voidaan koodata esineiden avaruusrakenne, mutta vaikka ihminen näyttää muistavan hyvin erilaisia valokuvia tai tietoa esitettynä graafisessa muodossa, mielikuvilla ei tarkoiteta ”päässä olevia valokuvia”. Sen sijaan on kyse hierarkkisen rakenteen omaavasta kuvan tulkinnasta, joka ei kuitenkaan sisällä kaikkia mahdollisia kuvassa olevia yksityiskohtia. Mielikuvien avulla voidaan prosessoida avaruudellisia asioita sekä suorittaa niillä erilaisia operaatioita,

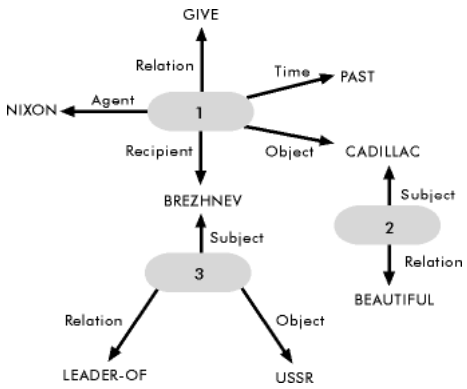
kuten esimerkiksi pyöritystä, kääntämistä tai sisällön muokkaamista.

Lineaarisia listoja puolestaan käytetään asioiden järjestyksen esittämiseen. Jos joku kysyy aakkosten kymmenettä kirjainta, useimmat alkavat käydä mielessään aakkosia läpi ensimmäisestä kirjaimesta alkaen, kunnes päätyvät kymmenenteen kirjaimeen. Tässä on kyse lineaarisesta listasta. Lineaarisilla listoilla on joitakin erityispiirteitä. On havaittu, että niitä yleensä käydään läpi mielessä alusta loppuun ja että listan ensimmäinen ja viimeinen alkio ovat nopeimmin saatavilla. Jos listan alkiot ovat lähellä toisiaan (esimerkiksi numerot 5 ja 6), niin niiden keskinäisen järjestyksen selvittämiseen kuluu enemmän aikaa kuin alkioilla, jotka ovat kauempana toisistaan (esimerkiksi numerot 5 ja 1000). Myös lineaarisella listalla on hierarkkinen rakenne: esimerkiksi aakkosjärjestys muistetaan pieninä palasina, joissa kirjaimet jakautuvat ryhmiin (abcd). Aakkosjärjestys muodostetaan koostamalla mielessä peräkkäin näitä kirjainryhmiä.

2.3.2 Merkityspohjainen tiedon esitys

Ihmisaivot ovat huonoja muistamaan täsmällisiä yksityiskohtia. Erilaisten tapahtumien sanalliset ja kuvalliset yksityiskohdat muistetaan noin minuutin ajan, mutta tämän jälkeen ne unohtuvat ja jäljelle jää ainoastaan havainnoitsijan tulkinta siitä, mitä kyseiset tapahtumat hänelle merkitsevät. Merkityksille käytetty muisti on siis kestävämpää kuin yksityiskohdille tarkoitettu muisti. Käytännössä tämä tarkoittaa, että yksityiskohtien muistamista voidaan parantaa, mikäli muistettavat yksityiskohdat muutetaan johonkin itselle merkitykselliseen muotoon. Merkityksiä voidaan esittää semanttisten verkkojen ja skeemojen sekä skeeman erikoistapauksen eli skriptien avulla.

Semanttinen verkko (propositional network) kuvaa johonkin tietoon liittyvät käsitteet ja niiden väliset assosiatiiviset suhteet. Mitä lähempänä toisiaan käsitteet ovat semanttisessa verkossa, sitä parempia vihjeitä ne ovat palautettaessa mieleen viereisiä käsitteitä. Myös semanttinen verkko voi olla hierarkkinen. Kuvassa 1 on esimerkki semanttisesta verkosta, joka on rakentunut lauseesta ”*Nixon gave a beautiful Cadillac to Brezhnev, who is leader of the USSR.*”. Kyseinen lause voidaan jakaa kolmeen yksinkertaisempaan lauseeseen, joiden merkitysten avulla lause koodataan semanttisen verkon muotoon. *1. Nixon gave a Cadillac to Brezhnev. 2. The Cadillac was beautiful. 3. Brezhnev is leader of the USSR.* Mikäli esitettävä tiedon määrä on suuri, on semanttinen verkko kuitenkin yksinään epäkäytännöllinen tapa esittää merkityksiä.



Kuva 1: Semanttinen verkko Andersonin (1985) mukaan.

Semanttisten verkkojen lisäksi merkityksiä voidaan esittää myös skeemojen avulla. *Skeemat* (schemas) ovat koottuja ja jäsenneltyjä esityksiä jostakin tietystä toimintakokonaisuudesta tai käsitteestä. Toisin sanoen ne ovat sisäisiä malleja ja yksinkertaistuksia todellisen maailman ilmiöistä ja asioista. Skeemoihin ovat varastoituneina aikaisemmin koetut kokemukset sekä tiedot ja ne vaikuttavat siihen, miten ja minkälaista tietoa ihminen voi ottaa vastaan ja miten uusi tieto liitetään vanhaan, jo olemassa olevaan tietoon. Skeemat siis muuttuvat ja kehittyvät oppimiskokemusten mukana, mutta ovat myös ”syyllisiä” siihen, että oppiminen on luonteeltaan myös poisoppimista, toisin sanoen vanha tieto haittaa osittain uuden tiedon omaksumista. Taulukossa 1 on yksinkertainen esimerkkiskeema käsitteestä ”appelsiini”.

Taulukko 1: Esimerkkiskeema appelsiinista.

käsite:	appelsiini
yläkäsite:	sitrushedelmä
muoto:	pyöreä
väri:	oranssi
paino:	200-400 g
maku:	raikas
kasvaa:	jossakin ulkomailla

Skeema kokoaa kuhunkin asiaan liittyvät tiedot attribuutteina. Attribuuttien arvoina voi olla myös käsitteitä, joilla on oma skeema, ja attribuuttien skeemoja tutkimalla taas voidaan johtaa uutta tietoa, jota ei muuten olisi suoraan olemassa. *Skriptit* (scripts) ovat skeemoja, jotka kuvaavat jonkin stereotyyppisen tapahtumasarjan, kuten kaupassakäynnin eri vaiheet: kauppaan menemisen, ostosten valitsemisen, maksamisen sekä kotiin palaamisen. Vaikka ei olisi koskaan käynyt lemmikkieläinkaupassa, kaikki tie-

tävät skriptin ansiosta, miten asioiminen siellä tapahtuu. Kaikki merkityspohjaisen tiedon esityksessä käytettävät muistirakenteet ovat hierarkkisia, eli semanttinen verkko voi yhtenä osanaan sisältää vaikkapa mielikuvia ja lineaarisia listoja.

Skeemojen ohella samaa asiaa kutsutaan usein myös *mentaalisiksi malleiksi* (mental models). Eri tutkijat määrittelevät mentaaliset mallit eri tavoin, mutta tämän työn yhteydessä mentaaliset mallit rinnastetaan skeemoihin. Ne voidaan määritellä esimerkiksi seuraavasti:

> Mentaaliset mallit ovat kognitiivisia artifakteja, eli mielen keksimiä rakennelmia, jotka edustavat, organisoivat ja uudelleenjärjestävät henkilön alakohtaista tietoa siten, että (havaittavissa olevan tai kuvitteellisen) maailman monimutkaisetkin ilmiöt tulevat vakuuttaviksi (Seel, 2001).

> Mentaaliset mallit ovat luonnollisesti kehittyviä malleja, joita ihminen muodostaa itsestään ja asioista ollessaan vuorovaikutuksessa ympäristönsä, muiden ihmisten tai teknisten artifaktien kanssa. Mallien avulla ihminen voi ennustaa ja selittää vuorovaikutukseen liittyviä ilmiöitä (Norman, 1983).

Mentaaliset mallit ovat usein puutteellisia, epävakaita ja epätarkkoja. Toimivakin mentaalinen malli voi sisältää ylimääräisiä elementtejä, joista ei kuitenkaan ole haittaa mallin soveltamisen kannalta. Usein ihmiset säilyttävät tarpeettomia elementtejä malleissaan, mikäli näin voidaan vähentää mentaalista rasitusta, eikä siitä aiheutuva fyysinen rasitus ole suuri verrattuna astetta monimutkaisemman mallin soveltamiseen (Norman, 1983). Mentaaliset mallit eivät voi muodostua täysin mielivaltaisesti, vaan ne ovat aina sidoksissa vallitsevaan ympäristöön. Tämä käy selkeästi ilmi erityisesti tietotekniikan yhteydessä, sillä toimimattomasta mentaalisesta mallista saadaan välitön palaute, jolloin mallia on muokattava (Ben-Ari, 2001).

2.4 Kognitiivisten prosessien käsittelytavat, lyhyt- ja pitkäkestoinen muisti ja oppiminen

Tarkkaavaisuuden, havaintojen tekemisen ja muistin eri osien yhteistoiminta luo perusedellytykset oppimiselle. Oppimisessa on kyse erittäin monimutkaisista prosesseista, ja

seuraavassa aihetta käydäänkin läpi ainoastaan korkeammalla abstraktiotasolla.

2.4.1 Kognitiivisten prosessien käsittelytavat

Kognitiivisten prosessien käsittelytavat voidaan jakaa kahteen päätyyppiin (Schneider & Shiffrin, 1977): *kontrolloituun* (controlled processing) ja *automaattiseen prosessointiin* (automatic processing). Kontrolloitua prosessointia tapahtuu jonkin uuden asian oppimisen yhteydessä. Tällainen prosessointi on hidasta ja se vaatii paljon keskittymistä. Kun jotakin uutta asiaa on harjoitellut tarpeeksi pitkään, sen prosessointi muuttuu aiempaa automaattisemmaksi. Tällöin kyseisen asian suorittaminen tarvitsee osakseen tarkkaavaisuutta enää hyvin vähän. Esimerkkinä tarkkaavaisuuden kohdistamisesta on pikkulapsen kävelemään oppiminen: aluksi pelkkä pystyssä pysyminen aiheuttaa miltei ylitsepääsemättömiä vaikeuksia ja vaatii osakseen suurta tarkkaavaisuutta, jolloin kyse on kontrolloidusta prosessoinnista. Harjoittelun myötä pystyssä pysyminen ja kävely muuttuu yhä helpommaksi ja luontevammaksi. Kyseisen taidon prosessointi muuttuu vähitellen automaattisemmaksi ja tarkkaavaisuutta voi kohdistaa enemmän muihin opittaviin asioihin.

2.4.2 Lyhytkestoinen muisti

Lyhytkestoisella muistilla (short-term memory) tarkoitetaan erityisessä aktiivisessa tilassa pidettävissä olevaa rajallista määrää tietoa. Tämä tieto on peräisin joko ympäristöstä tulleista havainnoista tai sitten se on aktivoitu pitkäkestoisesta muistista, josta kerrotaan lisää seuraavassa alakohdassa. Lyhytkestoisessa muistissa olevaa tietoa käsitellään mieltämisyksiköiden avulla. Yksittäinen mieltämisyksikkö voi olla melkein mitä tahansa: kirjain, sana, lause, numero, numerosarja, skeema tai joukko skeemoja. Kun tieto on aktivoitu lyhytkestoiseen muistiin, se ei kuitenkaan ole suoraan käytettävissä, vaan tiedonhakuun liittyy pieni viive. Tämän voi helposti havaita pitkien numerosarjojen kohdalla. Mitä kauempana numerosarjan alkukohdasta ollaan, sitä pitempi viive, ennen kuin tieto on saatavilla. Lyhytkestoisessa muistissa tapahtuneen käsittelyn jälkeen tieto voidaan tallentaa pitkäkestoiseen muistiin. Mitä enemmän jotakin asiaa opiskellaan, sitä voimakkaammin ja tehokkaammin kyseiseen asiaan liittyvä tietous voidaan tallentaa pitkäkestoiseen muistiin. Tutkimusten mukaan vaikuttaa siltä, että fyysisesti lyhytkestoinen muisti on pitkäkestoista muistia, mutta se on vain aktiivisessa

tilassa.

2.4.3 Pitkäkestoinen muisti

Pitkäkestoisessa muistissa (long-term memory) oleva tieto on normaalisti ei-aktiivisessa tilassa. Jotta tämä tieto saataisiin käyttöön, on se aktivoitava lyhytkestoisen muistin käyttöön. *Aktivaatio* leviää aktiivisesta muistin osasta eli lyhytkestoisesta muistista palautettavaan osaan assosiaatioyhteyksiä pitkin. Kun tieto on aktivoitu, se on käytettävissä lyhytkestoisessa muistissa. Tiedon aktivointia ja sen onnistumiseen vaikuttavia tekijöitä on tutkittu laajalti. Tiedon aktivoimiseen tarvittava aika on riippuvainen pitkäkestoiseen muistiin tallennetun muistirakenteen (semanttisen verkon, skeeman jne.) vahvuudesta. Toisin sanoen, mitä paremmin tieto on alunperin opiskeltu, sitä voimakkaammin ja tehokkaammin se on koodattuna pitkäkestoiseen muistiin ja sitä korkeampi on aktivaation taso. Aktivaation tasosta puolestaan riippuu, miten nopeasti pystytään suorittamaan kyseiseen tietoon liittyviä hakuja ja saadaan se käyttöön lyhytkestoisessa muistissa. Jonkin opittavan asian harjoittelu ei paranna siis pelkästään aktivaation onnistumisen todennäköisyyttä vaan nopeuttaa myös itse aktivaatiota.

Aktivaatio leviää itsestään aktiivisena olevan tiedon ympäristöön, mutta sen leviämisellä on kuitenkin rajoituksia. Jos ajatellaan aktivaation leviämisen tapahtuvan semanttisessa verkossa, voidaan ilmiö selittää seuraavalla tavalla. Kun jostakin verkon solmusta lähtee useita eri haaroja muihin solmuihin, aktivaatio ja sen leviämisen voimakkuus jakautuu monien eri haarojen ja solmujen kesken. Tämän seurauksena aktivaatio leviää useampaan solmuun, mutta rajallisena resurssina sitä riittää vähemmän kutakin haaraa kohden. Mitä kauemmas aktivaation on levittävä, sitä pitempään asian mieleen palauttaminen kestää. Jos aktivaation pitää levitä liian kauas, saattaa lyhytkestoisen muistin kapasiteetti ylittyä tai muistamiseen vaadittavan aktivaation taso jäädä saavuttamatta ja muistihaku epäonnistua. Muistihaun onnistumiseen vaikuttavat myös samaan aikaan meneillään olevat muut kilpailevat haut, jotka saattavat vaatia osakseen niin paljon aktivaatiota, että jokin muu muistihaku epäonnistuu.

Erilaiset muistirakenteet eivät ole ikuisia. Ne muuttuvat koko ajan, ja ne myös rappeutuvat ajan myötä. Rappeutumisen nopeuteen vaikuttaa muistirakenteen vahvuuden lähtötaso sekä tiedon mahdollinen virkistys, mikäli sitä käytetään. Muistirakenteen heikkeneminen näkyy reaktioaikojen pitenemisenä muistihaun yhteydessä sekä virheiden määrän kasvuna. Muistissa on myös paljon tietoa, joka ei ole aktiivisesti palautettavis-

sa.

2.4.4 Oppiminen

Olennaiseksi katsottu ulkopuolelta tuleva informaatio päätyy sensoristen muistien ja aistimusten tunnistamismekanismien välityksellä tietoisesti käsiteltäväksi lyhytkestoiseen muistiin. Tähän informaatioon voidaan kohdistaa tarkkaavaisuutta lyhytkestoisessa muistissa ja tekemällä hakuja pitkäkestoiseen muistiin sitä voidaan yhdistellä pitkäkestoisessa muistissa olevaan tietämykseen eri tavoin. Kun asiaa on käsitelty tarpeelliseksi katsottu määrä lyhytkestoisessa muistissa, se voidaan jälleen tallentaa pitkäkestoiseen muistiin. Tämä uuden ja vanhan tiedon yhdisteleminen ja muokkaaminen mahdollistaa skeemojen (tai mentaalisten mallien) kehittymisen eli asioiden oppimisen.

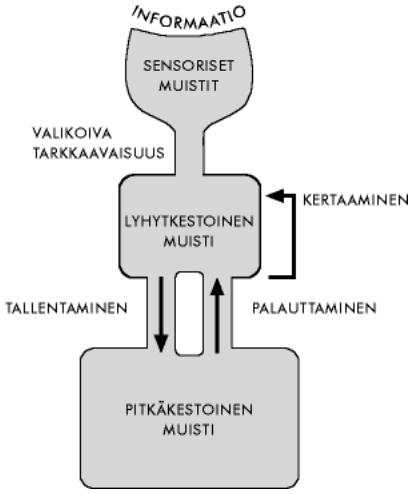
2.4.5 Yhteenveto muistin toiminnasta ja oppimisesta

Muistin toiminta voidaan jakaa kolmeen eri muistijärjestelmään: aistien toimintaan liittyviin sensorisiin muisteihin (visuaalinen sensorinen muisti näköaistimuksille ja ekoinen muisti kuuloaistimuksille), lyhytkestoiseen muistiin ja pitkäkestoiseen muistiin. Muistijärjestelmät eroavat toisistaan niin kestoltaan, kapasiteetiltaan kuin unohtamismekanismeiltaan. Olennainen käsite muistista puhuttaessa on myös tarkkaavaisuus, jonka valikoivaan käyttöön muistin toiminta pitkälti perustuu. Taulukossa 2 on lyhyesti merkitty eri muistijärjestelmät, niiden kestot, kapasiteetit ja unohtamismekanismit. Muistien kestot ja kapasiteetit ovat riippuvaisia monista eri asioista, joten annetut lukemat ovat lähinnä suuntaa-antavia.

Kuvassa 2 on esitetty yleisellä tasolla ihmismuistin toiminta kaaviomuodossa. Ylhäällä näkyy ympäristöstä koko ajan tulevaa informaatiota, jota käsitellään tiedostamatta alustavasti sensorisissa muisteissa. Tämän jälkeen informaatiosta olennaisimmaksi katsottuun osaan kiinnitetään tietoisesti tarkkaavaisuutta, ja prosessoitu tieto päätyy lyhytkestoisen muistin käytettäväksi. Mikäli on tarpeen, tätä tietoa voidaan aktiivisesti toistaa mielessä, jolloin se säilyy lyhytkestoisessa muistissa (esim. puhelinnumeron toistaminen päässä, kunnes numero on saatu valittua). Vaihtoehtoisesti se voidaan myös prosessoida edelleen itselle merkitykselliseen muotoon ja tallentaa pitkäkestoiseen muistiin. Vastaavasti pitkäkestoisesta muistista tehdään koko ajan hakuja, jotta siellä olevaa tietoa voidaan aktiivisesti käsitellä lyhytkestoisessa muistissa. Tiedon tie-

Taulukko 2: Yhteenveto muisteista.

	visuaalinen sensorinen muisti	**ekoinen muisti**	**lyhytkestoinen muisti**	**pitkäkestoinen muisti**
muistin kesto	alle 1 sekunti	muutama sekunti	ilman toistoa alle minuutti, toiston kanssa useita minuutteja	vuosia
muistin kapasiteetti	-	-	5-9 mieltämisyksikköä	rajoittamaton
unohtamismekanismi	vaimeneminen	vaimeneminen	syrjäyttäminen uudella informaatiolla tai tarkkaavaisuuden siirtäminen	muistijäljen katoaminen tai muistivihjeiden puuttuminen



Kuva 2: Muistin toiminta yleisellä tasolla.

toisella tasolla tapahtuva prosessointi on mahdollista tarkkaavaisuuden ansiosta.

2.5 Tiedon jaottelu ja ongelmanratkaisu

Oppimisen eri vaiheissa tiedot opittavasta asiasta ja kyky soveltaa tätä tietoutta muuttuvat. Tieto muuttuu vähitellen deklaratiivisesta tiedosta proseduraaliseksi ja opittavan

asian kannalta olennaisia tietoja ja taitoja osataan käyttää tehokkaammin hyväksi. Tämä voidaan selittää pitkäkestoisten tietämysrakenteiden eli skeemojen tai mentaalisten mallien kehittymisellä harjoittelun myötä. Deklaratiivisen ja proseduraalisen tiedon välisen ero lisäksi tässä yhteydessä esitellään proseduraalisen tiedon käyttötapoja, eli menetelmiä joita ihmiset käyttävät ratkaistessaan erilaisia ongelmia.

2.5.1 Deklaratiivinen ja proseduraalinen tieto

Kognitiivisessa psykologiassa tieto jaetaan usein kahteen kategoriaan: deklaratiiviseen ja proseduraaliseen tietoon. *Deklaratiivinen tieto* (declarative knowledge) on pitkäkestoisessa muistissa olevaa tietoa asioista ja tosiasioista, *proseduraalinen tieto* (procedural knowledge) puolestaan on tietoa siitä, kuinka suorittaa erilaisia kognitiivisia toimenpiteitä ja ratkaista ongelmia. Edellisissä kappaleissa on kerrottu, miten ihminen vastaanottaa informaatiota ympäröivästä maailmasta, prosessoi tämän informaation ja lopuksi tallentaa sen tietyllä tavalla aivoihin. Aluksi tämä tieto on tyypiltään ainoastaan deklaratiivista. Esimerkiksi noviisiohjelmoija tietää periaatteen tasolla, että kokonaislukuja voi tallentaa tietyn tyyppisiin muuttujiin, mutta ei osaa soveltaa tätä tietoa käytännössä. Harjoittelun myötä tietoa opitaan käyttämään yhä erilaisemmissa tilanteissa, jolloin se muuttuu deklaratiivisesta proseduraaliseen muotoon.

2.5.2 Ongelmanratkaisu ja ongelmanratkaisuheuristiikat

Kaikkien ongelmien ratkaisemiseen kuuluu tiettyjä yhteisiä piirteitä. On olemassa jokin alkutila, josta pyritään kohti tavoitetta. Ongelman ratkaisemiseksi se täytyy jakaa osatavoitteisiin, jotka suorittamalla saavutetaan alkuperäisen ongelman ratkaisu. Ongelman ratkaisemiseksi on olemassa operaattoreita. *Operaattorit* ovat toimenpiteitä, jotka suorittamalla voidaan siirtyä ongelmanratkaisussa nykytilasta johonkin toiseen tilaan. Toisin sanoen ongelmanratkaisu kokonaisuudessaan muodostuu sarjasta operaattoreita, jotka suoritetaan tietyssä järjestyksessä. Ongelmanratkaisun yhteydessä puhutaan usein myös *ongelma-avaruudesta* (problem space) ja siinä liikkumisesta. Ongelmanratkaisun kaikkiin vaiheisiin liittyy aina jokin tietty tila ja eri tilanteissa käytettävissä olevat operaattorit. Ongelman ratkaisemiseksi pitää siis löytää sarja eri tilojen ja niiden yhteydessä käytettävissä olevien operaattoreiden yhdistelmiä, jotka yhdessä mahdollistavat ongelman ratkaisemisen. Seuraavassa kerrotaan tunnettuja mene-

telmiä eli ongelmanratkaisuheuristiikkoja, joita ihmiset käyttävät ongelmia ratkaistessaan. Heuristiikoilla on yhteisenä piirteenä se, että niiden käyttäminen ei takaa haluttuun tavoitteeseen pääsemistä. Kulloinkin valittu ongelmanratkaisuheuristiikka riippuu monista eri seikoista. Ongelmanratkaisijan aiempi kokemus ja senhetkinen vireystila tai tapa, jolla ongelma esitetään, saattavat vaikuttaa heuristiikan valintaan. Mahdollista on myös useamman eri heuristiikan yhdistelmä, jos ongelma ei tunnu tietyllä tavalla ratkeavan.

Erotuksen pienentämisheuristiikka

Erotuksen pienentämisheuristiikassa (difference-reduction method) ongelman ratkaisu jaetaan alitavoitteisiin, jotka suorittamalla päästään askel kerrallaan nykyisestä tilasta lähemmäs kohti tavoitetilaa. Näin tilojen välinen erotus pienenee, ja lopulta tavoitetila saavutetaan. Usein käytetty yleisen tason esimerkki tällaisesta tilanteesta on tehtävä, jossa kivi pitää saada siirrettyä pisteestä A pisteeseen B. Erotuksen pienentämisheuristiikkaa käyttämällä toivottuun lopputilaan päästään yksinkertaisesti vierittämällä kivi vähitellen (suorittamalla operaattori) pisteestä A pisteeseen B. Erotuksen pienentämisheuristiikkaa käytettäessä saatetaan joutua ongelmiin, mikäli ongelmanratkaisu ei ole suoraviivainen, eli nykytilan ja lopputilan välistä erotusta joudutaan ratkaisun aikana väliaikaisesti kasvattamaan.

Tavoite-keino-heuristiikka

Tavoite-keino-heuristiikka (means-ends-analysis) voidaan nähdä erotuksen pienentämisheuristiikan kehittyneempänä versiona. Kuten erotuksen pienentämisheuristiikassakin, myös tavoite-keino-heuristiikassa pyritään pienentämään nykytilan ja halutun lopputilan välistä erotusta. Varsinainen ero menetelmien välillä on siinä, että jos tavoite-keino-heuristiikassa havaitaan, että jokin ratkaisun kannalta olennainen operaattori ei ole sillä hetkellä käytettävissä, pyritään luomaan tilanne, jolla operaattori saadaan käyttöön ja ongelma ratkaistua. Ideana on siis suorittaa operaattoreita, jotka ovat järkeviä kokonaisratkaisun kannalta, vaikka juuri kyseinen operaattori ei olisikaan erotuksen pienentämisheuristiikan mukainen. Esimerkkinä tavoite-keino-heuristiikan käyttämisestä voidaan käyttää kuuluisaa Hanoin tornit -ongelmaa, jossa on kolme tolppaa, eli tolpat numero yksi, kaksi ja kolme. Tolpassa yksi on erikokoisia levyjä, jotka

on aseteltu siten, että isoin levyistä on tolpassa alimmaisena ja pienin päällimmäisenä. Tavoitteena on siirtää levyt tolppaan kolme yksi kerrallaan siten, että isompi levy ei ole koskaan pienemmän levyn päällä. Tässä ongelmassa operaattorit ovat levyjen siirtoja jostakin tolpasta johonkin toiseen tolppaan. Jos tolpassa numero yksi on vaikkapa kolme levyä, ei pohjalla olevaa isointa levyä voida siirtää ensimmäiseksi, ts. kyseinen operaattori ei ole käytettävissä. Sen sijaan on aluksi siirrettävä päällä olevia pienempiä levyjä sopivasti, jotta ratkaisun kannalta tarpeellinen operaattori eli isoimman levyn siirto tolpan numero yksi pohjalta tolpan numero kolme pohjalle voitaisiin suorittaa.

Takautuva-työskentely-heuristiikka ja analoginen ongelmanratkaisu

Takautuva-työskentely-heuristiikka (working backward) on ongelmanratkaisumenetelmä, jossa lähdetään halutusta lopputilanteesta eli ratkaistusta ongelmasta kohti alkutilannetta, jossa sillä hetkellä ollaan. Tietojenkäsittelyn yhteydessä puhutaan useimmiten tästä menetelmästä nimellä *alhaalta-ylös* (bottom-up). *Analoginen ongelmanratkaisu* (problem solving by analogy) perustuu aiemmin kohdatun ongelman ja nykyisen ongelman vastaavuuteen. Kun on havaittu, että tietämys tämän tyyppisen ongelman ratkaisuun on jo olemassa, voidaan tätä jo aivoissa olevaa tietämystä soveltaa suoraan myös nykyiseen ongelman ratkaisemiseksi.

2.6 Taidon oppimisen vaiheet: noviisista ekspertiksi

Andersonin (1985) mukaan Anderson (1983) ja Fitts & Posner (1967) ovat havainneet, että minkä tahansa uuden taidon oppimisprosessi voidaan jakaa kolmeen eri vaiheeseen: kognitiiviseen, assosiatiiviseen ja autonomiseen. On helppo havaita taidon oppimisen vaiheiden yhtäläisyydet myös eksperttiyden eri vaiheisiin: noviisiin, edistyneeseen ja eksperttiin. Seuraavassa on selitetty aluksi yleisellä tasolla, mitä nämä taidon oppimisen vaiheet ovat sekä vaiheiden konkreettinen merkitys eri näkökulmista tarkasteltuna, kun opittavana taitona on ohjelmointi ja erityisesti ohjelmien kirjoittaminen. Opetuksen kannalta suuri kysymys on, että miten voidaan helpottaa oppijan taakkaa kahden ensimmäisen vaiheen aikana.

2.6.1 Kognitiivinen vaihe

Kognitiivisessa vaiheessa opitaan taitoon liittyvät peruskäsitteet. Ohjelmoinnin yhteydessä tämä tarkoittaa monia eri asioita: opitaan mm. muuttujan, alkeistietotyyppien, ehdollisuuden, silmukkarakenteiden ja algoritmien käsitteet. Samalla opitaan myös ohjelmointikielen syntaksi, eli kielioppi, jonka sääntöjä noudattamalla voidaan päätyä haluttuun lopputulokseen, ts. ohjelmointiongelmien ratkaisemiseen. Tässä vaiheessa oppijan käsitys itse varsinaisesta taidosta on hyvin rajoittunut, koska opittavaan taitoon liittyvät käsitteet ovat vielä deklaratiivisessa muodossa. Oppijalla on alustava ymmärrys siitä, mitä rakennuspalikoita hänellä on käytettävissä, mutta kyky käyttää niitä vielä puuttuu. Käytännön tasolla tämä tarkoittaa, että jopa lyhyen ”Hello world!!”-tyyppisen ohjelman tekeminen vaatii huomattavia ponnistuksia.

2.6.2 Assosiatiivinen vaihe

Assosiatiivisen vaiheen yhteydessä tapahtuvat merkittävimmät asiat oppimisen suhteen. Kognitiivisen vaiheen jäljiltä jääneet virheet ja epäselvyydet poistuvat vähitellen, ja taito muuttuu yhä automaattisemmaksi. Ohjelmoinnin oppimisen kannalta tämä merkitsee esimerkiksi sitä, että ohjelmoijan ei enää tarvitse aina alkuehtoista toistosilmukkaa käyttäessään miettiä aluksi siinä tarvittavaa syntaksia, vaan hän voi keskittyä enemmän varsinaisen ongelman ratkaisemiseen. Yhteydet ja suhteet taidossa tarvittavien elementtien välillä paranevat ja selkiytyvät, ja taidon harjoittaminen automatisoituu. Erilaisia ohjelmointitehtäviä ratkaistessaan oppijan tiedot kyseisestä taidosta muuttuvat deklaratiivisesta tiedosta yhä enemmän proseduraaliseksi tiedoksi. Tämä ei tarkoita, että deklaratiivinen tieto häviäisi. Vaikka käytettävä taito hallittaisiin proseduraalisella tasolla, on oppija edelleen tietoinen deklaratiivisessa muodossa olevasta tiedosta.

2.6.3 Autonominen vaihe

Autonomisessa vaiheessa taidon harjoittaminen muuttuu aiempaa nopeammaksi, automaattisemmaksi ja virheettömämmäksi ja sen harjoittamisessa tarvitaan koko ajan vähemmän resursseja. Ohjelmointitaidossa tämä ilmenee siten, että oppija osaa ratkaista yhä vaativampia ongelmia aiempaa nopeammin ja tehokkaammin, ja yksinkertaisim-

mat ongelmat pystytään ratkaisemaan miltei automaatiotasolla käyttämällä apuna jo pitkäkestoiseen muistiin tallentuneita valmiita ratkaisumalleja. Assosiatiivisen ja autonomisen vaiheen välinen raja taidon oppimisessa on usein hämärä, eikä usein voida sanoa, milloin oppija on siirtynyt vaiheesta toiseen.

2.7 Noviisin ja ekspertin eroavaisuuksia

Eksperttiyden syntyyn vaaditaan paljon aikaa, mutta se ei yksinään takaa korkean ammattitaidon kehittymistä, vaan aina tarvitaan määrätietoista ja sitkeää harjoittelua ja aitoa kiinnostusta opeteltavaan asiaan. Tästä seuraa se, että eksperttiys on aina rajoittunut tietylle erikoisalalle, eikä kaikkien mahdollisten taitojen eksperttejä ole olemassa. On lohdullista tietää, että eksperttikin voi käyttäytyä omalla erikoisalallaan joskus noviisin tavoin, mikäli törmää asiaan, josta ei omaa aiempia tietoja. Eksperttikin oppii koko ajan uutta. Seuraavassa on kootusti Glaserin & Chin (1988) noviisin ja ekspertin välisistä eroavaisuuksista tekemiä havaintoja, jotka pätevät yleisellä tasolla erikoisalasta riippumatta.

1. **Ekspertit ovat poikkeuksellisen hyviä useimmiten vain omalla alallaan.** Kun jonkin alan ekspertit laitetaan tekemään jonkin muun alan työtä, on heidän suoritustasonsa noviisin tasolla. Tämä viittaa siihen, että eksperttiydessä ei ole niinkään kyse poikkeuksellisesta älykkyydestä tai synnynnäisestä kyvystä, vaan kertyneestä tietyn erikoisalan tietämyksestä ja kokemuksesta, jota ekspertti osaa käyttää taidokkaasti hyödykseen.

2. **Ekspertit havaitsevat ja oivaltavat erinomaisesti suuria merkityksellisiä kokonaisuuksia omalla erikoisalallaan.** Tämä kyky ei tarkoita, että eksperteillä on poikkeuksellisen kehittynyt havainnointikyky verrattuna muihin. Kyse on siitä, että eksperttien oman erikoisalan tietämyspohja on jäsentynyt sellaiseen muotoon, josta se on paremmin käytettävissä kuin noviiseilla ja olennaisen havaitseminen ongelmatilanteessa on täten helpompaa. Noviiseilla on tapana juuttua ongelmassa johonkin häiritsevään yksityiskohtaan näkemättä kuitenkaan riittävän selkeästi kokonaisuutta.

3. **Ekspertit ovat nopeampia suorittamaan erikoisalansa tehtäviä ja he tekevät vähemmän virheitä kuin noviisit.** Tähän on olemassa ainakin kaksi mahdollista selitystä riippuen myös osaltaan opittavan taidon vaikeusasteesta: kova

harjoittelu sekä valmiit ongelmien ratkaisumallit, joita eksperteillä on, mutta jotka puuttuvat noviiseilta. Esimerkiksi todella nopeat konekirjoittajat ovat harjoitelleet tuntikaupalla saavuttaakseen taidossaan rutiinin ja korkean taidon automaatioasteen. Täten eksperttien sormet liikkuvat fyysisestikin nopeammin kuin noviiseilla, ja kirjoittaminen onnistuu nopeammin, vaikka käytetään vähemmän resursseja kuin noviisit käyttävät. Monimutkaisemmissa taidoissa on kyse siitä, että eksperttien ei tarvitse ratkaista ongelmaa aloittaen puhtaalta pöydältä, kuten noviisit tekevät. Ekspertit pyrkivät ongelmanratkaisussa pääsemään tilanteeseen, jonka ratkaisun he jo tietävät. Kun tähän tilaan on päästy, voidaan käyttää apuna tunnettuja operaattoreita.

4. **Eksperteillä on parempi lyhytkestoinen ja pitkäkestoinen muisti kuin noviiseilla.** Samat lyhytkestoisen muistin rajoitukset koskevat niin eksperttejä kuin noviisejakin, ja mieltämisyksiköitä voi olla käytössä rajallinen määrä. Rajoitusta on kuitenkin mahdollista kiertää: eksperttien omaa erikoisalaa koskevat mieltämisyksiköt ovat suuremmat kuin noviiseilla, ja samoin korkea automaatioaste oman erikoisalan taitojen suorittamisessa vapauttaa enemmän resursseja muuhun käyttöön.

5. **Ekspertit näkevät erikoisalansa ongelman syvemmällä tasolla kuin noviisit, jotka näyttävät käsittelevän ongelmaa enemmän pintatasolla.** Sekä noviisit että ekspertit pystyvät jakamaan ongelmat kategorioihin tietyllä perusteella. Eksperteillä nuo perusteet liittyvät enemmän ongelman syvällisempään merkitykseen ja ratkaisuperiaatteisiin, kun taas noviisit ryhmittelevät ongelmat enimmäkseen niiden pintarakenteiden perusteella.

6. **Ekspertit käyttävät enemmän aikaa ongelman kvalitatiiviseen analysointiin kuin noviisit.** Ongelmanratkaisun alkuvaiheessa ekspertit pyrkivät yleensä ymmärtämään ongelman kunnolla ja ryhtyvät vasta sitten laatimaan ratkaisua. Noviisit sen sijaan pyrkivät useammin umpimähkään kokeilemaan eri tapoja. Tällä pitempikestoisella analysoinnilla on selkeitä etuja etenkin, jos ongelmaa ei ole selkeästi määritelty. Analysoinnin avulla ekspertit voivat pienentää mahdollisten ratkaisujen määrää ja keskittyä ongelma-avaruudessa alueisiin, jotka todennäköisemmin auttavat ratkaisun löytämisessä.

7. **Ekspertit tarkkailevat omaa toimintaansa tarkemmin ja analyyttisemmin kuin noviisit.** Ekspertit näyttävät olevan noviiseja paremmin selvillä siitä, mil-

loin he ovat tehneet virheen ja mistä se johtui. Tämä kyky näyttäisi liittyvän eksperttien laajaan tietämykseen ja kokemukseen omasta erikoisalastaan.

Eksperttiyteen ja sen syntyyn liittyy selkeästi tiettyjä lainalaisuuksia ja erityispiirteitä. Eksperttiys jollakin alalla on tietoisen harjoittelun tulosta, mistä seuraa luonnollisesti, että eksperttiys on erikoisalakohtaista. Harjoittelun tuloksena ekspertit pystyvät käsittelemään oman alansa asioita nopeasti, tehokkaasti ja tekemällä vain vähän virheitä. Koska eksperttien mielessä olevat skeemat (tai mentaaliset mallit) heidän erikoisalaltaan ovat kehittyneempiä kuin noviiseilla, pystyvät ekspertit käsittelemään suuria kokonaisuuksia syvällisellä tasolla samalla tietoisena omasta toiminnastaan ja sen tavoitteista, toisin kuin noviisit.

Tässä luvussa tarkasteltiin oppimiseen liittyviä asioita ja käsitteistöä yleisellä tasolla, ja selvitettiin, kuinka nykytietämyksen mukaan asioiden syvällinen oppiminen etenee aktiivisesti skeemoja (tai mentaalisia malleja) hyödyntämällä, testaamalla, muokkaamalla ja rakentamalla. Näin noviisista voi kehittyä vähitellen ekspertti. Ennen kuin tietämys saadaan tallennettua pysyvämpiin muistirakenteisiin, tarvitaan kuitenkin monia eri välivaiheita, tarkkaavaisuuden käyttöä, ongelmanratkaisua, tiedon siirtämistä edestakaisin muistissa ja monia muita prosesseja, joita tässä yhteydessä voitiin tarkastella vain erittäin lyhyesti. Tämän työn yhteydessä tuo tarkkuus on kuitenkin riittävä, jotta voidaan edetä askel eteenpäin varsinaista aihetta, eli ohjelmoinnin oppimista ja opettamista kohti.

3 Ohjelmoinnin eksperttiys, ohjelmoinnin psykologian tutkimuksia ja ohjelmoinnin alkeiden opettaminen

Luvun kaksi alussa määriteltiin noviisi ja ekspertti yleisellä tasolla, ja sen lopussa määriteltiin noviisin ja ekspertin yleisiä eroavaisuuksia. Tämän luvun aluksi keskitytään pohdiskelemaan, mitä asioita voisi kuulua ohjelmoinnin eksperttiyteen, mitä noviisi ja ekspertti tarkoittaa ohjelmoinnin yhteydessä, ja pyritään löytämään näille termeille määritelmät. Tämän jälkeen esitellään ohjelmoinnin psykologiaan liittyviä tutkimuksia, joissa on löydetty tiettyjä lainalaisuuksia ohjelmointiin ja sen oppimiseen liittyen. Lisäksi pohditaan, mistä johtuvat noviisien keskeisimmät ohjelmointiin liittyvä ongelmat. Lopuksi tarkastellaan, miten ohjelmoinnin alkeita on yritetty opettaa ja pohditaan lyhyesti, miten nämä eri tavat sopivat yhteen ohjelmoinnin osa-alueisiin liittyvään tutkimustietoon.

3.1 Ohjelmoinnin eksperttiyden osa-alueet ja noviisiohjelmoijan ja eksperttiohjelmoijan määritelmät

On helppo määritellä noviisi ja ekspertti yleisellä tasolla, mutta ohjelmoinnin yhteydessä yksiselitteinen määritteleminen on paljon vaikeampaa. Tämä johtuu siitä, että ohjelmointiin liittyvä tietämys ja taidot jakautuvat niin moniin eri osa-alueisiin. Ohjelmoinnin eksperttiyteen voidaan ajatella kuuluvan ainakin jonkin ohjelmointiparadigman ja -kielen hallinnan eri tilanteissa, työskentelyvälineiden ja -ympäristön tuntemuksen ja käyttötaidon sekä johonkin erikoisalaan liittyvän tietämyksen. Nämä osa-alueet käydään lyhyesti läpi, jotta selviäisi, miten laajaa osaamista ohjelmointi vaatii. Tämän jälkeen määritellään, mitä noviisi ja ekspertti ohjelmoinnissa tarkoittavat tämän työn yhteydessä.

3.1.1 Ohjelmointiparadigma ja -kieli ja sen hallinta eri tilanteissa

Yleensä ohjelmointikielet jaetaan *imperatiiviseen* eli *proseduraaliseen* (C, Pascal), *oliokeskeiseen* (Java, C#, SmallTalk), *funktionaaliseen* (Lisp) ja *loogiseen* paradigmaan (Prolog). Tällä hetkellä laajimmin ovat käytössä erityisesti proseduraalisen ja oliokeskeisen paradigman kielet, mutta jokaisella paradigmalla ja sen edustamilla kie-

lillä on vannoutuneet puolestapuhujansa ja käyttötarkoituksensa. Kielen hallintaan eri tilanteissa puolestaan liittyy sen käyttämiseen ohjelmien kirjoittamisessa, kykyyn ymmärtää lähdekoodia, etsiä siitä virheitä sekä tehdä siihen korjauksia ja muutoksia.

3.1.2 Työskentelyvälineiden ja -ympäristön tuntemus ja käyttötaito

Ohjelmointi tapahtuu aina tiettyä ympäristöä varten (Windows, Linux, OpenBSD, Symbian, pesukoneeseen liittyvä sulautettu järjestelmä tms.) ja tiettyjä työvälineitä (Vi, Emacs, gcc, VC++, CVS, ...) ja niiden versioita käyttäen. Työskentelyvälineiden hyvä hallinta vaatii tietämystä, tapahtuu ohjelmointi sitten mitä ympäristöä varten ja työskentelyvälineitä käyttäen tahansa.

3.1.3 Erikoisalaan liittyvä tietämys

Nykyaikana ohjelmistoihin törmää miltei kaikilla elämän osa-alueilla. Ohjelmoijat eivät elä tyhjiössä, vaan käytössä olevilla ohjelmistolla on aina jokin tarkoitus. Tämä näkyy siten, että ohjelmoijilla tulee olla omaan erikoisalaan liittyvää tietämystä puhtaan ohjelmointitaidon lisäksi: kaupallisen alan ohjelmistojen suunnittelussa ja toteuttamisessa vaaditaan toisenlaista tietämystä kuin vaikkapa lennonhallinnan ohjausjärjestelmien, tietokonepelien tai Linux-kernelin ohjelmoinnissa. Itse asiassa Linux-kernelin ohjelmoijatkin ovat vielä erikoistuneet omaan, pienempään kernelin osaan, koska kerneli koostuu useista miljoonista koodiriveistä.

3.1.4 Noviisiohjelmoijan ja eksperttiohjelmoijan määritelmät

Mitä taitoja ohjelmoinnin eksperttiyteen kuuluu? Yleensä ottaen korostetaan vain edellä luetelluista asioista ensin mainittua, eli jonkin ohjelmointiparadigman ja -kielen hallintaa, vaikka se yksinään ei riitäkään. Onko yleensäkään mielekästä puhua ohjelmoinnin eksperttiydestä, koska tutkittava alue on näin laaja ja vaihtuvia tekijöitä on niin paljon? Yhtenäistä ja yksiselitteistä määritelmää noviisista ja ekspertistä ei olekaan olemassa. Tämä on nähtävissä myös ohjelmoinnin empiiriseen tutkimiseen liittyvissä tieteellisissä tutkimuksissa, joissa käytetään termejä noviisi ja ekspertti hyvin erilaisissa merkityksissä. Noviisi voi tarkoittaa esimerkiksi henkilöä, joka ei ole koskaan ohjelmoinut tai sitten vaikkapa henkilöä, joka on käynyt läpi yhden tai useamman ohjel-

mointikurssin, mutta ei omaa vielä useiden vuosien kokemusta. Myös ekspertin määrittelyssä on suurta vaihtelua, mutta useimmiten toistuu maininta usean vuoden kokemuksesta ohjelmoinnista, samoin saatetaan mainita jokin erikoisala.

Vaikka ohjelmoinnin eksperttiyden yksiselitteinen määritteleminen on vaikeata, se ei tarkoita, etteikö eksperttejä ohjelmoinnin alueella olisi. Määrittelyvaikeuksista johtuen pidetään noviisiohjelmoijan ja eksperttiohjelmoijan määritelmät tämän työn yhteydessä tarkoituksella laveana. Laajennetaan vain luvun kaksi alussa olleet noviisin ja ekspertin yleiset määritelmän koskemaan ohjelmointia seuraavasti: *noviisi* on henkilö, jonka tiedot ja taidot ohjelmoinnista eivät ole vielä täysin kehittyneet, tai henkilö, joka on vasta opettelemassa ohjelmointia eikä omaa paljon ohjelmointikokemusta. *Ekspertti* on ohjelmoinnin ammattilainen, joka on osaava, taitava ja omaa pitkän työkokemuksen ohjelmointiin liittyen.

Seuraavassa tarkastelussa jätetään työskentelyvälineet ja erikoisalaan liittyvän tietämyksen tietoisesti käsittelemättä, ja keskitytään ainoastaan ohjelmoijan ydinosaamiseen, eli ohjelmointikiel(t)en hallitsemiseen. Läpi käydään tutkimustietoa ohjelmien suunnittelemiseen ja kirjoittamiseen sekä lähdekoodin ymmärtämiseen liittyen. Osa seuraavaksi esiteltävistä tutkimuksista ja niihin liittyvistä malleista liittyy suoranaisesti vain johonkin ohjelmoinnin osa-alueeseen, kun taas osa on sovellettavissa ainakin jollakin tapaa kaikkiin ohjelmoinnin osa-alueisiin.

3.2 Ohjelmien suunnitteleminen ja kirjoittaminen

Ohjelmoijan kyky muodostaa ja esittää ohjelmien osia merkityksellisinä kokonaisuuksina on erittäin tärkeä osa ohjelmoijan asiantuntemusta (Davies, 1993). Mitä nuo merkitykselliset kokonaisuudet ovat? Tavoite/menetelmä-malli sekä fokaalirivit ja menetelmien luominen ovat tunnettuja ohjelmoinnin psykologiaan liittyviä teorioita, joissa asiaa on tarkasteltu. Lisäksi perehdytään parsing–gnisrap-malliin, jossa ohjelmointia tarkastellaan käytettävissä olevien rajallisten resurssien näkökulmasta.

3.2.1 Tavoite/menetelmä-malli

Noviisien vaikeudet yksinkertaisimpienkin ohjelmien kirjoittamisessa eivät suurimmaksi osaksi johdu siitä, etteivätkö he tuntisi käyttämänsä ohjelmointikielen syntaksia.

Ongelmat johtuvat sen sijaan enimmäkseen siitä, että noviisit eivät osaa yhdistellä tietämiään rakenteita järkeviksi kokonaisuuksiksi. Nämä havainnot käyvät ilmi Spohrer & al. (1985) ja Spohrer & Solowayn (1986) tutkimuksista, joissa he analysoivat noviisien kirjoittamia Pascal-kielisiä ohjelmia. He sovelsivat ohjelmien ja niissä esiintyvien virheiden luokittelussa tavoite/menetelmä-mallia.

Tavoite/menetelmä-mallin pohjana on ajatus, että eksperteillä on hallussaan hiljaista tietämystä, joka on kertynyt heidän aikaisemmin ratkaisemista ongelmista. Tämä tietämys on korkeammalla tasolla kuin pelkkä ohjelmointikielen syntaksi tai semanttinen rakenne, ja ekspertit hyödyntävät sitä uusia ohjelmointiongelmia ratkaistessaan. Soloway & al.(1982) nimittävät tätä tietämystä menetelmiksi (plans). *Menetelmät* voivat heidän mukaansa olla joko ohjelmointikielestä riippumattomia toimintatapoja tai strategioita, jotka abstrahoivat ongelman ratkaisuprosessin avainkohdat, tai sitten ohjelmointikielestä riippuvia toteutusmenetelmiä. Menetelmät ovat palautettavissa, ja niitä voidaan soveltaa uusien, samankaltaisten ohjelmointiongelmien ratkaisemiseksi. Menetelmillä on rikas sisäinen rakenne, ja ne voidaan ymmärtää tietämysrakenteina, jotka ovat yhteydessä toisiinsa erilaisten suhteiden kautta. Kognitiivisen psykologian näkökulmasta ajateltuna menetelmissä on kyse skeemoista tai mentaalisista malleista, jotka noviisilla ovat vähemmän ja ekspertillä taas enemmän kehittyneitä.

Ohjelmointiongelman kuvauksesta on useimmiten erotettavissa *tärkeimmät tavoitteet* (main goals), jotka ongelman ratkaisemiseksi pitää saavuttaa. Tavoitteet voidaan saavuttaa edellä mainittuja menetelmiä käyttämällä eli käytännössä jonkin menetelmän hyödyntäminen ilmenee ohjelmassa lähdekoodin pätkinä. Tavoitteet ja menetelmät puolestaan jakautuvat pienempiin alitavoitteisiin ja alimenetelmiin. Valmis, toimiva ohjelma muodostuu siis joukosta menetelmiä, jotka on nivottu oikealla tavalla yhteen. Tavoite/menetelmä-mallia käyttämällä ohjelmien tarkastelun abstraktiotaso nousee korkeammaksi, eikä enää liikuta pelkästään ohjelmointikielen syntaksitasolla. On huomattava, että saman tavoitteen ratkaisemiseksi voi olla useita erilaisia menetelmiä (Spohrer & al., 1985).

Noviiseilla menetelmätietämys on vielä vähäisestä ohjelmointikokemuksesta johtuen puutteellista. Tämä johtaa siihen, että he yrittävät käyttää ohjelmointiongelman ratkaisemisessa apuna aikaisemmin elämänsä aikana oppimaansa tietoa (preprogramming knowledge), mikä saattaa johtaa monenlaisiin ongelmiin (Bonar & Soloway, 1985).

3.2.2 Fokaalirivit ja menetelmien luominen

Ristin (1989) tekemät havainnot täydentävät tavoite/menetelmä-mallia ja pyrkivät selittämään, kuinka ohjelmoijat ongelmia ratkaistessaan käyttävät hyväksi valmiita skeemoja, yhdistelevät niitä ja luovat tarpeen mukaan uusia. Ristin mukaan ekspertit eivät käytä ohjelmoidessaan pelkästään *osittavaa suunnittelutapaa*, ts. palauta mielestään valmiita menetelmiä, jotka sitten kirjoittavat ohjelmamuotoon. Sen sijaan ekspertit menettelevät näin todennäköisimmin silloin, kun ongelman sovellusalue on heille riittävän tuttu. Mikäli sovellusalue vaikuttaa oudolta, ei valmiin menetelmän palauttaminen välttämättä onnistu, ja ekspertinkin pitää yrittää ratkaista ongelma puhtaalta pöydältä aloittaen, eli *kokoavaa suunnittelua* käyttäen. Ekspertit siis hyödyntävät joustavasti molempia suunnittelutapoja, kun taas noviisit joutuvat rajallisen menetelmiin liittyvän tietämyksensä vuoksi turvautumaan pääasiassa kokoavaan suunnittelutapaan.

Ristin kehittämää terminologiaa käyttämällä ohjelman suunnitteluprosessi voi edetä kahdella tavalla: joko fokusta tai skeemaa laajentamalla. *Skeeman laajentaminen* tarkoittaa valmiin menetelmän palauttamista mielestä ja sen hyödyntämistä, eli käytännössä osittavaa suunnittelua. *Fokuksen laajentamisessa* on kyse kokoavasta suunnittelusta, eli tarkastellaan lähemmin sitä, miten ohjelmoija luo uusia menetelmiä. Jotta tarkastelu olisi mahdollista, on ohjelman rakenne jaettava useisiin tasoihin. Kaikkein alimman tason rakenne Ristin mukaan on *fokus*, joka on yksittäinen, alle rivin mittainen koodinpätkä, joka toteuttaa halutun tavoitteen. Fokus voi tarkoittaa vaikkapa jonkin muuttujan arvon kasvattamista yhdellä ”+1”. Fokaalirivi taas on se koodirivi, johon fokus sisältyy. Tällöin fokus on osa fokaaliriviä, jossa vaikkapa `laskuri` nimisen muuttujan arvoa kasvatetaan yhdellä: ”`laskuri := laskuri+1;`”. Yksinkertainen menetelmä rakentuu fokuksesta ja sen ympärille rakennetusta fokaaliriveistä. Käytännössä se voi tarkoittaa vaikkapa edellä mainitun `laskuri`-muuttujan esittelyä, alustamista alkuarvolla, arvon asteittaista kasvattamista jne. Yksinkertaiset menetelmät saatetaan yhdistää monimutkaisemmiksi menetelmiksi, joista lopullinen ohjelma koostuu.

3.2.3 Parsing–gnisrap-malli

Green & al. (1987) lähestyy ohjelmien kirjoittamista rajallisen työmuistin ja käytössä olevien ohjelmien kirjoitusvälineiden, kuten tekstieditoreiden käyttämisen näkökul-

masta *Parsing–gnisrap-mallissa*. Malliin liittyvässä tutkimuksessa havaittiin, että ohjelmia ei juuri koskaan kirjoiteta lineaarisessa järjestyksessä, ja tähän vaikuttavat monet erilaiset syyt. Näitä syitä ovat työmuistin rajallisuus, käytettävä ohjelmointikieli, ohjelmien kirjoitusympäristön asettamat rajoitukset (esimerkiksi minkä verran ohjelmakoodista on kerralla näkyvissä) ja ohjelmoijan omaama tietämys ohjelmoinnista. Lisäksi havaittiin tiettyjä säännönmukaisuuksia epälineaarisuuden suhteen, minkä tulkittiin viittaavan menetelmien käyttämiseen ohjelmia kirjoitettaessa.

Lähtökohdaksi on otettu Ristin (1989) kehittämä malli fokuksesta ja fokaaliriveistä, jota täydennetään joiltakin osin, eli liittyen siihen, milloin ohjelmointi etenee lineaarisesti ja milloin taas ei. Parsing–gnisrap ei ole kokonainen malli ohjelmien kirjoittamisesta, vaan se keskittyy selvittämään sitä, mitä tapahtuu, kun ongelman korkeamman tason ratkaisu on mietitty päässä valmiiksi ja ratkaisu pitäisi kirjoittaa lähdekoodin muotoon jollekin ulkoiselle välineelle (väline on esimerkiksi tekstieditori tai kynä ja paperi). Välinekieli on tässä mallissa kieli, jolla käyttäjä kommunikoi välineen kanssa, esimerkkinä vaikkapa Emacs-editorin komennot. Tehtäväkieli puolestaan voi olla jokin ohjelmointikieli tai muu sääntöihin pohjautuva tapa ratkaista ongelma. Mikäli ihmisen työmuistin määrä ei olisi rajallinen, voisi ns. ”ideaaliohjelmoija” ottaa käsiteltäväkseen minkä tahansa ohjelmointiongelman, käsitellä ja ratkaista sen mielessään tarvittavia skeemoja apuna käyttäen ja yhdistellä ja muodostaa niistä korkeamman tason menetelmärungon. Lopuksi tarvitsisi vain kirjoittaa valmis menetelmärunko tehtäväkieltä käyttäen lineaarisessa järjestyksessä jollekin ulkoiselle välineelle, kuten näytölle.

Ohjelmoijat käyttävät kuitenkin ulkoisia välineitä paitsi valmiin koodin kirjoittamiseen, myös väliaikaisena informaation varastoimispaikkana sekä ns. ylivuotoalueena, kun työmuisti on vaarassa joutua ylikuormitetuksi. Parsing–gnisrap-mallissa *parsing* tarkoittaa ulkoiselle välineelle kirjoitetun aineksen siirtämistä ja palauttamista takaisin muistiin ja prosessointia ymmärtämisen varmistamiseksi, *gnisrap* tarkoittaa lähdekoodin kirjoittamista muistista välineelle. Ohjelman kirjoittaminen tapahtuu siis mallin mukaan parsing–gnisrap-sykleissä, jotka etenevät luonnostaan epälineaarisesti. Tämä johtuu siitä, että ohjelmoija ei useinkaan saa kirjoitettua menetelmiä kerralla kokonaisuudessaan, vaan hän täydentää, korjailee ja muuttaa niitä tarpeen mukaan myöhemmin.

3.3 Ohjelmien ymmärtäminen

Ohjelman lähdekoodin ymmärtäminen, siinä olevien virheiden etsiminen ja korjaaminen sekä lähdekoodin muokkaaminen liittyvät läheisesti toisiinsa. Jos lähdekoodin toimintaperiaatetta ei ymmärrä, on virheiden korjaaminen ja muokkausten tekeminenkin miltei mahdotonta. Brooksin (1983) ja Penningtonin (1987) esittämillä malleilla on aiheeseen liittyvässä tutkimuksessa keskeinen asema. Lisäksi esitellään Shih'n & Alessin (1994) tekemiä havaintoja.

3.3.1 Brooksin malli ja majakat ymmärtämisen apuna

Brooksin (1983) tavoitteena on ollut kuvata kognitiivisia prosesseja, joita ohjelmoijat käyttävät, kun tavoitteena on yrittää ymmärtää ohjelman lähdekoodia. Hänen mukaansa ohjelman kirjoittamisessa on kyse ohjelmointiongelman ja ongelman ratkaisemiseen tarvittavien välimuotojen muodostamisesta. Ohjelman ymmärtämisessä nämä välimuodot pitää pystyä rakentamaan uudelleen etenemällä lähdekoodista takaisin ratkaistavaa ongelmaa kohti.

Lähdekoodin ymmärtäminen ei etene rivi riviltä tai suoraviivaisesti ylhäältä alas. Sen sijaan ohjelmoijat muodostavat lähdekoodiin perehtyessään hypoteeseja sen toiminnasta ja tutkivat sitä useaan otteeseen eri järjestyksissä. Hypoteeseja ei luoda tyhjästä eikä sattumanvaraisesti, vaan todennäköisimmin ongelmaan sopivat hypoteesit, jotka rajoittavat tehokkaimmin ongelma-avaruutta, ovat etusijalla. Ohjelmoijan aiemmalla kokemuksella on kuitenkin suuri merkitys tehtyjen hypoteesien "järkevyyteen".

Hypoteesit ovat aluksi yleisellä tasolla, ja ohjelmoijat testaavat niiden toimivuutta sekä paikkansa pitävyyttä tekemiensä havaintojen pohjalta ja tarkentavat niitä. Tarvittaessa tehty hypoteesi hylätään ja muodostetaan tilalle uusia, mikäli entiset havaitaan toimimattomiksi. Hypoteesien tutkiminen tapahtuu etsimällä lähdekoodista tiettyjä stereotyyppisiä rakenteita eli *majakoita* (beacons), kuten vaikkapa koodia, joka viittaa taulukossa olevien alkioiden lajitteluun. Majakoita ymmärtämisen apuna ovat tutkineet myös Gellenbeck & Cook (1991), jotka selvittivät proseduurien ja muuttujien nimien merkitystä ohjelman ymmärtämisessä.

Mikäli ohjelmaan perehtymisprosessi onnistui, on ohjelmoijalla useiden eri muodostettujen hypoteesien ja koodiin perehtymisen jälkeen selkeä kokonaiskäsitys ohjelmasta

ja tarvittavat välimuodot on pystytty rakentamaan.

Ohjelman ymmärtäminen ei tapahdu suoraviivaisesti, ja eri ohjelmoijien välillä on erittäin suuria eroja paitsi ymmärtämisen nopeudessa myös sen laajuudessa. Tähän on useita mahdollisia syitä, joista keskeisimpinä Brooks mainitsee erot ohjelmoijien ohjelmointitietämyksessä ja ohjelman sovellusalueeseen liittyvässä tietämyksessä sekä strategioissa, joita ohjelmoijat käyttävät etsiessään tietoa ohjelman lähdekoodista.

3.3.2 Ohjelmatietämyksen lajit

Penningtonin (1987) tavoitteena oli tutkia, millaisia strategioita eri tasoiset ohjelmoijat soveltavat ohjelmien ymmärtämisessä ja millaisia mentaalisia representaatioita eli mielessä olevia esityksiä jonkin ohjelman toiminnasta muodostuu. Penningtonin mukaan ohjelmoijat tietävät yksittäisestä ohjelmasta erilaisia asioita. Hän jakaa tuon tietämyksen seuraaviin osiin: operaatiot (OPER), ohjausvuo (CF), tietovuo (DF), tila (ST) ja tavoitteet (FN). *Operaatiot* ovat toimintoja, jotka ohjelma suorittaa, kuten vaikkapa kahden muuttujan arvon vertailu toisiinsa. *Ohjausvuo* liittyy toimintojen suoritusjärjestykseen, eli ohjelman toimintaan eri tapauksissa, kun vaikka kaksi muuttujaa ovat erisuuruisia tai yhtä suuria. *Tietovuossa* tarkastellaan tietoalkioihin, kuten yksittäisiin muuttujiin kohdistuvia muutosten sarjoja. *Tilan* avulla ilmaistaan seikkoja, jotka jonkin toiminnon suoritushetkellä pitävät paikkansa. *Tavoitteet* puolestaan ilmaisevat ohjelman päätavoitteiden ja ohjelman käytön kannalta välttämättömien alitavoitteiden hierarkian, eli ohjelmoijalla on esimerkiksi tietämys siitä, että johonkin muuttujaan tallennetaan annettujen syötteiden summa, jotta voidaan myöhemmin laskea syötteiden keskiarvo.

Pennington jakaa ohjelman ymmärtämisen tuloksena syntyvät mentaaliset representaatiot ohjelmamalliin (program model) ja sovellusaluemalliin (domain model). *Ohjelmamalli* on representaatio, joka korostaa ohjelman eri osien välillä olevia toiminnallisia relaatioita ohjelmointikielen tasolla. *Sovellusaluemallissa* taas painotetaan ohjelman eri osien välillä olevia toiminnallisia relaatioita sovellusalueen tasolla.

Penningtonin suorittamassa kokeessa koehenkilöiden tuli tehdä muutos tuotantokäytössä olevaan 200-riviseen ohjelmaan, jolla oli ainoastaan minimaalinen dokumentointi. Jotta tehtävän pystyi ratkaisemaan, piti koehenkilöiden pystyä erottamaan ohjelmasta tavoitteet sekä tietovuon ja ohjausvuon eri elementit suhteellisen hyvin. Koehenki-

löillä oli kaksi kysymyslistaa, joissa mitattiin ohjausvuon, tietovuon, tavoitteiden ja ohjelman tilojen ymmärtämistä. Ensimmäisen niistä he täyttivät ohjelmaan tutustumisen jälkeen, ja toisen ohjelmaan tekemänsä muokkauksen jälkeen. Koehenkilöt olivat ohjelmoinnin eksperttejä, joilla kaikilla oli vähintään kolmen vuoden kokemus.

Tutkimuksesta saatujen tulosten mukaan ohjelman ymmärtäminen etenee järjestyksessä ohjausvuo, tietovuo ja tavoitteet, eli ohjelmointikielen syntaksitason toimintojen tasolta edetään vähitellen korkeammalle abstraktiotasolle. Parhaiten ohjelmia ymmärtävät käyttävät ns. *ristiinviittausstrategiaa* (cross-referencing strategy), jossa vuorottelevat ohjelman systemaattinen opiskelu, saavutetun tietämyksen kääntäminen sovellusalueen ilmaisuiksi sekä sovellusaluetta koskevan hypoteesin verifiointi ohjelmakoodista. Toisin sanoen parhailla ymmärtäjillä on hyvä käsitys ohjelman toiminnasta paitsi lähdekoodin, myös sovellusalueen tasolla. Henkilöt, jotka keskittyvät vain ohjelmatasoon ymmärtävät mitä ohjelma tekee, mutta eivät ymmärrä, että miksi. Pelkästään sovellusaluetasoon keskittyvät eivät puolestaan ymmärrä ohjelmatason tapahtumia.

3.3.3 Erot mentaalisissa ja käsitteellisissä malleissa

Shih & Alessi (1994) ovat tutkineet käsitteellisten mallien (conceptual model) käyttämistä noviisien mentaalisiin malleihin vaikuttamiseksi ohjelmoinnin opettamisen yhteydessä. *Käsitteellisellä mallilla* he tarkoittavat opettajan oppilaalle tarjoamaa "oikeaa" mallia jostakin asiasta ja sen toiminnasta ja mentaalisella mallilla oppijan senhetkistä käsitystä opittavasta asiasta.

Tutkimukseen osallistuneilla koehenkilöillä ei ollut aiempaa ohjelmointikokemusta. Tavoitteena oli ensinnäkin selvittää, millä tavalla ohjelman ymmärtämisen yhteydessä annetut käsitteelliset mallit, jotka annettiin visualisointien ja animaatioiden muodossa, vaikuttavat koehenkilöiden kykyyn ymmärtää ohjelmia. Lisäksi tutkittiin, vaikuttavatko ohjelmien ymmärtämisen yhteydessä saadut käsitteelliset mallit myös koehenkilöiden kykyyn kirjoittaa ohjelmia. Vaikka tutkijat toteavat, että paras tapa oppia ohjelmien kirjoittamista on kirjoittaa niitä, käsitteellisten mallien käyttö ja ohjelmien ymmärtämistehtävät näyttävät tutkimuksen mukaan helpottavan niin ohjelmien ymmärtämistä kuin myös kirjoittamista.

3.3.4 Yhteenveto tutkimuksista ja noviisien keskeisimmistä ongelmista

Kokemuksen myötä ohjelmoijille kertyy tietämystä, jota kutsutaan menetelmiksi. Tätä kertynyttä tietämystä voidaan soveltaa, kun ratkaistaan uusia ohjelmointiongelmia. Mikäli osittain tai täysin valmis menetelmä ongelman ratkaisuun on tiedossa, voi ratkaisu edetä skeemaa laajentamalla. Muussa tapauksessa joudutaan ratkaisu laatimaan alusta alkaen käyttämällä apuna mm. ongelmanratkaisuheuristiikkoja. Ohjelmien kirjoittaminen ei etene lineaarisessa järjestyksessä kuin poikkeustapauksissa. Tämä johtuu muun muassa lyhytkestoisen muistin, ohjelmointikielen ja työskentelyvälineiden rajoituksista.

Ohjelman ymmärtämiseksi ohjelmoija tekee hypoteeseja sen toiminnasta. Hypoteesien toimivuutta testataan, ja tarvittaessa niitä muokataan, tarkennetaan ja hylätään. Ymmärtämisprosessin nopeus ja sen lopputuloksena saatava ymmärrys vaihtelee hyvin voimakkaasti eri ohjelmoijien kohdalla. Ymmärtäminen etenee lähdekoodin tasolta kohti korkeampaa abstraktiotasoa. Hyvän ymmärryksen lopputuloksena ohjelmoijalla on selkeä käsitys ohjelman toiminnasta paitsi ohjelmatasolla, myös sovellusaluetasolla.

Noviisien ongelmat ohjelmoinnissa eivät enimmäkseen liity siihen, etteivätkö he tuntisi ohjelmointikielten rakenteita, vaan kyse on siitä, että heillä ei ole hallussaan tarvittavia keinoja (menetelmiä), joiden avulla he osaisivat soveltaa ohjelmointikielen rakenteita ohjelmia kirjoitettaessa. Sama ilmiö tulee esiin myös ohjelmien ymmärtämisessä. Koska noviisilla ei ole tietämystä siitä, kuinka jokin asia toteutetaan ja miltä se voisi näyttää lähdekoodin tasolla, on vaikea esittää valistuneita hypoteeseja lähdekoodin toimintaperiaatteista, saati tehdä ohjelmaan järkeviä muutoksia. Vaarana on, että ymmärrys jonkin ohjelman toiminnasta jää tällöin ainoastaan ohjelmointikielen syntaksin tasolle.

Kaikilla tässä luvussa esitellyillä ohjelmointiin liittyvillä malleilla on voimakkaita rajoituksia. Niiden avulla pystytään selittämään ja tarkastelemaan tiettyjä ohjelmointiin liittyviä ilmiöitä ja lainalaisuuksia ainoastaan yleisellä tasolla, sillä ohjelmoinnissa on kyse niin monimutkaisista prosesseista. Tutkimustuloksiin ja niiden yleispätevyyteen kannattaa suhtautua aina kriittisesti, sillä esimerkiksi ohjelmien ymmärtämisprosessi etenee hyvin eri tavoin, mikäli ohjelman lähdekoodi pituus on 50 riviä tai vaikka 500 riviä.

Maallikko saattaisi yksinkertaisesti sanoa noviisin ongelmallisen tilanteen johtuvan ko-

kemuksen puutteesta ja tässä tiivistyykin asian ydin. Kognitiivisen psykologian puolella asian sanotaan johtuvan kehittymättömistä skeemoista tai mentaalisista malleista (jotka kehittyvät kokemuksen myötä), ohjelmoinnin empiirisen tutkimuksen ja ohjelmoinnin psykologian termeillä tilannetta voitaisiin kutsua vaikkapa puutteelliseksi menetelmätietämykseksi.

3.4 Ohjelmoinnin alkeiden opettamisesta

Ohjelmoinnin alkeita voi opettaa erittäin monella tavalla. Alkeiden opetukseen liittyy useimmiten sama perussisältö, eli käydään läpi erilaisia ohjelmoinnin kannalta keskeisiä käsitteitä, kuten muuttujat, ehdollisuuden ilmaiseminen, alkeistietotyypit, silmukkarakenteet ja erilaiset operaattorit. Tästä huolimatta ohjelmointikurssien käytännön toteutustavat saattavat poiketa toisistaan hyvinkin voimakkaasti johtuen mm. käytettävissä olevista resursseista, käytetystä ohjelmointiparadigmasta ja työelämän asettamista vaatimuksista. On syytä korostaa, että alkeiskurssien tehtävänä ei missään nimessä ole muuttaa noviiseja yhdessä yössä eksperteiksi, vaan ainoastaan tarjota mahdollisuus hankkia vaadittava pohjatietous, jonka varaan jokainen voi alkaa rakentaa omaa ohjelmointitietämystään.

Valitun opetuskielen ja siihen aina liittyvän ohjelmointiparadigman vaikutus opetuksen sisältöön saattaa olla hyvinkin suuri, joten valinta on suoritettava huolella. Työelämässä käytössä olevien kielten lisäksi ohjelmointikursseilla käytetään myös erilaisia varta vasten alkeiskurssien tarpeisiin kehitettyjä yksinkertaistettuja ohjelmointikieliä. Eräs esimerkki on Joensuun yliopiston tietojenkäsittelytieteen laitoksen Johdatus ohjelmointiin -kurssi, jossa on käytetty apuna Karel-kieltä. Yksinkertaistettujen ohjelmointikielten käyttöä perustellaan mm. sillä, että keskeiset ohjelmoinnissa käytettävät periaatteet on helpompi ymmärtää, mikäli käytössä ei ole lyhytkestoista muistia kuormittavaa ylimääräistä tietoa. Onkin otettava huomioon, että ohjelmoinnin opetuksen kohderyhmät saattavat poiketa hyvinkin paljon toisistaan. Yksinkertaistettujen ohjelmointikielten käyttö saattaa ärsyttää niitä opiskelijoita, jotka ovat aiemmin kokeilleet ohjelmointia ja päässeet ns. oikeiden ohjelmointikielten makuun. Kun ohjelmointia taas opetetaan noviiseille, jotka eivät ole koskaan ohjelmoineet tai esimerkiksi lapsille, voi yksinkertaistetun ohjelmointikielen käyttäminen olla erittäin toimiva ratkaisu. Vaikka ohjelmointi on erittäin käytännönläheinen taito, niin sitä opetetaan myös käyttämättä mitään konkreettista ohjelmointikieltä (Fincher, 1999), jolloin ohjelmoinnissa

tarvittavat käsitteet selvennetään pseudokieliä käyttämällä.

Lähestymistavat ohjelmoinnin opetuksen kehittämiseksi voidaan jakaa karkealla tasolla kahteen kategoriaan, eli on keskitytty siihen, *miten* opetetaan ja *mitä* opetetaan. Yleisesti ottaen on painotettu näistä ensimmäistä, eli viime vuosina muutoksia ei ole kokenut ohjelmointikurssin asiasisältö, vaan ainoastaan tapa, jolla asiat esitetään. Tästä esimerkkeinä käydään läpi käsitteiden konkretisointi, Sokrateen tekniikka, oppipoikamalli, ongelmalähtoinen oppiminen, etäopetus sekä visualisoinnin ja animoinnin käyttö. Huomattavasti harvinaisempia ovat olleet innovaatiot ohjelmointikurssien varsinaiseen asiasisältöön liittyen, eli siihen, mitä opiskelijoille opetetaan. Esimerkkinä tästä mainitaan suunnittelumallien käyttäminen, ja tämän työn kannalta äärimmäisen olennainen muuttujien roolitus, josta kerrotaan tarkemmin seuraavassa luvussa. Tarkoituksena ei ole asettaa edellä mainittuja menetelmiä paremmuusjärjestykseen, koska niillä kaikilla on omat hyvät ja huonot puolensa. Kyseessä ei myöskään ole mikään kaiken kattava lista, vaan olennaisinta on saada yleiskuva siitä, mitä kaikkea ohjelmoinnin opetuksen saralla on kokeiltu.

3.4.1 Käsitteiden konkretisointi

Käsitteiden konkretisoinnissa ideana on selventää uuteen opetettavaan asiaan liittyviä käsitteitä käyttämällä apuna opetettaville ennestään tuttuja käytännön esimerkkejä muilta elämän alueilta. Ohjelmointiin liittyen käsitteiden konkretisointia hyödynnetään usein oliokeskeisten kielten opetuksessa. Luennoitsijat käyttävät oliomaailman käsitteitä kuvatessaan erilaisia reaalimaailman esimerkkejä, mutta tätä voidaan viedä vielä askeleen verran eteenpäin: tuodaan konkreettiset esineet esimerkiksi luentosaliin selventämään vaikealta tuntuvia käsitteitä. Konkretisoinnissa on käytetty apuna mm. teletappeja (Poon, 2000) ja laitteistoja, joissa LED-valot vilkkuvat ohjelmoijan tekemien muutosten mukaan (Lobo & al. 2000). Joensuun yliopiston tietojenkäsittelytieteen laitoksella ohjelmointikäsitteitä on konkretisoitu vaikkapa Kids' Club -projektissa, jossa ohjelmointia on pyritty tekemään tutuksi 10–15-vuotiaille mm. ohjelmoitavien legopalikoiden muodossa (Kids' Club, 2004). Konkreettisilla esimerkeillä näyttää olevan opiskelijoita motivoiva ja innostava vaikutus.

3.4.2 Sokrateen tekniikka

Sokrateen tekniikassa (Socratic approach) tavoitteena on saada oppija käyttämään aktiivisesti hyväkseen niin paljon omasta tietämyksestään kuin mahdollista. Ohjelmoinnin yhteydessä keinoa on sovellettu esimerkiksi ohjelmissa olevien virheiden korjaamiseen. Aluksi noviisilla on hänen itsensä kirjoittama ohjelmanpätkä, jossa on jokin virhe. Häntä pyydetään kertomaan ohjaajalle tai tutorille omalla äidinkielellä tai pseudokieltä käyttäen, mitä ohjelman pitäisi tehdä ja mikä kohta siinä aiheuttaa vaikeuksia. Tavoitteena on opastaa noviisia analysoimaan tehokkaammin omaa ohjelmaansa, koska perustietämys ongelman ratkaisemiseksi on noviisilla jo olemassa, se on vain piilevässä muodossa. Ongelmaa toiselle ihmiselle selitettäessä ongelman aiheuttaja usein selviää, vaikka kuunteleva osapuoli ei sanoisi mitään (Wilson, 1987).

3.4.3 Oppipoika-malli

Oppipoika-mallia (apprenticeship model) on käytetty perinteisesti etenkin käsityöalan ammattien yhteydessä. Mestari(e)n töitä ja työskentelytapoja tarkkailemalla ja itse kokeilemalla ja tekemällä oppipoika saavuttaa vähitellen haluamansa tiedot ja taidot. Ohjelmoinnissa malleina voidaan käyttää kokeneempien ohjelmoijien kirjoittamia sovelluksia, joiden lähdekoodia noviisi aluksi lukee, perehtyy vähitellen tarkemmin sekä viimeisessä vaiheessa muokkaa sitä lopulta itse. Tavoitteena on, että noviisi näkee todellisen elämän esimerkkejä, jotka vaikeutuvat pikkuhiljaa tietämyksen kasvaessa. Samalla noviisi oppii selkeän ja loogisen suunnittelun merkityksen ohjelman kirjoittamisessa (Astrachan & Reed, 1995).

3.4.4 Ongelmalähtöinen oppiminen

Ongelmalähtöisessä oppimisessa (PBL, problem based learning) korostetaan oppijan aktiivista ja itsenäistä roolia uusien asioiden omaksumisessa. Opettajan tehtävänä on luoda tarvittavat puitteet oppimisen mahdollistamiseksi. Käytännössä tämä tarkoittaa paitsi opettajan esittämiä ongelmia, joita oppijoiden pitää yrittää ratkaista yksin tai ryhmätyönä, myös opastamista tarvittavien resurssien pariin, jotka voivat auttaa ongelman ratkaisussa. Ohjelmointikurssien yhteydessä ratkaistavat ongelmat voivat aluksi olla tarkkaan määriteltyjä ja rakennettuja, mutta pitemmälle ehtineille voidaan antaa rat-

kaistavaksi ongelmia, jotka ovat huonommin jäsenneltyjä ja joihin ei ole yhtä ja oikeaa ratkaisua. Toisin sanoen kyseessä ovat sentyyppiset ongelmat, joita tietojenkäsittelyn ammattilaiset tulevat kohtaamaan myös työelämässä (Ellis & al., 1998).

3.4.5 Etäopetus

Tekninen kehitys on mahdollistanut kokonaisten kurssien suorittamisen etäopetuksena verkkomateriaalia käyttäen ilman, että opiskelijan tarvitsee lähteä kotoaan opiskelujensa vuoksi. Tarjolla on niin maksullisia kuin ilmaisiakin kursseja eri aiheista, eräänä niistä ohjelmointi. Tämä avaa uusia mahdollisuuksia syrjäisemmillä seuduilla asuville, mutta oman aktiivisuuden merkitys oppimisessa korostuu. Vaikka opetus järjestettäisiinkin etäopetuksen muodossa, on silti tärkeää, että opiskelija voi tarvittaessa saada nopeasti vastaukset mieltään askarruttaviin kysymyksiin. Tässä mielessä ihmiseltä–ihmiselle-näkökulma on olennaisen tärkeää myöskin etäopetusta hyödynnettäessä. Mm. Internetin vaikutus näkyy kurssien järjestelyssä muutenkin kuin etäopetuksen muodossa. Nykyään useilla kursseilla on omat verkkosivut, vaikka itse kurssin suorittaminen vaatisikin luennoille tai harjoituksiin osallistumista. Tämän voi havaita vaikkapa Joensuun yliopiston tietojenkäsittelytieteen laitoksen sivuilta. Verkkosivujen tarpeellisuus on helppo perustella jo pelkästään sisäisen tiedonkulun parantamisen tarpeella.

3.4.6 Visualisoinnin hyödyntäminen

Yksinkertaisimpia esimerkkejä visualisointiin liittyen ovat tekstieditorit, joissa lähdekoodin luettavuutta yritetään parantaa värejä ja sisennystä käyttämällä (Baecker, 1988). Tällaisten editorien käyttäminen ei sinänsä ole mikään erityinen ohjelmoinnin opetusmenetelmä, mutta se kannattaa mainita tarpeellisena apukeinona, koska ohjelmoinnin oppimiseen liittyy myös kunnollisten työvälineiden käytön hallitseminen.

Hundhausen & al. (2002) on tutkinut algoritmien visualisoinnin hyödyntämistä tietojenkäsittelytieteen opetuksessa ja sitä, kuinka visualisoinnista saadaan irti suurin mahdollinen hyöty oppimisen kannalta. Algoritmien visualisointeja on käytetty apuna esimerkiksi luennoilla, abstraktien tietotyyppien toimintaperiaatteiden selvittämisessä ja etsittäessä virheitä oppilaiden tekemistä linkitettyjä listoja sisältävistä ohjelmista. Tutkimuksen keskeinen tulos oli, että eniten huomiota algoritmien visualisoinnissa ei kan-

nata kiinnittää siihen, mitä algoritmien visualisointityökalu näyttää oppijalle. Tämän sijasta on syytä panostaa siihen, miten oppija käyttää visualisointeja oppimiseen. Mikäli opiskelijat vain seuraavat passiivisena visualisoinnin etenemistä, eivät tulokset oppimisen kannalta ole hyviä. Jos sen sijaan opiskelija voi itse ottaa aktiivisesti osaa visualisoinnin etenemiseen vaikkapa kokeilemalla erilaisia entä jos -tilanteita tai pyrkimällä ennustamaan mitä seuraavaksi visualisoinnissa tapahtuu, ovat tulokset paljon parempia. Algoritmien visualisoinnin käyttäminen sinänsä ei siis takaa parempia oppimistuloksia, mutta jos visualisointien hyödyntämistä mietitään etukäteen huolellisesti, voi siitä olla paljon apua vaikeiden asioiden sisäistämisessä.

3.4.7 Suunnittelumallit

Suunnittelumallien (software design patterns) idea pohjautuu arkkitehti Christopher Alexanderin ajatuksiin rakentamisesta ja arkkitehtuurista. Alexander määritteli malliin kuuluvan kolme osaa: *kontekstin* (context), *ongelman* (system of forces) sekä *ratkaisun* (solution). Yhdessä ne muodostavat siis kuvauksen, jossa on määritelty tietty konteksti, ongelma sekä ratkaisu kyseiseen ongelmaan, joka pätee määritellyssä kontekstissa. Ohjelmoinnin puolella suunnittelumalleja alettiin ensimmäisenä hyödyntää olio-ohjelmoinnissa, mutta suunnittelumallit on kuitenkin ohjelmointikielestä ja -paradigmasta riippumaton apukeino. Tarkoituksena on helpottaa ohjelmointiongelmien ratkaisemista käyttämällä apuna käytännössä hyväksi havaittuja ja toimivia korkeamman tason ratkaisumalleja. Niiden avulla suunnitellun ohjelmiston toteuttaminen käytännössä ei ole niin vaikeaa, ja lisäksi saadaan aikaan vähemmän virheitä sisältäviä, helpommin uudelleenkäytettäviä ja hyvin suunniteltuja ohjelmistoja. Pyrkimyksenä on samalla välttää turhaa työtä eli ”keksinpä-juuri-pyörän-uudelleen”-tyyppisiä tilanteita (Hillside.net, 2004).

Pohjimmiltaan suunnittelumalleissa on kyse samasta asiasta kuin aiemmin tässä luvussa mainituissa menetelmissä (Clancy & Linn, 1999). Suunnittelumallien avulla menetelmille yritetään vain antaa jokin konkreettinen esitysmuoto, ja näin välittää ohjelmoijien saavuttamaa tietämystä jostakin ongelma-alueesta ja niihin tunnetuista ratkaisuista myös muille.

Suunnittelumalleja voidaan käyttää paitsi suurten ohjelmistokomponenttien suunnittelussa ja toteutuksessa, myös esimerkiksi tarkasti rajattujen, yksittäisten ohjelmointiin liittyvien alueiden, kuten tietorakenteiden (Nguyen, 1998) ja ohjelmoinnin alkeiden

(Wallingford, 2004) opetuksessa. Yksittäinen suunnittelumalli voi olla melkein mitä tahansa: se voi olla vaikka eksperteille tarkoitettu malli webbikaupan ostoskorin toteuttamisesta tai esimerkiksi noviiseille tarkoitettu malli silmukkarakenteiden käyttämisestä.

Suunnittelumallien käytölle erityisesti ohjelmoinnin alkeiden opetuksessa on hyvät perusteet, sillä ne tarjoavat keinon esitellä eksperttien piilevän tietouden tasolla olevaa tietämystä myös noviisien käytettäväksi. Ohjelmoinnin alkeiden opetuksen yhteydessä suunnittelumallien käytöllä pyritään siihen, että noviisit oppivat joukon yksinkertaisten ohjelmointiongelmien ratkaisumalleja. Vähitellen noviisit oppivat käyttämään malleja systemaattisesti apunaan sekä järjestelevät, analysoivat ja yhdistelevät mielessään itselleen parhaiten sopivia malleja. Suunnittelumallien hyödyntäminen ohjelmoinnin alkeiden opetuksessa vaatii kurssin vetäjiltä suurta huolellisuutta ja tietämystä siitä, miten esittää suunnittelumalleihin liittyvä opetus siten, että oppijat oikeasti ymmärtävät, mistä malleissa on kyse (Clancy & Linn, 1999).

3.4.8 Yhteenveto opetuksesta

Yllä mainittujen opetustapojen yhteydessä voi helposti havaita tietyn yhtenäisen piirteen: ne kaikki yrittävät saada oppijan jollakin tavalla aktiivisesti itse ajattelemaan ja miettimään opetettavaa asiaa. Kun asian yhdistää toisessa luvussa mainittuihin skeemoihin (tai mentaalisiin malleihin), niin perustelu tälle toiminnalle on selvä: tavoitteena on saada oppija tavalla tai toisella kyseenalaistamaan ja kehittämään mentaalisia mallejaan ja skeemojaan. Tapoja tuon tavoitteen saavuttamiseksi on monia, eikä yhtä ainoaa ja oikeaa keinoa, joka auttaisi kaikkia oppimaan tehokkaasti, luultavasti koskaan löydetä.

Jenkinsin (2001) mukaan aivan liian usein näkee viimeisen vuoden tietojenkäsittelytieteen opiskelijoita, jotka yrittävät vältellä ohjelmointiin liittyviä kursseja kaikin mahdollisin keinoin. Olisi helppoa laittaa kyseinen ilmiö pelkästään opiskelijoiden laiskuuden syyksi, mutta täytyy ohjelmoinnin opetuksessakin olla jotakin korjattavaa. Ohjelmointikurssien vetäjien tärkein tehtävä onkin Jenkinsin mielestä löytää keinot, joilla mahdollisimman moni opiskelijoista voidaan motivoida oppimaan ja uskomaan omiin kykyihinsä. Opettajan ja oppilaiden välisellä aidolla ja molemminpuolisella luottamuksella sekä arvostuksella on tässä suhteessa suuri merkitys. Toisin sanoen jos ajatellaan, että pitääkö valita keppi vaiko porkkana, niin opetustavan pitää olla suurimmalta osal-

ta porkkanan puolella. Ohjelmointikielen syntaksin yksityiskohtainen läpikäynti luentojen yhteydessä on siis toissijaista. Jos motivointi onnistuu hyvin, ovat odotettavissa olevat tuloksetkin hyviä. Lisäksi opiskelijoille tulisi muistaa korostaa, että virheiden tekemistä ei tarvitse pelätä, sillä nehän johtavat todennäköisesti paremmin kehittyneisiin mentaalisiin malleihin.

Tässä luvussa pohdittiin ohjelmoinnin eksperttiyteen liittyviä asioita ja määriteltiin laveasti, mitä noviisi ja ekspertti tarkoittavat ohjelmoinnin yhteydessä. Ohjelmoinnin empiirinen tutkimus näyttää tukevan kognitiivisen psykologian puolella yleisellä tasolla tehtyjä havaintoja oppimisesta ja siihen liittyvistä prosesseista. Samat rajoitukset niin fyysisellä kuin henkisellä puolella pätevät myös ohjelmoinnissa.

Eksperteillä on hallussaan suuri joukko erilaisia menetelmiä (=kehittyneitä skeemoja tai mentaalisia malleja), jotka noviiseilta vielä puuttuvat. Pelkkä tietämys ohjelmointikielen syntaksista ei auta, jos kyky soveltaa sitä puuttuu, eli menetelmiin liittyvä tieto on puutteellista. Tällöin noviisin on ongelmatilanteessa turvauduttava toisessa luvussa mainittuihin ongelmanratkaisutapoihin ja yritettävä kehittää ratkaisu alusta alkaen, kun taas ekspertti voi usein soveltaa kokemuksensa myötä hankkimiaan valmiita ratkaisumalleja.

4 Muuttujien roolit, PlanAni-animaattori ja muuttujien roolien opettaminen

Edellisen luvun perusteella havaittiin, että ohjelmointikursseilla voidaan käyttää runsaasti erilaisia lähestymistapoja oppimistulosten parantamiseksi. Suunnittelumalleja lukuun ottamatta edellä mainitut opetusmenetelmät keskittyvät kuitenkin siihen, miten asia opetetaan, eikä siihen, mitä opetetaan. Tässä luvussa esitellään käsite *muuttujien roolit*, joka on erittäin potentiaalinen sisällöllinen uudistus ohjelmointikursseja ajatellen. Lisäksi kerrotaan muuttujien roolien opettamisessa käytettävästä PlanAni-animaattorista sekä käydään läpi muuttujien roolien opettamiseen liittyviä asioita. Lopuksi mainitaan vielä lyhyesti muuttujien roolien mahdollisia muita käyttötarkoituksia. Ellei muuta lähdettä ole mainittu, tiedot muuttujien rooleista kohtaan 4.4 ”Muuttujien roolien visualisointi ja PlanAni-animaattori” asti perustuvat Sajaniemen (2003) Roles of Variables -sivustoon. Muuttujien roolien visualisoinnista ja PlanAni-animaattorista kertova osuus puolestaan pohjautuu Sajaniemen & Kuittisen (2003) artikkeliin ja roolien opettamiseen liittyvä teksti Kuittisen & Sajaniemen (2004) artikkeliin.

4.1 Johdatus muuttujien rooleihin

Muuttujien roolit on alunperin kehitetty proseduraalisten kielten opetuksen yhteydessä käytettäväksi, mutta sen soveltuvuutta myös olio-ohjelmoinnin yhteydessä käytettäväksi tutkitaan. Muuttujien roolituksen etuina ohjelmoinnin alkeiden opetuksessa suunnittelumallien käyttöön nähden on roolituksen yksinkertaisuus ja nopea opittavuus, kun taas suunnittelumalleja on olemassa rajaton määrä. Ennen kuin selitetään tarkemmin, mitä muuttujien roolit ovat, tarkastellaan, miksi kyseinen käsite on yleensä kehitetty ja mitä sen avulla pyritään saamaan aikaan.

4.1.1 Ohjelmointitietämyksestä

Ohjelmointitietämys voidaan jakaa kolmeen osa-alueeseen, eli *ohjelmointikieleen liittyvään tietämykseen, ohjelmatietämykseen* jostakin tietystä ohjelmasta sekä *ohjelmointiin liittyvään tietämykseen*. Ohjelmoijan kannalta tärkein osa-alueista on viimeisin, eli ohjelmointiin liittyvä tietämys. Perinteisesti ohjelmointikursseilla opetus keskittyy kui-

tenkin kahteen ensimmäiseen tietämyksen osaan, johtuen lähinnä siitä, että ei tarkalleen tiedetä, mitä ohjelmointiin liittyvä tietämys oikeastaan on. Ohjelmointikurssien aikana käydään lävitse yleensä jonkin ohjelmointikielen syntaksia, mahdollisesti joitakin suoranaisesti kyseiseen ohjelmointikieleen liittyviä ohjelmointitekniikoita, kuten rekursiota, sekä lisäksi vielä joitakin esimerkkiohjelmia. Täten noviisit jäävät suurilta osin ilman tärkeintä ohjelmointitietämyksen osaa, eli ohjelmointiin liittyvää tietämystä. Tämän vuoksi noviisien on pakko yrittää kehittää ohjelmointiin liittyvää tietämystä itsenäisesti yhdistelemällä mielessään tietoutta kahdesta ensimmäisestä osa-alueesta.

4.1.2 Muuttujien rooleista yleisellä tasolla

Muuttujien roolit -käsitteen avulla voidaan päästä käsiksi ohjelmointiin liittyvään tietämykseen, joka on eksperttiohjelmoijilla jo olemassa piilevänä tietoutena. Muuttujien rooleja käyttämällä osa tästä piilevästä tietoudesta voidaan suoraan opettaa noviiseille tarkoitetuilla ohjelmointikursseilla. Ohjelmoinnin kanssa vähänkin toimineet ovat luultavasti kuulleet Charles Simonyin kehittämästä *unkarilaisesta notaatiosta* (Microsoft MSDN, 2003), jossa muuttujat nimetään tietyn ennalta mietityn käytännön mukaisesti riippuen siitä, miten muuttujia kyseisessä ohjelmassa käytetään. Yksiselitteistä ja johdonmukaista nimeämiskäytäntöä käytetään unkarilaisessa notaatiossa mm. lähdekoodin ymmärtämisen helpottamiseen. Muuttujien rooleissakin ideana on luokitella muuttujia, mutta huomiota ei kuitenkaan kiinnitetä suoranaisesti siihen, miten muuttujaa käytetään yksittäisessä ohjelmassa. Sen sijaan rooleihin jako tapahtuu yleisemmällä tasolla eikä se ole riippuvainen yksittäisestä ohjelmasta: luokittelun idea perustuu siihen, millä tavoin muuttujia käytetään apuna ohjelmointiongelmien ratkaisemisessa eri ohjelmissa, mitkä ovat muuttujien perättäiset arvot sekä mitkä ovat näiden arvojen eliniät ohjelman suorituksen aikana.

4.1.3 Esimerkki muuttujien rooleista

Muuttujia ei käytetä ohjelmoinnissa sattumanvaraisesti, vaan tietyt muuttujien käyttötavat toistuvat eri ohjelmissa yhä uudelleen. Seuraavan lyhyen kuvassa 3 olevan Pascal-kielisen ohjelman avulla voidaan havainnollistaa roolien toimintaa.

Ohjelmassa on kolme integer-tyyppistä muuttujaa: `tieto`, `lukumaara` ja `arvo`. Ensimmäisessä silmukassa käyttäjältä pyydetään syötteenä, kuinka monta kierrosta tois-

```
(*1*)   program tuplaus;
(*2*)   var tieto, lukumaara, arvo: integer;
(*3*)   begin
(*4*)     repeat
(*5*)       write('Syota haluttu kierrosmaara: '); readln(tieto)
(*6*)     until tieto > 0;
(*7*)     lukumaara := tieto;
(*8*)     while lukumaara > 0 do begin
(*9*)       write('Syota jokin luku: '); readln(arvo);
(*10*)      writeln('Luku ', arvo, ' kerrottuna kahdella on ', 2*arvo);
(*11*)      lukumaara := lukumaara - 1
(*12*)    end
(*13*)  end.
```

Kuva 3: Pascal-kielinen rooli-esimerkkiohjelma Sajaniemeä (2003) mukaillen.

ta silmukkaa pyöritetään. Lukua pyydetään toistuvasti, kunnes käyttäjä on syöttänyt positiivisen luvun. Muuttujaa `tieto` käytetään viimeksi syötetyn tiedon tallentamiseen, eikä ohjelma voi tietää, mitä arvoja käyttäjä syöttää. Muuttujaa `arvo` käytetään vastaavalla tavalla toisessa silmukassa, eli siihen tallennetaan käyttäjän viimeksi antama syöte, eikä muuttujan uudella arvolla ole mitään yhteistä sen vanhan arvon kanssa. `Lukumaara`-muuttuja sen sijaan käyttäytyy eri tavalla kuin kaksi ensin mainittua muuttujaa. Kun se on kerran alustettu, se muuttaa arvoaan yksi kerrallaan alaspäin, kunnes saavuttaa raja-arvonsa eli nollan.

4.2 Muuttujien roolien esittely

Muuttujien roolituksen avulla voidaan havaita, mikä on eri muuttujien tehtävä ohjelmassa ja miten ne käyttäytyvät. Muuttujien roolituksessa esimerkkiohjelman (kuva 3) muuttujat `tieto` ja `arvo` olisivat nimeltään *tuoreimman säilyttäjiä* ja `luku` olisi rooliltaan *askeltaja*. Tietyt roolit toistuvat ohjelmissa kerta toisensa jälkeen, ja kymmenen erilaisen roolin avulla voidaan luokitella 99% noviisitason ohjelmien muuttujista. Seuraavaksi käydään läpi kaikki kymmenen roolia Pascal-kielisiä esimerkkiohjelmia apuna käyttäen.

Muuttujan roolilla tarkoitetaan muuttujan dynaamista luonnetta, joka ilmenee sarjana muuttujan perättäisiä arvoja suhteessa muihin muuttujiin ja ulkoisiin tapahtumiin. Varsinaisia muuttujien rooleja on kymmenen kappaletta. Seuraavassa on mainittu roolien suomenkielisten nimien jälkeen suluissa myös englanninkieliset vastineet. Roolit ovat nimeltään *kiintoarvo* (fixed value), *askeltaja* (stepper), *tuoreimman säilyttäjä* (most-recent holder), *sopivimman säilyttäjä* (most-wanted holder), *kokooja* (gatherer), *yksi-*

suuntainen lippu (one-way flag), *seuraaja* (follower), *tilapäissäilö* (temporary), *järjestelijä* (organizer) ja *muunnos* (transformation). Muuttujat, jotka eivät kuulu mihinkään näistä mainituista rooleista, luokitellaan roolijaossa kategoriaan *muu* (other), joka tosi asiassa on käytössä vain siksi, että kaikki muuttujat sopisivat johonkin rooliin.

4.2.1 Kiintoarvo (fixed value)

Kiintoarvo on muuttuja, jonka arvo ei muutu ohjelman suorituksen aikana muuttujan alustamisen jälkeen. *Kiintoarvo* ja ohjelmointikielen vakio (esim. Pascalin const) ovat eri asia. Olennaista on kuitenkin, että muuttujan arvo ei ohjelman suorituksen aikana muutu. Seuraavassa esimerkkiohjelmassa (kuva 4) muuttuja `r` on *kiintoarvo*. Kyseinen muuttuja saa ohjelman suorituksen aikana yhden kerran arvon (rivillä 7), joka ei sen jälkeen muutu. *Kiintoarvoa* voidaan luonnollisesti käyttää ohjelman eri kohdissa alustamisen jälkeenkin, kunhan sen arvo ei käyttämisen yhteydessä muutu. Esimerkkiohjelmassa *kiintoarvoa* käytetään apuna rivillä 8.

```
(*1*)  program YmpyranAla (input,output);
(*2*)  const PII = 3.14;
(*3*)  var r: real;
(*4*)  begin
(*5*)     writeln;
(*6*)   write (’Anna ympyrän säde: ’);
(*7*)   readln (r);
(*8*)     writeln (’Ympyrän ala on ’, PII * r * r)
(*9*)  end.
```

Kuva 4: Kiintoarvo-roolin esimerkkiohjelma Sajaniemen (2003) mukaan.

4.2.2 Askeltaja (stepper)

Askeltaja on muuttuja, joka käy läpi arvoja jollakin systemaattisella tavalla. Esimerkkiohjelmassa (kuva 5) on silmukkarakenne, jossa käytetään muuttujaa `kertoja` *askeltajana*. *Askeltajaa* voidaan käyttää myös esimerkiksi lukumäärän laskemiseen ja taulukon indeksien läpikäymiseen.

```
(*1*) program Kertotaulu (output);
(*2*) var kertoja: integer;
(*3*) begin
(*4*)    for kertoja := 1 to 10 do
(*5*)       writeln(kertoja, ' * 3 = ', kertoja*3)
(*6*) end.
```

Kuva 5: Askeltaja-roolin esimerkkiohjelma Sajaniemen (2003) mukaan.

4.2.3 Tuoreimman säilyttäjä (most-recent holder)

Tuoreimman säilyttäjä on muuttuja, jonka arvo on viimeisin jostakin joukosta läpikäyty arvo tai arvo, joka on syötetty viimeksi. Esimerkkiohjelma (kuvassa 6) pyytää käyttäjältä syötettä toistuvasti (rivillä 5), kunnes syöte on kelvollinen. Tässä ohjelmassa muuttuja `s` on *tuoreimman säilyttäjä*, koska siitä löytyy kulloinkin viimeksi syötetty arvo.

```
(*1*)  program NelionAla (input,output);
(*2*)  var s: real;
(*3*)  begin
(*4*)     repeat
(*5*)      write ('Anna neliön sivu: ');
(*6*)      readln (s)
(*7*)   until s > 0;
(*8*)   writeln('Neliön ala on ', s * s)
(*9*)  end.
```

Kuva 6: Tuoreimman säilyttäjä -roolin esimerkkiohjelma Sajaniemen (2003) mukaan.

4.2.4 Sopivimman säilyttäjä (most-wanted holder)

Sopivimman säilyttäjä on muuttuja, jonka arvo on ”paras” tai muulla tavalla halutuin siihen asti läpikäydyistä arvoista. Arvojen paremmuuden mittaamisessa ei ole mitään rajoituksia: halutuin voi tarkoittaa esimerkiksi pienintä tai suurinta lukua tai sellaista lukua, joka on lähinnä jotain tiettyä arvoa. Esimerkkiohjelma (kuva 7) selvittää, mikä käyttäjän syöttämistä kymmenestä kokonaisluvusta on pienin. Muuttuja `pienin` on *sopivimman säilyttäjä*, koska siihen sijoitetaan (rivillä 8) tuorein arvo, mikäli se on pienempi kuin pienin tähän mennessä läpikäydyistä. Muuttuja `i` on *askeltaja* ja `luku` on *tuoreimman säilyttäjä*.

```
(*1*)  program EtsiPienin (input,output);
(*2*)  var i, pienin, luku: integer;
(*3*)  begin
(*4*)     write('Anna 1. luku: '); readln(pienin);
(*5*)     for i := 2 to 10 do begin
(*6*)        write('Anna ',i,'. luku: ');
(*7*)        readln(luku);
(*8*)        if luku < pienin then pienin := luku
(*9*)     end;
(*10*)    writeln ('Pienin luku oli ', pienin)
(*11*) end.
```

Kuva 7: Sopivimman säilyttäjä -roolin esimerkkiohjelma Sajaniemen (2003) mukaan.

4.2.5 Kokooja (gatherer)

Kokooja on muuttuja, jonka arvo kerääntyy kaikista siihen mennessä läpi käydyistä arvoista. Esimerkkiohjelmassa (kuva 8) muuttuja `summa` on *kokooja*: siihen kootaan (rivillä 10) syötteiden kokonaissummaa. Muuttuja `lkm` on *askeltaja* ja `luku` on *tuoreimman säilyttäjä*.

```
(*1*)  program Keskiarvo (input,output);
(*2*)  var lkm: integer;
(*3*)      summa, luku: real;
(*4*)  begin
(*5*)     summa := 0;
(*6*)     lkm := 0;
(*7*)     repeat
(*8*)        write('Anna luku, -999 lopettaa: ');
(*9*)        readln(luku);
(*10*)       if luku <> -999 then summa := summa+luku;
(*11*)       if luku <> -999 then lkm := lkm+1
(*12*)    until luku = -999;
(*13*)    if lkm > 0 then writeln('Keskiarvo on  ', summa / lkm)
(*14*) end.
```

Kuva 8: Kokooja-roolin esimerkkiohjelma Sajaniemen (2003) mukaan.

4.2.6 Yksisuuntainen lippu (one-way flag)

Yksisuuntainen lippu on Boolen muuttuja, joka ei saa enää alkuperäistä arvoaan sen jälkeen, kun se on kerran muuttunut. Esimerkkiohjelma (kuva 9) tulostaa käyttäjän antamien lukujen summan ja ilmoittaa, oliko syötteiden joukossa yhtään negatiivista lukua. *Yksisuuntainen lippu* `neg` (rivillä 10) tarkkailee, esiintyykö syötteiden joukossa yhtään negatiivista arvoa, ja jos yksikin negatiivinen arvo löytyy, ei muuttuja enää palaa arvoon false. Muuttuja `luku` on *tuoreimman säilyttäjä* ja `summa` on *kokooja*.

Yksisuuntaista lippua voidaan käyttää myös esimerkiksi tarkkailemaan virheen esiintymistä syöttötiedoissa, jotta ohjelma huomaisi pyytää syötteitä uudelleen.

```
(*1*)  program Summa (input,output);
(*2*)  var luku, summa: integer;
(*3*)      neg: Boolean;
(*4*)  begin
(*5*)     summa := 0;
(*6*)   neg := false;
(*7*)     repeat
(*8*)        write('Anna luku, 0 lopettaa: '); readln(luku);
(*9*)        summa := summa + luku;
(*10*)       if luku < 0 then neg := true
(*11*)    until luku = 0;
(*12*)    writeln('Summa on ', summa);
(*13*)    if neg then writeln('Joukossa oli negatiivisia lukuja')
(*14*) end.
```

Kuva 9: Yksisuuntainen lippu -roolin esimerkkiohjelma Sajaniemen (2003) mukaan.

4.2.7 Seuraaja (follower)

Seuraaja on muuttuja, joka saa aina arvokseen jonkin tietyn toisen muuttujan vanhan arvon. Esimerkkiohjelma pyytää käyttäjältä 12 kokonaislukua ja kertoo lopuksi, mikä oli suurin kahden perättäisen syötetyn luvun ero. Muuttuja `edellinen` on *seuraaja*: se seuraa muuttujaa `nykyinen` (rivillä 8). Kuvassa 10 olevassa esimerkkiohjelmassa muuttuja `kuukausi` on *askeltaja*, `nykyinen` on *tuoreimman säilyttäjä* ja `suurinEro` on *sopivimman säilyttäjä*. *Seuraajia* käytetään paljon linkitettyjen tietorakenteiden yhteydessä osoittamaan käsiteltävää alkiota edeltänyttä alkiota.

```
(*1*)  program SuurinEro (input,output);
(*2*)  var kuukausi, nykyinen, edellinen, suurinEro: integer;
(*3*)  begin
(*4*)     write('Anna 1. arvo: '); readln(edellinen);
(*5*)     write('Anna 2. arvo: '); readln(nykyinen);
(*6*)     suurinEro := nykyinen - edellinen;
(*7*)     for kuukausi := 3 to 12 do begin
(*8*)        edellinen := nykyinen;
(*9*)        write('Anna ', kuukausi, '. arvo: ');
(*10*)       readln(nykyinen);
(*11*)       if nykyinen - edellinen > suurinEro
(*12*)          then suurinEro := nykyinen - edellinen
(*13*)    end;
(*14*)    writeln('Suurin ero oli ', suurinEro)
(*15*) end.
```

Kuva 10: Seuraaja-roolin esimerkkiohjelma Sajaniemen (2003) mukaan.

4.2.8 Tilapäissäilö (temporary)

Muuttuja on *tilapäissäilö*, jos sen arvoa tarvitaan aina vain hyvin lyhyen ajan. Kuvassa 11 oleva esimerkkiohjelma tulostaa syötteenään saamasta kahdesta luvusta ensin suuremman ja sitten pienemmän. Muuttujien sisällöt vaihdetaan keskenään käyttämällä *tilapäissäilönä* muuttujaa `tmp` (riveillä 7-9), jonka arvolla ei ole jatkossa merkitystä, vaikka ohjelma jatkuisi kuinka pitkään tahansa. *Tilapäissäilöä* käytetään usein *järjestelijän* kahden alkion keskinäisen paikan vaihtamiseen.

```
(*1*)  program Vaihda (input, output);
(*2*)  var luku1, luku2, tmp: integer;
(*3*)  begin
(*4*)     write('Anna luku: '); readln(luku1);
(*5*)     write('Anna toinen luku: '); readln(luku2);
(*6*)     if luku1 < luku2 then begin
(*7*)        tmp := luku1;
(*8*)        luku1 := luku2;
(*9*)        luku2 := tmp
(*10*)    end;
(*11*)    writeln('Luvuista suurempi on ', luku1,
(*12*)            ' ja pienempi on ', luku2)
(*13*) end.
```

Kuva 11: Tilapäissäilö-roolin esimerkkiohjelma Sajaniemen (2003) mukaan.

4.2.9 Muunnos (transformation)

Muunnos on muuttuja, joka saa uuden arvonsa jonkin tietyn ja muuttumattoman, muiden muuttujien arvoihin perustuvan laskentakaavan mukaan. Kuvassa 12 olevassa esimerkkiohjelmassa muuttujat `rad` ja `valit` ovat *muunnoksia*. Muuttuja `i` on *askeltaja*. Tässä ohjelmassa *muunnoksen* sijasta ohjelmassa voitaisiin käyttää pelkkää lauseketta. *Muunnosta* käytetään, koska sen ansiosta lähdekoodi on selkeämpää, tehokkaampaa tai sitten sitä tarvitaan väliaikaisena säilytyspaikkana.

4.2.10 Järjestelijä (organizer)

Järjestelijä on taulukko, jota käytetään siinä olevien tietojen uudelleen järjestämiseen sen jälkeen, kun taulukko on ensin alustettu joillakin arvoilla. Esimerkkiohjelma kuvassa 13 pyytää käyttäjältä merkki kerrallaan yhteensä kymmenen merkkiä *järjestelijään* `merkki`, kääntää niiden järjestyksen taulukossa ja lopuksi tulostaa merkit tässä

```
(*1*)   program sini;
(*2*)   const DegToRad = 2.0 * 3.14159 / 360.0;
(*3*)   var i: integer;
(*4*)       rad: real;
(*5*)       valit: integer;
(*6*)   begin
(*7*)     for i := 0 to 36 do begin
(*8*)         rad := i * 10.0 * DegToRad;
(*9*)         valit := trunc( (sin(rad)+1.0) * 40 );
(*10*)        writeln(' ' : valit, '*')
(*11*)    end
(*12*)  end.
```

Kuva 12: Muunnos-roolin esimerkkiohjelma Sajaniemeä (2003) mukaillen.

käännetyssä järjestyksessä. Muuttuja `tmp` on *tilapäissäilö* ja `i` on *askeltaja*. *Järjestelijää* voidaan käyttää taulukon lajitteluun tai muuhun uudelleenjärjestelyyn.

```
(*1*)  program Kaanna (input, output);
(*2*)  var merkki: array[1..10] of char;
(*3*)      tmp: char;
(*4*)      i: integer;
(*5*)  begin
(*6*)     for i := 1 to 10 do begin
(*7*)        write('Anna kirjain: '); readln(merkki[i])
(*8*)     end;
(*9*)     for i := 1 to 5 do begin
(*10*)       tmp := merkki[i];
(*11*)       merkki[i] := merkki[11-i];
(*12*)       merkki[11-i] := tmp
(*13*)    end;
(*14*)    for i:=1 to 10 do write(merkki[i]);
(*15*)    writeln
(*16*) end.
```

Kuva 13: Järjestelijä-roolin esimerkkiohjelma Sajaniemen (2003) mukaan.

4.2.11 Muu (other)

Muu on muuttuja, joka ei sovi mihinkään yllä mainituista rooleista. Sen ainoa tarkoitus on täydentää muuttujien roolitus niin kattavaksi, että jokainen muuttuja sopisi johonkin rooliin, toisin sanoen jäljelle jäävät 1% muuttujista noviisitason ohjelmissa. Noviisitason ohjelmissa *muuta* tarvitaan luokittelun apuna siis erittäin harvoin.

4.3 Muuttujien roolien ominaisuuksia

Seuraavaksi tarkastellaan hieman syvällisemmin muuttujien roolien ominaisuuksia ja erikoispiirteitä. Läpi käydään mm. asioita roolien määrittämiseen, roolien vaihtumi-

seen ja roolien ryhmittelyyn liittyen. Lisäksi tarkastellaan muuttujien rooleja vaativampien ohjelmointitekniikoiden yhteydessä.

4.3.1 Roolin määrittäminen

Muuttujien roolit eivät ole teknisiä, vaan kognitiivisia käsitteitä, eli kaksi eri ihmistä voi pitää saman ohjelman samaa muuttujaa eri roolin edustajana, ja molemmat voivat olla oikeassa. Asian voi helposti todentaa seuraavan esimerkin avulla. Fibonaccin luvut etenevät seuraavasti: 1, 1, 2, 3, 5, 8, 13, 21, ... eli jokainen numero on kahden edeltävän luvun summa. Matemaatikolle Fibonaccin luvut voivat olla samanlainen luonnollinen asia kuin matematiikkaa harrastamattomille lukusuora 1, 2, 3, 4, 5, ... olisi. Jos siis noviisi, joka ei ole koskaan kuullutkaan Fibonaccin luvuista, oppii tavan laskea niitä, jokainen uusi numero numerosarjassa on yllätys. Aiemmasta tietämyksestä johtuvien erojen takia matemaatikko saattaa siis pitää Fibonaccin luvun sisältävää muuttujaa *askeltajana*, koska muuttuja askeltaa läpi tiettyjä, ennalta määrättyjä perättäisiä arvoja säännönmukaisesti, kun taas noviisin mielestä kyseessä olisi *kokooja*, koska kyseisen muuttujan arvo kerääntyy kaikista siihen mennessä läpi käydyistä arvoista (Kuittinen & Sajaniemi, 2004).

Vaikka näkemykset muuttujan roolista saattavat yllä mainitun kaltaisissa tapauksissa olla ristiriitaisia, esiintyy roolijaossa kuitenkin tiettyjä säännönmukaisuuksia. Analysoituaan kolmen noviisitason ohjelmointikirjan esimerkkiohjelmia Sajaniemi (2002) on havainnut, että 84 % kyseisten ohjelmien muuttujista kuuluivat johonkin seuraavista kolmesta roolista: *kiintoarvo*, *askeltaja* sekä *tuoreimman säilyttäjä*.

4.3.2 Muuttujan roolin vaihtuminen

Muuttujan rooli ei ole kiinteästi sidoksissa johonkin muuttujaan koko ohjelman suorituksen ajan, vaan muuttujan rooli voi myös vaihtua. Usein tämä muutos tapahtuu kahden silmukan välillä. Seuraavassa esimerkkiohjelmassa (kuva 14) muuttuja `tieto` on ensimmäisessä silmukassa rooliltaan *tuoreimman säilyttäjä*, mutta toisessa silmukassa sitä käytetään *askeltajana*.

Roolivaihdoksia on olemassa kahta perustyyppiä. *Oikeasta roolinvaihdoksesta* puhutaan silloin, kun muuttujan edellisen roolin viimeisintä arvoa käytetään seuraavan roo-

```
(*1*)   program tuplaus;
(*2*)   var tieto, arvo: integer;
(*3*)   begin
(*4*)     repeat
(*5*)       write('Syota haluttu kierrosmaara: '); readln(tieto)
(*6*)     until tieto > 0;
(*7*)     while tieto > 0 do begin
(*8*)       write('Syota jokin luku: '); readln(arvo);
(*9*)       writeln('Luku ', arvo, ' kerrottuna kahdella on ', 2*arvo);
(*10*)      tieto := tieto - 1
(*11*)    end
(*12*)  end.
```

Kuva 14: Esimerkki roolien vaihtumisesta Sajaniemeä (2003) mukaillen.

lin alkuarvona. *Satunnaisesta roolinvaihdoksesta* taas on kyse silloin, kun muuttuja roolinvaihdoksen yhteydessä alustetaan aivan uudella arvolla (Sajaniemi, 2004a). Toisinaan tämä voi merkitä huonohkoa ohjelmointitapaa muuttujien käytön suhteen, kun saman muuttujan käytöllä toisessa tarkoituksessa yritetään ”säästää” muistia.

4.3.3 Muuttujien roolien erikoistapauksia

Yksinkertaisilla tietotyypeillä (kokonaisluku, reaaliluku jne.) voi olla ainoastaan yksi rooli samanaikaisesti. Taulukoilla kokonaisuudessaan on yksi rooli, joka perustuu siihen, mitä rooleja sen alkioilla on. Toisin sanoen, mikäli taulukon kaikki alkiot ovat rooliltaan *tuoreimman säilyttäjiä*, niin taulukon kokonaisuudessaan ajatellaan olevan *tuoreimman säilyttäjä*. Ainoa poikkeus tästä säännöstä on *järjestelijän* rooli, joka kuvaa taulukkoa yhtenä kokonaisuutena.

Muuttujien roolitusta voidaan käyttää vaativampienkin ohjelmointitekniikoiden yhteydessä kuin mitä ohjelmoinnin alkeiskursseilla käydään läpi. Esimerkiksi osoittimet voidaan roolittaa: osoittimen rooli kuvaa sitä, miten osoitin käyttäytyy, eikä sitä, miten muuttuja, johon osoitetaan, käyttäytyy. Monimutkaisempien tietorakenteiden (tietueiden) tapauksissa roolit pitää jakaa jokaiselle tietueen kentälle erikseen, koska eri kentillä voi olla hyvinkin erilaisia tehtäviä ohjelmissa. Tietue voi koostua esimerkiksi kahdesta osasta: elementin identifioivasta kentästä, joka on *kiintoarvo* sekä toisesta kentästä, johon kootaan elementtiin liittyvää tietoa, rooliltaan vaikkapa *kokooja*.

Pascal-kielen proseduurien ja funktioiden muodollisiin arvoparametreihin sekä paikallisiin muuttujiin voidaan suhtautua kuten normaaleihin muuttujiin. Muuttujaparametreihin proseduurien ja funktioiden sisällä tehdyt muutokset taas vaikuttavat suoraan

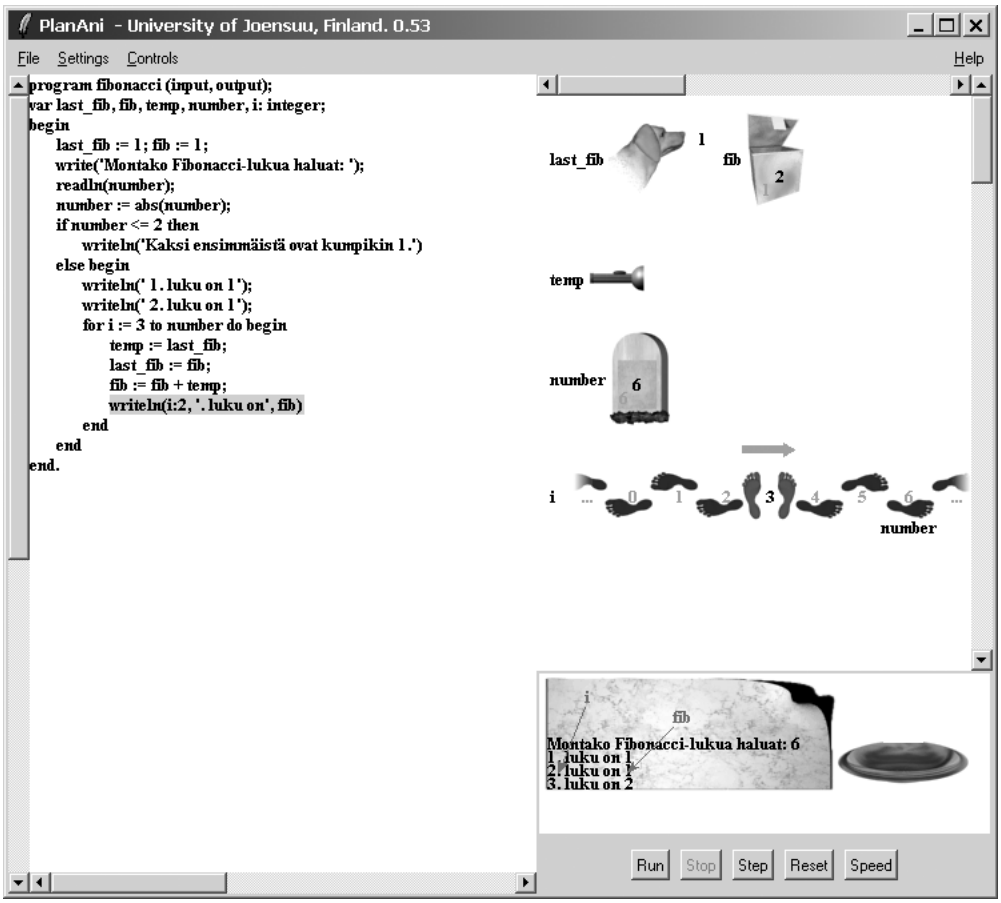
viittaamiinsa muuttujien arvoihin, joten niitä pitäisi kohdella samoina muuttujina kuin mihin ne viittaavat.

Tiedostomuuttujat käyttäytyvät muista muuttujista poikkeavalla tavalla ja siksi ne jätetään roolitarkastelun ulkopuolelle. Tiedostomuuttujat itsessään ovat tavallisesti kiintoarvoja, eli ne liittyvät samaan tiedostoon koko ohjelman suorituksen ajan. Tiedostojen sisällöt ovat erityisasemassa. Vaikka yksittäiset tiedot tai tietueet, jotka on luettu tiedostosta, voidaankin tallentaa johonkin muuttujaan, eivät itse tiedostossa olevat tietueet lukutapahtuman yhteydessä muutu. Ne ovat siis ikään kuin kiintoarvoja, mutta ne eivät oikeasti ole muuttujia, vaan sijaitsevat "ohjelman ulkopuolella" (Sajaniemi, 2004b).

4.4 Muuttujien roolien visualisointi ja PlanAni-animaattori

PlanAni on Joensuun yliopiston tietojenkäsittelytieteen laitoksella kehitetty muuttujien rooleihin pohjautuva animointiohjelma. Sen avulla voidaan visualisoida esimerkkiohjelmia ja niiden toimintaa ohjelmointikielestä riippumattomalla tasolla sekä konkretisoida samalla muuttujien rooleja. PlanAnin tarkoituksena on toimia apuvälineenä ohjelmoinnin alkeiden opetuksessa. Perinteisesti ohjelmien visualisointijärjestelmät ovat kuvanneet ohjelman tai esimerkiksi yksittäisen algoritmin toimintaa pelkästään ohjelmointikielen tasolla. Petren & Blackwellin (1999) mukaan visualisoinnin ei pitäisi kuitenkaan tapahtua ohjelmointikielen tasolla, sillä ohjelmointikielen rakenteisiin liittyvä tietämys ei ole niin tärkeää kuin korkeamman tason ohjelmointirakenteiden ja ideoiden selventäminen. Jos visualisointi siis ohjelmointikielen sijasta keskittyykin muuttujien roolitukseen, voidaan visualisoinnissa tarkastella enemmän ohjelmointitasoa ja täten antaa enemmän tietoa myös itse ohjelman toiminnasta etenkin noviiseille.

Kuvassa 15 on ruutukaappaus PlanAni-animaattorin toiminnasta. Vasemmalla puolella näkyy ajettava Pascal-kielinen esimerkkiohjelma, ja oikealla ylhäällä näkyy muuttujien roolien kuvia kyseiseen ohjelmaan liittyen. Oikealla alhaalla sijaitsee paperi ja lautanen, joiden avulla ilmaistaan tulosteet ja syötteet esimerkkiohjelmaan liittyen.



Kuva 15: Ruutukaappaus PlanAni-animaattorista.

4.4.1 Roolien kuvat ja niihin liittyvät animoinnit

PlanAni-animaattorissa jokaisella muuttujan roolilla on oma kuvallinen vastineensa. Muuttujan roolin kuva pyrkii antamaan vihjeitä siitä, millaisia ominaisuuksia kyseisen roolin omaavalla muuttujalla on, miten se suhtautuu muihin muuttujiin, niiden arvoihin ja yleensä ohjelman suorituksen aikaisiin tapahtumiin.

Esimerkiksi *kiintoarvoa* symboloi kivi (kuva 16). Tarkoituksena on antaa vaikutelma sen tyyppisestä muuttujasta, jonka ”kiveen hakattua arvoa on hankala muuttaa”. *Sopivimman säilyttäjä* (kuva 17) muodostuu kahdesta kukkasesta, joista ensimmäisessä, harmaalla tekstillä varustetussa, säilytetään entistä, ja toisessa, selkeämmin erottuvassa, nykyistä kyseisen muuttujan arvoa. *Tuoreimman säilyttäjässä* (kuva 18) näkyy myöskin sekä muuttujan edellinen ja nykyinen arvo kahden eri neliön sisällä. *Askeltajan* kuvana on jalanjälkiä (kuva 19), joiden yhteydessä näkyy arvoja, joita kyseisen roolin omaavalla muuttujalla on ollut tai tulee olemaan. Jalanjälkien ja arvojen lisäk-

si kuvassa näkyy myös nuoli, joka kertoo, mihin suuntaan ollaan menossa. *Seuraajan* kuvana on koiran pää, jonka vieressä on muuttuja, jonka arvoa se seuraa (kuva 20). *Kokoojaa* esittää laatikko, jonka yhteydessä näkyy sekä muuttujan edellinen että nykyinen arvo (kuva 21). *Yksisuuntaista lippua* (kuva 22) esittää hehkulamppu, joka rikkoutuu, kun lipun arvo ei enää ole tosi. *Tilapäissäilöä* (kuva 23) kuvaa taskulamppu, joka on päällä niin kauan kuin kyseisen tyyppisen muuttujan arvoa käytetään. Kun arvoa ei enää tarvita, taskulamppu sammuu ja arvo katoaa. Viimeiseksi mainitaan vielä *järjestelijä* (kuva 24), jonka alkiot ovat samanlaisia kiviä kuin *kiintoarvolla*, mutta kivet on varusteltu pienillä pyörillä. Tarkoituksena on kuvata, että tarvittaessa kiviä voidaan helposti siirrellä. Muita taulukoita kuvataan vastaavalla määrällä muita roolikuvia, eli jos taulukossa on vaikkapa viisi *kokoojaa*, kyseistä taulukkoa kuvataan viiden laatikon avulla. Koska tätä kirjoitettaessa *muunnos*-roolilla ei ollut PlanAnissa kuvaa, ei sitä esitetä tässä yhteydessä.

Kuva 16: Kiintoarvo

Kuva 17: Sopivimman säilyttäjä

Kuva 18: Tuoreimman säilyttäjä

Kuva 19: Askeltaja

Kuva 20: Seuraaja

Kuva 21: Kokooja

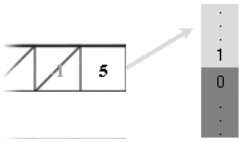
Kuva 22: Yksisuuntainen lippu

Kuva 23: Tilapäissäilö

Rooleja vastaavien kuvien ymmärrettävyys on luonnollisesti jossakin määrin riippuvaista vallitsevasta kulttuurista. Tarvittaessa rooleihin liittyvät kuvat voidaan vaihtaa PlanAnin lähdekoodeja muokkaamatta muihin kulttuureihin paremmin sopivaksi.

Kuva 24: Järjestelijä

PlanAnissa olevissa animaatioissa käytetään siis apuna muuttujien roolien kuvia. Koska eri rooleihin liittyvät samat operaatiot, kuten muuttujien arvojen vertailut, poikkeavat toisistaan merkitykseltään muuttujien roolien ja ohjelman sisäisen toimintalogiikan kannalta, näkyy tämä eroavaisuus käytetyissä animoinneissakin. Jos muuttujat ovat rooleiltaan vaikkapa *tuoreimman säilyttäjä* ja *askeltaja* ja vertailtava lauseke tyyppiä ”onko muuttuja x > 0”, niin animoinnit poikkeavat toisistaan voimakkaasti. *Tuoreimman säilyttäjän* yhteydessä tarkistetaan, onko arvo sallitulla alueella. Animoinnin yhteydessä tämä näkyy roolikuvan viereen ilmestyvänä palikkana, jossa on vihreällä pohjalla sallittujen arvojen alue ja punaisella pohjalla kiellettyjen arvojen alue. Jos muuttujan arvo on sallitulla alueella, roolikuvasta lähtee vihreän värinen nuoli, joka osoittaa vihreälle alueelle (kuva 25). Jos puolestaan muuttuja on kielletyllä alueella, on nuoli väriltään punainen, ja se osoittaa punaiselle alueelle. Kun kyseessä on muuttuja, joka on rooliltaan *askeltaja*, on sallitut ja kielletyt arvot jälleen vertailun yhteydessä värjätty vastaavalla tavalla punaisella ja vihreällä, mutta ne kuuluvat jo osana roolia vastaavaan kuvaan (kuva 26).



Kuva 25: Tuoreimman säilyttäjän vertailu.

Kuva 26: Askeltajan vertailu.

Käyttäjä voi nähdä vertailun lopputuloksen paitsi roolikuvasta, jossa muuttujan nykyinen arvo on vihreänä, jos se on sallitulla alueella, myös pop-up-ikkunasta, jossa vielä todetaan kertaalleen vertailun lopputulos. Roolikuvassa olevat arvot muuttuvat vertailun jälkeen takaisin mustaksi ja harmaaksi.

Muuttujien alustaminen jollakin arvolla näkyy eri tavoin eri roolien kohdalla. Mikäli muuttuja on *tuoreimman säilyttäjä* ja sillä on jo jokin arvo, siirtyy muuttujan vanha arvo vasemmalle päin toisen neliön sisään ja muuttuu väriltään harmaaksi. Uusi arvo puolestaan siirretään käyttäjän antaman syötteen perusteella oikeasta alakulmasta

kyseisen muuttujan uudeksi arvoksi. *Askeltajan* yhteydessä alustus tapahtuu täysin eri tavalla, eli roolikuvassa olevat luvut liukuvat sopivasti siten, että uusi arvo jää keskellä olevan jalanjälkien väliin. Animoinnilla halutaan korostaa sitä, että *askeltajan* yhteydessä tuleva arvo on jo tiedossa ja arvon muuttuminen noudattaa tiettyä kaavaa.

Muuttujan roolien vaihtumiset on animoitu myöskin toisistaan poikkeavasti riippuen siitä, onko kyseessä *aito* vai *satunnainen roolinvaihdos*. Aidot roolinvaihdokset on animoitu siten, että roolikuva muuttuu vähitellen vaihe vaiheelta toiseksi kuvaksi, kun taas satunnaisen roolin vaihtumisen yhteydessä roolikuvat vaihtuvat siten, että vanha kuva häviää ja uusi tulee tilalle. Tällä halutaan korostaa vanhan ja uuden roolin välisen suhteen vahvuutta.

4.4.2 PlanAnin toiminta

Kuten luvussa kaksi jo todettiin, ihminen ei pysty keskittymään kahteen tai useampaan paljon tarkkaavaisuutta vaativaan asiaan kerralla. Tämä ilmenee PlanAnin yhteydessä siten, että noviisiohjelmoijan olisi vaikea seurata käyttöliittymän eri osissa tapahtuvia asioita, ellei niitä rytmitettäisi jollakin tavalla. Jotta tarkkaavaisuuden kohdetta ei tarvitsisi vaihdella jatkuvasti ohjelmakoodin ja muuttujien välillä, on otettu käyttöön pop-up-ikkunat, joissa selitetään, mitä ohjelmassa kullakin hetkellä on tekeillä. Niissä kerrotaan tapahtumista, kuten muuttujan luominen ja mitä roolia se edustaa, roolien muutokset, erilaiset operaatiot sekä kontrollirakenteet. Päästäkseen ikkunasta eroon, täytyy käyttäjän joko klikata sitä hiiren vasemmalla painikkeella tai painaa enteriä tai välilyöntiä. Lopullisessa PlanAnin versiossa on tarkoitus käyttää pop-up-ikkunoiden sijasta puhesyntetisaattoria.

Pop-up-ikkunoiden kuittausten lisäksi käyttäjältä pyydetään ohjelman suorituksen etenemisen mukaisesti syötteitä. Käyttöliittymän eri osien välisiä yhteyksiä pyritään lisäksi tuomaan esiin käyttämällä esimerkiksi muuttujan luomisen yhteydessä nuolta lähdekoodista muuttujan roolia kohti. Turhien yksityiskohtien välttämiseksi lausekkeiden vertailuja ei animoida, vaan näytetään ainoastaan vertailujen lopputulos ja sen seurannaisvaikutukset. Käyttäjät voivat säädellä animointinopeutta ja säädellä ohjelmassa käytettäviä fontteja. Animointi voidaan lopettaa kesken ohjelman, mutta animoinnin seuraaminen takaperin ei kuitenkaan ole mahdollista.

4.5 Muuttujien roolien opettaminen ja käyttötarkoitukset

Muuttujien roolien ei ole tarkoitus kasvattaa ohjelmointikursseilla opittavien asioiden määrää, vaan niiden pitää palvella kokonaisuutta siten, että ne auttavat ohjelmointiin liittyvien käsitteiden sisäistämisessä. Sen sijaan roolien tulee toimia työvälineenä, jonka avulla noviisit voivat paitsi aiempaa helpommin ymmärtää muuttujien elinkaaria, myös suunnitella ohjelmia ja työstää niitä mielessään uudella tavalla. Seuraavassa on kuvaus siitä, kuinka rooleihin liittyvää tietämystä voidaan opettaa, kuinka rooleja voidaan käyttää apuna ohjelmointistrategioiden opettamisessa sekä mitä muita käyttötarkoituksia rooleilla voisi olla.

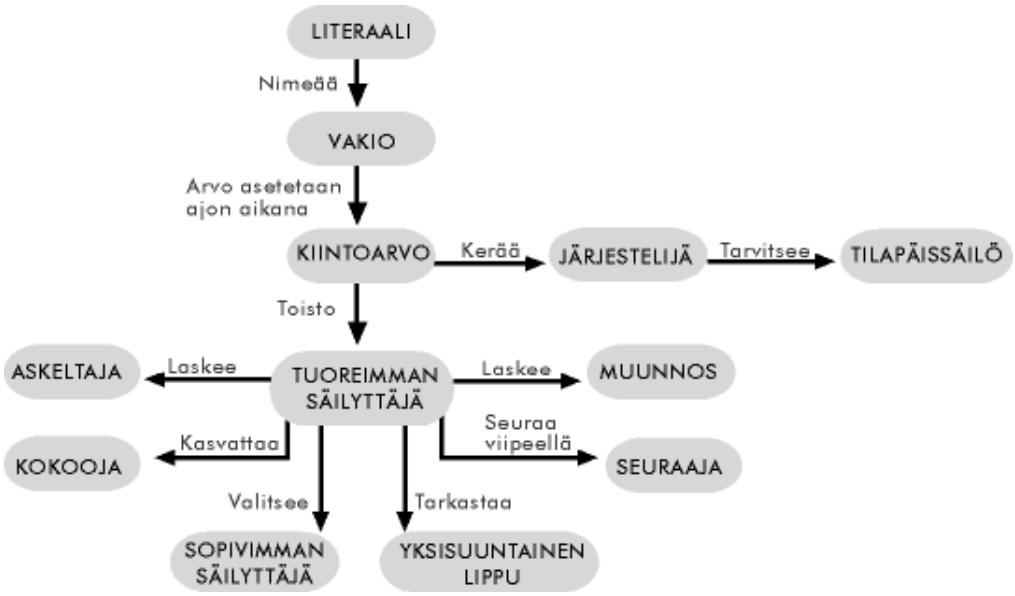
4.5.1 Roolien esittelystä

Kuten luvussa kaksi kerrottiin, konstruktivistisen oppimiskäsityksen mukaan tietämystä jostakin asiasta tulisi kasvattaa aktiivisesti vähitellen aiemmin opittuun tietoon pohjautuen. Oppimisessa on tärkeätä painottaa olennaisen ymmärtämistä mekaanisen toistamisen sijasta. Oppijan taustasta riippuen oppijat sitten tulkitsevat ja ymmärtävät opetettavaa asiaa eri tavoin (Ben-Ari, 2001).

Opetettaessa roolit voidaan esitellä sitä mukaa kuin ne tulevat esille ensimmäistä kertaa luennolla jonkin esimerkkiohjelman yhteydessä. On olennaista, että uuden roolin esittely pohjautuu aiemmin opittuun tietoon ja erovaisuudet roolien välillä tuodaan selkeästi esille. Kuvassa 27 on esiteltynä muuttujien roolien väliset suhteet.

Tämä määrittelee käytännössä, missä järjestyksessä muuttujien rooleja voidaan opettaa, eli loogisinta on lähteä liikkeelle literaaleista. Kuvassa ”literaali” ja ”vakio” ovat ohjelmointikielen rakenteita, muut solmukohdat ovat muuttujien rooleja. Ylimmässä nuolessa oleva teksti kuvaa literaalin ja vakion suhdetta, ja toiseksi ylin vakion ja *kiintoarvon* välistä suhdetta. Muut nuolet kuvaavat roolien välisiä suhteita.

Ohjelmoinnin alkeiden kursseilla ensimmäinen esitettävä ohjelma sisältää useimmiten literaaleja, kuten vaikkapa ”Terve, maailma!”-viestin. Vakiot voidaan esitellä seuraavana antamalla literaalille jokin nimi. Seuraavassa vaiheessa esitellään useimmiten muuttujat, ja tässä yhteydessä voidaan ottaa esiin myös muuttujien roolit. On suositeltavaa, että rooleista esitellään ensimmäisenä *kiintoarvo*, jotta se voidaan nivoa yhteen vakioon liittyen. Tällöin voidaan selittää, että *kiintoarvon* arvoa voidaan muuttaa ohjel-



Kuva 27: Roolien esittelyjärjestys Kuittista & Sajaniemeä (2004) mukaillen.

man suorituksen aikana esimerkiksi käyttäjän antamalla syötteellä, toisin kuin vakion kohdalla, jossa sen arvo on määritetty kyseisen ohjelman lähdekoodissa.

Seuraava esiteltävä rooli voisi olla joko *järjestelijä* tai *tuoreimman säilyttäjä*, mutta ainoa oikeasti järkevä vaihtoehto on *tuoreimman säilyttäjä*, koska *järjestelijän* esittelyssä tarvitaan apuna taulukko-käsitettä. Kun *tuoreimman säilyttäjä* on esitelty vertaamalla sitä kiintoarvoon vaikkapa ”*kiintoarvona*, joka voi saada arvon monta kertaa ohjelman suorituksen aikana”, voidaan samaa periaatetta noudattaen esittää loputkin roolit, yhdistäen ne samalla aiemmin esiteltyihin rooleihin ja korostamalla roolien toisistaan eroavia piirteitä.

Roolien esittelyn tulisi sisältää paitsi roolin määrittelyn, myös konkreettisia esimerkkiohjelmia kyseisen roolin käyttämisestä ja lisämainintoja kyseisen roolin käyttötarkoituksiin liittyen. Tämä voidaan järjestää paitsi käymällä rooleja läpi luentojen yhteydessä, myös jakamalla kurssin osallistujille materiaalia, joka sisältää roolien määritykset sekä rooleihin liittyviä esimerkkiohjelmia. Lisäksi opettaja voi ottaa vielä erikseen esille joitakin erikoistapauksia, jotka kuuluvat roolista annetun määritelmän piiriin.

Koska ohjelmointikursseilla ohjelmointikielen rakenteita esitellään tyypillisesti esimerkkiohjelmien avulla, voidaan vaikkapa Pascal-kurssin yhteydessä forsilmukkarakennetta esiteltäessä tuoda esiin roolien näkökulma kertomalla *askeltaja*-roolista. Liittämällä tällä tavoin rooleihin liittyvä tieto jo mahdollisesti aiemmin laa-

dittuun opetusmateriaaliin, ei roolien sisällyttäminen ohjelmointikurssiin vaadi paljon ylimääräistä työtä.

4.5.2 Rooleihin liittyvän tietämyksen syventäminen

Luvussa kaksi kerrottiin, että tiedon tallentumista muistiin voidaan tehostaa, mikäli asiaan, joka halutaan muistaa, liitetään jotakin itselle merkityksellistä lisätietoa. Pelkkä uuden asian ulkoa opettelu tai merkityksettömän tiedon liittäminen kyseiseen asiaan ei johda hyvään lopputulokseen, vaan olennaisinta on se, miten oppija mielessään prosessoi tietoa sitä opiskellessaan (Anderson, 1985). Käytännön opetustyössä tämä tarkoittaa, että jos luennoitsija esimerkkiohjelmaa läpi käydessään vain kertoo siinä olevien muuttujien roolien määritelmät, ei lopputulos opiskelijan muistamisen kannalta todennäköisesti ole kovin hyvä. Olisi sen sijaan parempi, jos luennoitsija ohjelman yhteydessä kertoisi, kuinka jokin kyseisessä ohjelmassa oleva muuttuja ilmentää tietyn roolin käyttäytymistä. Näin opiskelijalle jää suuremmalla todennäköisyydellä mieleen merkityksellinen esimerkki kyseiseen rooliin liittyen.

Jotta opiskelijan rooleihin liittyvää muistamista voitaisiin syventää edelleen, tulee roolien nimiä esitellä toistuvasti uusissa, merkityksellisissä asiayhteyksissä. Muuttujan roolin ja sen tarkoituksen kyseisessä ohjelmassa voi tuoda esille vaikkapa ohjelman lähdekoodiin liittyvissä kommenteissa. Kommenttien tarkoituksena on siis paitsi toistaa rooleihin liittyvää tietoa, myös antaa uutta tietoa kyseisen roolin mahdollisista käyttötarkoituksista eri ohjelmissa (*sopivimman säilyttäjää* voidaan käyttää säilyttämään annetuista syötteistä pienin jne.).

Rooleihin liittyvää muistamista ja tietämystä voidaan parantaa myös keskustelemalla opiskelijoiden kanssa siitä, minkä roolin he antaisivat tietylle muuttujalle jossakin ohjelmassa. Kuten aiemmin tässä luvussa mainittiin, roolit ovat kognitiivinen käsite eikä yhtä ainutta ja oikeaa roolin määritystä tietylle muuttujalle välttämättä ole olemassa. Jos opiskelijat siis pohtivat eri kanteilta, kuinka jokin muuttuja sopii yhteen eri roolien määritelmien kanssa, saadaan jälleen aikaan merkityksellistä lisäinformaatiota, joka helpottaa muistamista ja asian ymmärtämistä.

Kullekin muuttujien roolille voidaan antaa jokin tietty visuaalinen ilmentymä, joka havainnollistaa kunkin roolin keskeisiä ominaisuuksia. Näin on tehty aiemmin tässä luvussa mainitussa PlanAni-animaattorissa, jota voidaan hyödyntää roolien opettamises-

sa. Kun jokin rooleista (eli roolin kuva) esiintyy animaation yhteydessä ensimmäistä kertaa, on opettajan tehtävänä kertoa, mistä kuvassa on kysymys ja kuinka kyseisessä kuvassa pyritään korostamaan kyseisen roolien keskeisimpiä ominaisuuksia. Koska ohjelmoinnin alkeiskurssille osallistuvilla ohjelmointiin liittyvä tietämys on vielä lapsenkengissä, on syytä mainita esimerkiksi *askeltaja*-roolin kuvaan liittyen, että siinä näkyvät monet eri muuttujan arvot eivät ole oikeasti olemassa eikä niihin pääse käsiksi jonkin ”korkeamman tietämyksen” avulla, jota ohjelmoinnissa pidemmälle edenneillä on hallussaan. Tämä seikka on tuotava riittävän monta kertaa esille animointeja katsottaessa, jotta kaikille käy varmasti selväksi, että muut kuin muuttujan sen hetkinen arvo ovat kuvassa näkyvillä ainoastaan visualisointitarkoituksessa.

PlanAni selventää roolien ominaisuuksia paitsi rooleihin liittyvien kuvien avulla, myös erilaisten vertailujen yhteydessä, jotka roolista riippuen animoidaan eri tavoin. Tämä johtuu siitä, että vertailut, jotka ohjelmointikielen tasolla näyttävät samoilta, omaavat erilaisen merkityksen roolien kannalta. Tästä on kerrottu aiemmin alakohdassa 4.4.1 "Roolien kuvat ja niihin liittyvät animoinnit".

4.5.3 Roolit ja ohjelmointistrategioiden kehittäminen

Ohjelmointi on pohjimmiltaan ongelmanratkaisua, jossa tarvitaan ongelmanratkaisuoperaattoreita. Noviiseilla on usein suuria ongelmia, kun heitä pyydetään laatimaan jonkin ongelman ratkaiseva ohjelma. Ongelmat saattavat ilmetä eri tavoin, esimerkiksi: noviisi ei tiedä, pitäisikö ohjelman laatiminen aloittaa jostakin muuttujasta vaiko silmukasta. Tämä viittaa puuttuviin ohjelmointistrategioihin, eli noviiseilla ei ole käytettävissä tarvittavia ongelmanratkaisuoperaattoreita. Noviiseja voidaan opettaa tällaisissa tilanteissa käyttämään ongelmanratkaisussa muuttujien rooleja. Seuraavassa esimerkkikuvaus siitä, miten ongelmanratkaisu voisi edetä muuttujien rooleja käyttämällä.

Oletetaan, että noviisille annetaan tehtäväksi laatia ohjelma, joka muuntaa lämpötilojen arvoja eri lämpöasteikkojen (Celsius, Kelvin, Fahrenheit) välillä. Ohjelman suunnittelu voidaan aloittaa valitsemalla tehtävänantoon sopivat muuttujat. Tässä ohjelmassa käyttäjän antamat syötteet koostuvat lämpötilasta ja valitusta lämpöasteikosta, jossa kyseinen lämpötila on annettu. Normaalisti syötteet tallennetaan joko *kiintoarvoihin* (mikäli syötettä kysytään ainoastaan kerran) tai *tuoreimman säilyttäjään* (mikäli syötettä kysytään useamman kerran silmukassa). Syötteiden tallentamiseen tarvitaan siis

kaksi jompaakumpaa edellä mainittua roolia edustavaa muuttujaa. Annetaan kyseisille muuttujille tunnukset asteet ja asteikko. Oletetaan, että käyttäjä antaa tarvittavat syötteet ainoastaan kerran, jolloin molemmat syötteet ovat *kiintoarvoja*. Ohjelman tulisi tuottaa lämpötilat eri asteikkoja käyttäen. Tämän suorittamiseen vaaditaan *muunnos*-roolia. Suoritettavat muunnokset tehdään alkuperäistä asteet-muuttujan arvoa muokkaamalla, joten jokaista muunnosta kohden tarvitaan ilmeisesti yksi *muunnos*-rooli. Tähän mennessä aikaansaatu ohjelma näyttää seuraavalta:

```
(*1*)   program ?nimi? (input, output);
(*2*)   var asteet:        ?tyyppi? (* kiintoarvo: syöte on astemäärä        *)
(*3*)       asteikko:      ?tyyppi? (* kiintoarvo: syötteen lämpöasteikko    *)
(*4*)       astCelsius:    ?tyyppi? (* muunnos: lämpötila Celsius-asteina    *)
(*5*)       astKelvin:     ?tyyppi? (* muunnos: lämpötila Kelvin-asteina     *)
(*6*)       astFahrenheit: ?tyyppi? (* muunnos: lämpötila Fahrenheit-asteina *)
(*7*)   begin
(*8*)     lue asteet
(*9*)     lue asteikko
(*10*)    astCelsius:= ?laskutoimitus, joka perustuu asteet-muuttujan arvoon?
(*11*)    astKelvin:=  ?laskutoimitus, joka perustuu asteet-muuttujan arvoon?
(*12*)    astFahrenheit:= ?laskutoimitus, joka perustuu asteet-muuttujan arvoon?
(*13*)    tulosta saadut arvot
(*14*)  end.
```

Kuva 28: Lämpötilojen muuntaminen Kuittista & Sajaniemeä (2003) mukaillen.

Ohjelmassa käytetään kahta roolia niille tyypillisellä tavalla. Muuttujien rooleja käyttämällä ei tiedetä, mitä kysymysmerkkien väliin pitäisi laittaa, mutta nyt ollaan jo selvillä siitä, mitä kyseisissä kohdissa voisi olla. Suurin hyöty roolien käyttämisessä onkin siinä, että noviisilla on ongelmaa ratkaistessaan jonkinlainen lähtökohta, joka auttaa alkuun pääsemisessä.

Roolien hyödyntäminen ongelmanratkaisun yhteydessä ei takaa millään tavalla sitä, että kyseisessä ongelmassa päädytään optimiratkaisuun. Sen sijaan roolien avulla pystytään laatimaan alustava suunnitelma sille, miten ohjelma voisi toimia, joten rooleihin liittyvän tietämyksen avulla noviiseille voidaan opettaa perusohjelmointistrategioita.

4.5.4 Muuttujien roolien käyttötarkoituksista

Muuttujien rooleja voidaan käyttää apuna ohjelmoinnin alkeiden opettamisessa tarjoamalla noviiseille eksperttien piilevässä muodossa olevaa tietämystä. Edellä mainittujen ohjelmointistrategioiden opetuksen lisäksi muuttujien rooleja voidaan käyttää noviisien ohjelmissaan tekemien virheiden selittämisessä ja analysoimisessa, sillä jos muut-

tujia käytetään roolituksesta poikkeavalla tavalla, voi se olla merkki virheestä ohjelmassa. Roolien avulla voidaan myös visualisoida ohjelmien toimintaa ohjelmointikielestä riippumattomalla tasolla, ja tuoda esille muutakin kuin ohjelmointikieleen liittyvää tietoutta. Roolien käyttö ei ole kuitenkaan rajoittunut pelkästään noviisien opettamiseen, sillä yhtä hyvin niitä voivat hyödyntää myös pidemmälle edenneet ekspertit esimerkiksi analysoidessaan suurempikokoisten ohjelmien toimintaa.

5 Muuttujien rooleihin liittyvä empiirinen koe

Kolmannessa luvussa esiteltiin mm. erilaisia ohjelmointiin liittyviä tutkimuksia sekä kerrottiin, mitä ohjelmoinnista, sen oppimisesta ja noviisien vaikeuksista tällä hetkellä tiedetään. Luvussa neljä esiteltiin uusi ohjelmointiin liittyvä käsite, muuttujien roolit sekä PlanAni-animaattori, joita voidaan hyödyntää ohjelmoinnin alkeiden opettamisessa. Lisäksi läpi käytiin muuttujien roolien opettamiseen liittyviä asioita.

Tässä luvussa esitellään empiirinen koe, jossa testattiin muuttujien roolien toimivuutta käytännön opetustyössä. Ellei muuta lähdettä ole mainittu, on ”Tulokset”-kohtaan asti lähteenä ollut Kuittinen & Sajaniemi (2003) ja tästä luvun loppuun asti Sajaniemi & Kuittinen (in press).

Empiirisen kokeen tarkoituksena oli testata hypoteesia, jonka mukaan muuttujien roolien esitteleminen ohjelmoinnin alkeiden opetuksessa helpottaa ohjelmoinnin oppimista. Lisäksi tutkittiin roolipohjaisen animoinnin vaikutuksia oppimistuloksiin. Koe suoritettiin syksyllä 2002 Joensuun yliopiston tietojenkäsittelytieteen laitoksella järjestetyn Ohjelmoinnin alkeet -kurssin yhteydessä. Kurssin kesto oli viisi viikkoa, ja joka viikko oli neljä tuntia luentoja ja kaksi tuntia harjoituksia, eli yhteensä luentoja oli 20 tuntia ja harjoituksia 10 tuntia.

Seuraavassa on kuvaus kokeen suorittamisen eri vaiheista, joka aloitetaan koehenkilöiden ryhmäjaon suorittamisesta, minkä jälkeen vuorossa ovat käytännön opetustyöhön liittyvät asiat kokeen yhteydessä, protokollatehtävät, kurssiin liittyvä lopputentti ja tenttitulosten analysointi paitsi arvosanojen myös kokeeseen liittyvän hypoteesin kannalta.

5.1 Koehenkilöiden ryhmäjako

Koehenkilöinä toimivat yllä mainitun Ohjelmoinnin alkeet -kurssin opiskelijat. Koe suoritettiin between-subject-tyyppisenä. Koehenkilöt jaettiin kolmeen ryhmään, joista jokaisella ryhmällä kokeiltiin hieman toisistaan poikkeavia opetustapoja. Erottavana tekijä kokeeseen kuuluvien ryhmien välillä (between-subject factor) oli järjestetyn opetuksen sisältö, eli esitettiinkö koehenkilöille muuttujien rooleihin liittyvää tietoa vaiko ei.

Ensimmäiseen eli perinteiseen ryhmään kuuluville opiskelijoille opetus oli vanhan kaavan mukaista, toisin sanoen luennot ja harjoitukset pidettiin normaalisti eikä muuttujien rooleista kerrottu mitään. Toiseen ja kolmanteen ryhmään kuuluville opiskelijoille pidettiin yhteiset luennot, joissa kerrottiin muuttujien rooleista. Toisen ja kolmannen ryhmän harjoitukset poikkesivat kuitenkin toisistaan, sillä toisen ryhmän harjoituksissa ei käytetty PlanAni-animaattoria, kun taas kolmannen ryhmän harjoituksissa käytettiin. Jatkossa ryhmiä voidaan nimetä seuraavasti: ensimmäinen ryhmä eli perinteinen ryhmä, toinen ryhmä eli rooliryhmä sekä kolmas ryhmä eli animaatioryhmä.

Eri ryhmiin kuuluvat henkilöt eivät olleet tietoisia muiden ryhmien saamasta materiaalista ja opetuksesta, lukuun ottamatta toisen ja kolmannen ryhmän luentoja, jotka olivat ryhmillä yhteiset. Koehenkilöt jaettiin kolmeen ryhmään, koska haluttiin vertailla paitsi perinteistä opetusta saaneen ryhmän ja muuttujien rooleista kuulleiden ryhmien eroavaisuuksia, myös selvittää, minkälainen vaikutus PlanAni-animaattorin käyttämisellä muuttujien roolien opetuksessa on verrattuna tavallisen debuggerin käyttöön.

5.1.1 Ryhmiin jako - esitietolomake ja sen tietojen hyödyntäminen

Kurssin ensimmäisen luennon yhteydessä paikalle saapuneita opiskelijoita ($n = 80$) pyydettiin täyttämään esitietolomake (liite 1). Esitietolomakkeessa tiedusteltiin nimeä, sukupuolta, sähköpostiosoitetta, lukioarvosanoja äidinkielestä, matematiikasta, tietotekniikasta sekä kuvaamataidosta, taulukkolaskentakokemuksen määrää, mitä ohjelmointikursseja on käynyt sekä käytetyt ohjelmointikielet ja minkä kokoisia ohjelmia ja minkä verran on kirjoittanut.

Äidinkielen ja kuvaamataidon arvosanoja kysyttiin, koska haluttiin tietää, miten hyvin koehenkilö pystyy ymmärtämään ja ilmaisemaan asioita niin sanallisessa kuin kuvallisessa esitysmuodossa. Matematiikan ja tietotekniikan arvosanojen katsottiin korreloivan ohjelmoinnissa vaadittavan loogisen ajattelukyvyn kanssa. Taulukkolaskentakokemuksen ja ohjelmointikokemuksen määrää puolestaan kysyttiin, koska haluttiin tietää, kuinka paljon koehenkilöllä on käytännössä tietämystä ohjelmoinnista. Tulevien koehenkilöiden esitietolomakkeissa antamat tiedot kerättiin talteen, ja ne koodattiin etukäteen määritetyllä tavalla, jolloin niitä voitiin hyödyntää koehenkilöiden ryhmäjakoa suoritettaessa.

Esitietolomakkeista kerättyjen tietojen perusteella koehenkilöt jaettiin kolmeen eri ryh-

mään. Tavoitteena oli muodostaa ryhmistä tiedoiltaan ja kokemukseltaan niin tasavertaisia kuin suinkin mahdollista. Ryhmien tasavertaisuus varmistettiin vertailemalla tilastollisesti χ^2-testin (khii toiseen) avulla kokeen suorittamisen kannalta tärkeimmiksi katsottuja tietoja ryhmien välillä, eli matematiikan arvosanaa, taulukkolaskennan tietämystä ja ohjelmointikokemuksen määrää. Ryhmiin jakamista ja tilastollisen analyysin tekemistä toistettiin niin monta kertaa, että yllä mainituissa tiedoissa ei ryhmien välillä enää ollut tilastollisesti merkittäviä eroja. Tilastollisesti merkittävin ero oli ohjelmointikokemuksessa ($\chi^2 = 4.054, df = 2, p = 0.3988$).

Jotta ryhmäjako voitiin suorittaa mahdollisimman nopeasti, piti tehdä joitakin ennakkojärjestelyjä. Käytännössä tämä tarkoitti lyhyitä tietokoneohjelmia, joiden avulla saatiin automatisoitua raskaimmat ja pisimpään kestävät tehtävät, eli koehenkilöiden ryhmiin jakamiset sekä tilastollisten laskelmien teko. Kun sopiva ryhmäjako oli lopulta löydetty, generoitiin tähän liittyen verkkosivu, jossa oli esiteltynä koehenkilöiden sijoittuminen eri ryhmiin. Lisäksi lähetettiin massapostituksena koehenkilöille sähköpostitse etukäteen laaditut tiedotusviestit, joissa ilmoitettiin, mihin ryhmään koehenkilö kuuluu sekä mitkä ovat mahdolliset harjoitusajat. Tehtävien automatisoinnilla saavutettiin merkittävä ajan ja hermojen säästö sekä parannettiin myös tiedotusta koehenkilöiden suuntaan.

Ensimmäisen luennon jälkeen Ohjelmoinnin alkeet -kurssille ilmoittautui vielä 11 opiskelijaa lisää. Tämä aiheutti ongelmia kokeen järjestämisen kannalta ryhmäjaon muodostamisessa, sillä ensimmäisen luennon yhteydessä ilmoittautuneille 80 opiskelijalle oli jo ehditty ilmoittaa heille valitusta ryhmästä ja mahdollisista harjoitusajoista. Täten tehtyä ryhmäjakoa ei voinut enää muuttaa, sillä lopputuloksena olisi mitä suurimmalla todennäköisyydellä ollut koehenkilöiden kapinointi eri muodoissa ja kokeen epäonnistuminen. Kokeen järjestämisen kannalta täydellisessä maailmassa ryhmiin jako olisi voitu uusia kokonaan, mutta tosielämässä koehenkilöillä on luonnollisesti muutakin tekemistä kuin kokeeseen osallistuminen. Tämän vuoksi kurssille myöhemmin ilmoittautuneet piti sijoittaa jo arvottuihin ryhmiin käsin yksitellen siten, että tilastollisesti merkittäviä eroja ryhmien taitotasoissa ei syntyisi ainakaan merkittävässä määrin. Taulukossa 3 on kuvattu perustietoja koeryhmistä kaikkien koehenkilöiden ilmoittautumisten ja ryhmiin sijoittamisten jälkeen. Taulukossa olevat *p*-arvot liittyvät χ^2-testeihin, ja käytännössä ne ilmaisevat, että tilastollisesti merkittäviä eroja ryhmien välillä tärkeimmissä kategorioissa ei ole.

Taulukko 3: Perustiedot koehenkilöistä Kuittista & Sajaniemeä (2003) mukaillen.

	Ryhmä				*p*
	Perinteinen	Rooli	Animaatio	Kaikki	
Koehenkilöiden määrä	26	32	33	91	
Naisia koehenkilöistä (%)	30.8	18.9	24.2	24.2	0.7380
Lukiomatematiikan arvosana (1-3)	2.4	2.2	2.1	2.2	0.1343
Lukion äidinkielen arvosana (1-3)	2.4	2.2	2.1	2.2	0.1851
Lukion tietotekniikan arvosana (1-3)	2.5	2.6	2.6	2.6	0.9828
Lukion kuvaamataidon arvosana (1-3)	2.2	2.3	2.4	2.3	0.7039
Taulukkolaskentakokemus (0-2)	1.3	1.0	1.2	1.1	0.2827
Ohjelmointikurssien lukumäärä (0-2)	0.8	0.9	1.0	0.9	0.9539
Ohjelmointikokemus (0-2)	0.8	0.5	0.8	0.7	0.4566

5.1.2 Ryhmien erillisyyden varmistaminen

Jotta opiskelijat eivät olisi vaihdelleet eri ryhmien välillä, pidettiin toisistaan sisällöltään eriävät luennot samanaikaisesti, eli kurssilla tarvittiin kahta luennoitsijaa. Molemmilla luennoitsijoilla on pitkä luennointikokemus, ja molemmat heistä olivat pitäneet kyseisen kurssin aiemminkin. Perinteiseen tapaan pidettyjen luentojen luennoitsija ei tiennyt muuttujien roolituksesta mitään. Tästä johtuen perinteisten luentojen sisältö ei kärsinyt sen vuoksi, että luennoitsija olisi joutunut tietoisesti välttelemään joitakin muuttujien roolitusta sivuavia aiheita. Myös kurssiin liittyvät harjoitusryhmät oli jaettu edellä mainitun ryhmäjaon mukaisesti. Vaihtelua eri harjoitusryhmien välillä ei sallittu kuin samaan opetusryhmään kuuluvien kesken. Jos esimerkiksi perinteiseen ryhmään kuuluva henkilö ei päässyt osallistumaan jollekin harjoituskerralle, saattoi hän tulla ainoastaan toiseen aikaan pidettävään perinteisen ryhmän harjoituskertaan eikä osallistua vaikkapa animaatioryhmän harjoituksiin.

5.2 Käytännön opetustyö sekä protokollatehtävät

Kaikki Ohjelmoinnin alkeet -kurssille osallistuneet ryhmät saivat muuttujien rooleihin liittyviä asioita lukuun ottamatta informaatiosisällöltään samantasoista opetusta. Seuraavassa tarkastellaan lyhyesti opetukseen liittyviä asioita sekä kerrotaan kurssin yhteydessä järjestetyistä protokollatehtävistä.

5.2.1 Oppimateriaaleista

Kaikille ryhmille esitettiin sisällöllisesti muuten sama oppimateriaali ja samat esimerkkiohjelmat, mutta rooliryhmä ja animaatioryhmä käsittelivät kyseistä materiaalia muuttujien roolituksen kannalta. Muuttujien roolit esiteltiin rooliryhmässä ja animaatioryhmässä luvussa neljä mainittujen ohjeiden mukaan vähitellen luennoilla, kun ne tulivat esiin esimerkkiohjelmien yhteydessä. Luentomateriaalissa jokaisen muuttujan yhteydessä oli kommentti, jossa mainittiin kyseisen muuttujan rooli. Lisäksi rooliryhmälle ja animaatioryhmälle jaettiin nelisivuinen kokonaisuus, jossa lyhyiden esimerkkiohjelmien avulla esiteltiin muuttujien roolit. Vastaavasti harjoitustilaisuuksissa rooliryhmälle ja animaatioryhmälle mainittiin toistuvasti muuttujien roolit harjoitustehtäviä läpi käydessä.

Perinteisen ryhmän käyttöönsä saamat materiaalit olivat identtisiä rooliryhmän ja animaatioryhmän materiaalien kanssa, mutta muuttujien rooleihin liittyvä tietous puuttui. Koska näin oli, jaettiin perinteiselle ryhmälle roolikuvausten sijasta vastaava ”lyhyitä esimerkkiohjelmia” sisältävä kokonaisuus. Ohjelmien kommenteista vain puuttuivat viittaukset muuttujien roolitukseen. Sekä perinteinen ryhmä että rooliryhmä ja animaatioryhmä kävivät samat esimerkkiohjelmat läpi luentojen yhteydessä.

5.2.2 Visualisointien käyttämisestä

Harjoitusten rakenne oli kaikilla ryhmillä samantapainen. Jokainen harjoituskerta alkoi annettujen harjoitustehtävien läpikäynnillä, jossa opiskelijat esittivät laatimiaan ratkaisuja muille. Ensimmäistä kokoontumiskertaa lukuun ottamatta harjoituskerrat päättyivät aina jonkin esimerkkiohjelman toiminnan visualisointiin. Perinteinen ryhmä ja rooliryhmä käyttivät ohjelman suorituksessa apunaan visuaalista virheenjäljittäjää (debuggeria) Turbo Pascal v. 7.0:ää, kun taas animaatioryhmä käytti roolipohjaista ohjelma-animaattori PlanAnia. Animaatioiden suorittaminen kesti 20-40 minuuttia. Jokaisen animaatiokerran aluksi harjoitusten vetäjä esitteli animaation kulun askel askeleelta tietokonetta ja videoprojektoria apunaan käyttäen. Animaatioryhmässä harjoitusten vetäjä esitteli jokaisen uuden muuttujan roolin yhteydessä, mitä roolia ilmentävällä kuvalla haluttiin esittää ja mitkä ovat kyseisen roolin tärkeimmät ominaisuudet. Seuraavaksi jokaisessa ryhmässä käytiin ohjelman suoritus läpi opettajan valitsemilla syötteillä ja tämän jälkeen opiskelijat animoivat ohjelman suoritusta vapaasti valitsemiaan syötteitä

käyttämällä. Lopuksi harjoitusten vetäjät keskustelivat opiskelijoiden kanssa ongelmia aiheuttaneista asioista. Animointisessioiden aikana opiskelijoita rohkaistiin käymään animoinnit läpi hitaasti ajatuksen kanssa, ja kehotettiin miettimään etukäteen, mitä seuraava suoritettava ohjelmalause tekee ja miten se muuttaa ohjelmassa esiintyvien muuttujien arvoja tai vaikuttaa muuten ohjelman suoritukseen.

Jotta jokaisessa ryhmässä saatavat kognitiiviset ärsykkeet ja prosessit olisivat mahdollisimman samankaltaisia, suoritettiin jokaisessa ryhmässä aluksi samat esimerkkiohjelmat samoilla muuttujien arvoilla. Näin ollen eroavaisuudet opiskelijoiden ohjelmointitaidoissa eivät johtuisi erilaisista kognitiivisista ärsykkeistä vaan visualisoinnissa olleista sisällöllisistä eroista.

5.2.3 Protokollatehtävät

Kurssin keski- ja loppuvaiheessa järjestettiin osalle kurssin opiskelijoista ohjelman ymmärtämiseen ja ohjelman kirjoittamiseen liittyviä protokollatehtäviä, joiden avulla kerättiin pelkästään tietoa tutkimuksen suorittamista varten, eikä tehtävillä ollut mitään vaikutusta kurssiarvosanaan. Koehenkilöt näihin tehtäviin valittiin satunnaisesti sellaisten kurssin opiskelijoiden joukosta, joilla ei ollut aiempaa ohjelmointikokemusta tai kokemusta oli hyvin vähän. Tehtäviä oli kahdentyyppisiä: ohjelman ymmärtämistehtäviä sekä ohjelman kirjoittamistehtäviä. Protokollatehtävät oli testattu etukäteen ylempien vuosikurssin tietojenkäsittelytieteen opiskelijoilla ja niiden vaikeusastetta ja esitystapaa oli muokattu saadun palautteen mukaan.

Ensimmäinen ohjelman ymmärtämistehtävä, jota käytettiin kurssin puolivälissä, sisälsi 48 koodiriviä, eikä siinä ollut käytetty silmukkarakenteita. Toinen ohjelman ymmärtämistehtävä oli pituudeltaan 29 koodiriviä, ja tätä tehtävää käytettiin kurssin lopussa. Ohjelmien ymmärtämiseen liittyvät sessiot olivat kestoltaan 9 - 47 minuuttia. Koehenkilöiden tehtävänä oli paitsi perehtyä ohjelmaan, myös tehdä yhteenveto ohjelman toiminnasta suullisesti sekä selittää kaikkien ohjelmassa esiintyvien muuttujien tarkoitus.

Ohjelman kirjoittamistehtäviin koehenkilöt osallistuivat pareittain, ja molemmat pyrkivät yhdessä ratkomaan samaa ohjelmointiongelmaa. Koehenkilöitä yritettiin rohkaista ajattelemaan ääneen ongelmaa ratkaistessaan. Kun koehenkilöt olivat suorittaneet tehtävän loppuun, kokeen valvoja pyysi koehenkilöitä selittämään kunkin ohjelmassa käytetyn muuttujan tarkoituksen. Ohjelman kirjoittamissessioiden kesto vaihteli 18 ja

65 minuutin välillä. Sessioiden päätteeksi protokollatehtäviin osallistuneet opiskelijat palkittiin vaivoistaan kahvilippusella. Tehtävien suoritukset taltioitiin videonauhalle. Nauhoja tutkimalla yritetään selvittää, onko ryhmien välillä kvalitatiivisia eroja sen suhteen, miten he puhuvat ohjelmista käsitteellisellä tasolla. Protokollatehtäviä, niiden suorittamista ja niistä saatuja materiaaleja ei analysoida tarkemmin tässä työssä.

5.3 Lopputentti

Nelituntiseen lopputenttiin osallistuminen oli kurssin läpäisemiseksi pakollista. Lopputentti oli kaikille koehenkilöille samanlainen, ja se arvosteltiin normaalisti, eikä tenttikysymyksissä vaadittu tietämystä muuttujien rooleista. Täten tenttivastausten eri ryhmien välillä saattoi olettaa olevan samantasoisia, koska kuuluminen johonkin tiettyyn opetusryhmään ei anna mitään lisätietoa lopputentin suhteen, jota muilla ryhmillä ei olisi. Mahdolliset erot koehenkilöiden saamissa tenttituloksissa riippuivat siis pelkästään opettajien toisistaan poikkeavasta opetuksesta tai siitä, miten paljon koehenkilöt olivat panostaneet kyseiseen kurssiin. Koehenkilöiden antamat tenttivastaukset analysoitiin jälkikäteen kokeen järjestäjien tarpeita ajatellen mahdollisten ryhmien välisten kvalitatiivisten erojen löytämiseksi ja selvittämiseksi.

Lopputentti (liite 2) muodostui ”alkulämmittelykysymyksestä”, jossa kysyttiin, miten aktiivisesti opiskelija on ollut mukana luennoilla ja harjoituksissa sekä viidestä varsinaisesta tenttikysymyksestä. Varsinaiset viisi tenttikysymystä voidaan jakaa neljään eri tehtävätyyppiin. Suluissa on aina mainittu kunkin tehtävätyypin määrät: *muuttujiin liittymättömät tehtävät* (2), *ohjelman simuloimistehtävät* (1), *ohjelman ymmärtämistehtävät* (1) ja *ohjelman kirjoittamistehtävät* (1). Lämmittelykysymykseen vastaamisesta sai yhden pisteen ja varsinaisista tenttikysymyksistä maksimissaan kuusi pistettä kustakin. Seuraavassa käydään tenttikysymykset läpi.

5.3.1 Alkulämmittelykysymys

Alkulämmittelykysymyksellä oli olennaisen tärkeä merkitys kokeen tulosten tulkitsemisessa: sen avulla haluttiin varmistaa, että koehenkilöt, jotka eivät ole osallistuneet aktiivisesti opetukseen, eivät vaikuttaisi kokeesta saataviin lopputuloksiin. Onhan vähintäänkin kyseenalaista, onko annetulla opetuksella opiskelijaan mitään vaikutusta, jos hän on luennoilla tai harjoituksissa läsnä vähemmän kuin 40 % ajasta, joten tällai-

set koehenkilöt rajattiin tutkimuksen lopputuloksista pois. Jotta kaikki tenttiin osallistujat varmasti vastaisivat alkulämmittelykysymykseen, annettiin siihen vastaamisesta yksi piste.

5.3.2 Muuttujiin liittymättömät kysymykset

Kaksi ensimmäistä tehtävää (EI-MUUT/1 ja EI-MUUT/2) olivat tyypiltään muuttujiin liittymättömiä tehtäviä. Näiden tehtävien tarkoituksena oli selvittää kurssin luennoitsijoiden aiheuttamat mahdolliset eroavaisuudet, saada jonkinlaista osviittaa siitä, millainen on kunkin koehenkilön taitotaso ja missä määrin kukin koehenkilö on panostanut kyseiseen kurssiin. Kun näiden tehtävien tuloksia analysoitiin, niistä saatiin vertailupohja sille, minkälaisia pisteitä pitäisi kunkin koehenkilön kohdalla olla odotettavissa muista tehtävistä. Näissä kahdessa tenttikysymyksessä tarvittiin kykyä soveltaa opittuja asioita koehenkilöille uusissa tilanteissa.

Ensimmäisessä tehtävässä kyseltiin erilaisia silmukkarakenteita sekä niiden soveltuvuutta eri tilanteisiin. Toisessa kysymyksessä puolestaan esiteltiin jonkin tuntemattoman kielen syntaktiset säännöt sekä mahdollisia kyseisen kielen merkkijonoja. Tehtävänä oli kertoa, mitkä merkkijonoista olivat laillisia ja miksi.

5.3.3 Ohjelman simuloimistehtävä

Ohjelman simuloimistehtävässä (OHJ-SIMU) koehenkilöitä pyydettiin kertomaan, mitä 15-rivinen Pascal-kielinen ohjelma tulostaa annetulla syötteellä. Ohjelma etsi alkulukuja Eratostheneen seulaa käyttämällä, ja sen tulosteena oli rivittäin välilyönnillä erotettuna alkuluku sekä siihen mennessä löydettyjen alkulukujen summa. Ohjelma sisälsi kaksi *askeltajaa*, yhden *kiintoarvon*, yhden *kokoojan* sekä yhden *yksisuuntainen lippu* -taulukon. Muuttujien roolit ohjelmassa olivat helposti ymmärrettävissä, mutta ohjelman sisäinen toimintaperiaate oli tarkoituksella sekavan oloinen. Samoin ohjelmassa esiintyvien muuttujien tunnukset olivat yksikirjaimisia ja merkityksettömiltä tuntuvia. Se pakotti koehenkilöt simuloimaan ohjelman toimintaa vaihe vaiheelta, jotta he voisivat päätellä ohjelman antaman tulosteen.

5.3.4 Ohjelman ymmärtämistehtävä

Ohjelman ymmärtämistehtävässä (OHJ-YM) oli 19-rivinen Pascal-kielinen ohjelma, joka tulosti viikon lääkeannostelutaulukon sekä yhteensä tarvittavan lääkkeen määrän. Tehtävänä oli kuvailla kyseisen ohjelman tarkoitus ja sen toimintaperiaate. Ohjelmassa oli yksi *kiintoarvo*, *askeltaja*, *tuoreimman säilyttäjä* sekä *kokooja*. Muuttujien tunnukset olivat yhtä muuttujaa lukuun ottamatta yksikirjaimisia, mutta silti jonkin merkityksen omaavia. Poikkeuksena oli muuttuja, johon tallennettiin käyttäjän antama syöte (potilaan paino). Tämän muuttujan tunnus oli kokonainen merkityksellinen sana.

Ohjelmalla oli selkeä ja yksinkertainen toimintaperiaate, ja oli helppo ymmärtää, mitä ohjelma tekee. Oletusarvona oli, että kaikki opiskelijat ymmärtäisivät, mistä ohjelmassa on kysymys, joten kiinnostuksen kohteena olikin se tapa, jolla he selittäisivät ohjelman toimintaa. Muuttujien nimet oli nimetty tarkoituksenmukaisesti, jotta se helpottaisi ohjelman sovellusalueen tunnistamista. Täyspitkiä sanoja ei kuitenkaan muuttujien nimissä käytetty, jotta voitaisiin tehdä ero siinä, milloin opiskelijoiden kirjoittamissa ohjelman toimintakuvauksissa on kyse pelkästään muuttujien tunnuksista ja milloin taas ohjelman sovellusalueesta.

5.3.5 Ohjelman kirjoittamistehtävä

Viimeisenä oli vuorossa ohjelman kirjoittamistehtävä (OHJ-KIRJ). Tavoitteena oli laatia Pascal-kielinen ohjelma, joka aluksi saa syötteenä harjoitusten ja harjoitustehtävien lukumäärän. Seuraavaksi ohjelma kysyy ratkaistujen harjoitustehtävien määrän, minkä jälkeen ohjelma laskee, onko harjoitustehtäviä ratkaistu riittävä määrä. Kyseiset toimenpiteet pitää suorittaa niin monta kertaa, kunnes kaikkien opiskelijoiden tiedot on syötetty ja analysoitu. Ohjelman kirjoittamistehtävä oli sellainen, että sen oletettiin olevan ”järkevä” tenttiä suorittavan opiskelijan kannalta, ja ratkaisussa tarvittaisiin useita erilaisia muuttujia ja täten myös muuttujien rooleja. Tehtävän optimiratkaisussa käytettäisiin kahta *tuoreimman säilyttäjää*, jotka muuttuvat *kiintoarvoiksi* ohjelman alkuvaiheen jälkeen, kahta *askeltajaa*, yhtä *tuoreimman säilyttäjää* sekä yhtä *kokoojaa*.

5.4 Tulokset

Alkuperäisistä 91 koehenkilöistä 60 otti osaa lopputenttiin. Kuten aiemmin jo mainittiin, kokeen kannalta ”epäilyttävä aines”, eli henkilöt, jotka osallistuivat joko luennoille tai harjoituksiin vähemmän kuin 40% kokonaismäärästä, jätettiin huomioimatta kokeen lopputuloksissa. Tämän seurauksena kokeen lopputuloksiin asti vaikuttamaan selvisi kaikkiaan 44 koehenkilöä. Ryhmien kesken henkilöiden jakautuma oli seuraavanlainen: perinteisestä ryhmästä mukaan kelpuutettiin 10, rooliryhmästä 17 ja animaatioryhmästä 17 koehenkilön tulokset.

Taulukko 4: Ryhmien arvosanat Sajaniemeä & Kuittista (in press) mukaillen.

Kysymys	Ryhmä					
	Perinteinen *n* = 10		Rooli *n* = 17		Animaatio *n* = 17	
	KA	KH	KA	KH	KA	KH
EI-MUUT/1	4.9	0.98	3.9	1.69	4.6	1.48
EI-MUUT/2	4.3	1.11	3.5	1.61	3.9	1.56
OHJ-SIMU	3.4	2.21	2.5	2.20	2.8	2.36
OHJ-YM	4.5	0.59	4.1	1.69	3.6	1.52
OHJ-KIRJ	3.8	1.48	3.8	1.72	3.9	1.51

Taulukossa 4 on listattuna kunkin ryhmän saamien arvosanojen keskiarvot tehtävittäin. Taulukossa KA tarkoittaa keskiarvoa ja KH keskihajontaa. Ryhmien väliset eroavaisuudet eivät ole tilastollisesti merkittäviä. Kahden ensimmäisen tehtävän (EI-MUUT/1 ja EI-MUUT/2) arvosanat käyttäytyvät samalla tavoin, ts. perinteisen ryhmän arvosanan keskiarvo on paras, animaatioryhmän toiseksi paras ja rooliryhmä on molemmissa kysymyksissä heikoin. Pearsonin korrelaatiokerroin näiden kahden arvosanan välillä on $r = 0.412$, kaksisuuntaisen t-testin mukaan todennäköisyys, että noin suuri korrelaatio näiden kahden arvosanan välillä olisi sattumanvaraista on tilastollisesti merkittävä ($SE(b) = 0.281, t = 2.931, df = 42, p = 0.0054$).

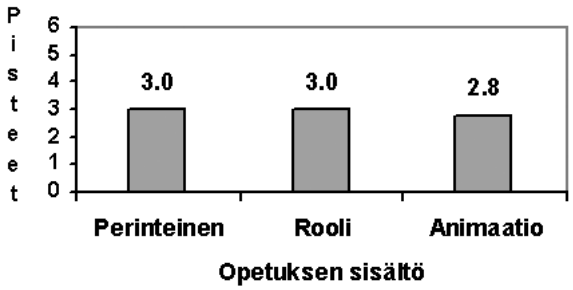
Koska nämä kaksi kysymystä eivät liity millään tavalla muuttujiin, eivät arvosanoissa olevat erot voi johtua erottavasta tekijästä eli opetuksen sisällöstä. Sen sijaan ero on selitettävissä tekijöillä, joita ei voitu kontrolloida, eli opettajien välisillä eroavaisuuksilla sekä sillä, miten aktiivisesti koehenkilöt keskittyivät kyseiseen kurssiin.

Jotta nämä erot tulisivat huomioiduiksi, ei taulukossa olevia arvosanoja käytetä sellai-

senaan, vaan ryhmien välinen tasoero on pyritty huomioimaan. Aluksi laskettiin jokaisen koehenkilön kohdalta erotukset hänen ohjelman simuloimistehtävästä (OHJ-SIMU), ymmärtämistehtävästä (OHJ-YM) ja kirjoittamistehtävästä saamien pisteiden ja muuttujiin liittymättömien kysymysten (EI-MUUT/1 ja EI-MUUT/2) keskiarvon välillä (jokaisesta mainitusta tehtävästä (OHJ-SIMU, OHJ-YM, OHJ-KIRJ) erikseen). Saaduista erotuksista laskettiin seuraavaksi ryhmäkohtainen keskiarvo. Jotta saatuja ryhmäkohtaisia keskiarvoja olisi helpompi lukea, muunnettiin ne muotoon, jossa ryhmäkohtaisiin keskiarvoihin on lisätty jokin vakioarvo, jonka jälkeen perinteisen ryhmän saama pistemäärä on aina 3.0.

5.4.1 Simuloimistehtävän tulokset

Ohjelman simuloimistehtävään (OHJ-SIMU) liittyvät skaalatut keskiarvopisteet ovat nähtävillä kuvassa 29. Ryhmien välillä ei ole tilastollisesti merkittäviä eroja.

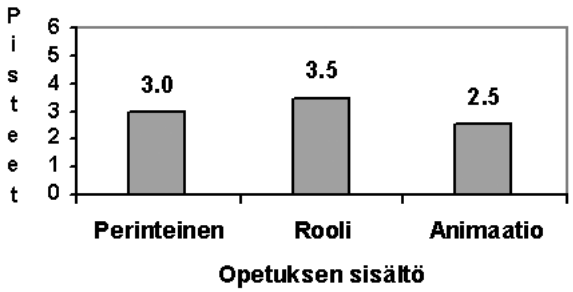


Kuva 29: Skaalatut keskiarvopisteet ohjelman simulointitehtävästä Sajaniemeä & Kuittista (in press) mukaillen.

5.4.2 Ymmärtämistehtävän tulokset

Kuvassa 30 ovat ohjelman ymmärtämistehtävään (OHJ-YM) liittyvät skaalatut keskiarvopisteet. Rooliryhmän ja animaatioryhmän saamien pisteiden välillä on tilastollisesti merkittävä ero (kaksisuuntainen t-testi, $t = 2.026, df = 41, p = 0.0493$).

Ohjelman ymmärtämistehtävässä (OHJ-YM) annetut vastaukset analysoitiin varsinaisen lopputenttiarvostelun lisäksi vielä erikseen empiirisen kokeen tarpeita ajatellen. Analysoinnin kohteeksi otettiin 44 koehenkilön vastausten joukosta löydetyt virheettömät vastaukset, joista kävi ilmi, että vastaaja oli ymmärtänyt ohjelman jokaisen yksi-



Kuva 30: Skaalatut keskiarvopisteet ohjelman ymmärtämistehtävästä Sajaniemeä & Kuittista (in press) mukaillen.

tyiskohdan toiminnan. Näistä valituista vastauksista laskettiin ryhmäkohtaiset keskiarvot kyseiseen tehtävään liittyen. Perinteisen ryhmän keskiarvo oli 4.6, rooliryhmän 5.0 ja animaatioryhmän 4.1 pistettä. (rooliryhmä vs. animaatioryhmä, kaksisuuntainen t-testi, $t = 1.718, df = 27, p = 0.0972$).

Lopputentissä kyseisen tehtävän pisteet perustuivat paitsi siihen, miten syvällisesti ohjelman toiminta oli selvitetty, mutta myös tenttivastauksen arvioijan mielipiteeseen siitä, miten "laadukas" selvitys oli. Koska kaikki empiiriseen kokeeseen tutkittavaksi valitut vastaukset osoittivat hyvää ohjelman ymmärtämistä, viittaavat koehenkilöiden saamat erilaiset lopputenttipisteet eroavaisuuksiin tavoissa, joilla he kuvasivat ohjelman toimintaa.

Jotta voitiin löytää kvalitatiivisia eroavaisuuksia koehenkilöiden antamista ohjelman toiminnan kuvauksista, otettiin avuksi sovellettu tapa Penningtonin (1987) alakohdassa 3.3.2 esitellystä ohjelmatietämyksen jaottelusta. Penningtonin esittämää jakoa ei voinut soveltaa sellaisenaan, koska hänellä koehenkilöinä olivat ekspertit, jotka perehtyivät keskikokoiseen ohjelmaan, kun taas tämän kokeen yhteydessä oli kyse noviiseista ja lyhyestä ohjelmasta. Avuksi otettiin analysointitapa (Good & Brna (2003)), jossa on kaksi luokitteluperustetta: *informaatiotyypit* (information types) sekä *oliokuvaukset* (object descriptions). Informaatiotyyppeihin perustuva jaottelu on hienojakoisempi ja täsmällisempi kuin Penningtonin esittämä jaottelu, kun taas oliokuvauksiin pohjautuva tapa on sisällöltään rajoittuneempi Penningtonin tapaan verrattuna eikä yhtä tarkka.

Informaatiotyyppeihin perustuvan luokittelun avulla koehenkilöiden kirjoittamissa ohjelmakuvauksissa olevat lauseet voidaan jakaa kymmeneen eri luokkaan niiden sisältämän informaatiotyypin perusteella. Luokat ovat *tehtävä* (function), *toiminnot* (actions), *operaatiot* (operations), *tila-korkea* (state-high), *tila-matala* (state-low), *tieto* (data),

kontrolli (control), *tarkentava* (elaborate), *meta* (meta) ja *epäselvä* (unclear).

Tehtävään liittyvässä lauseessa kuvataan ohjelman kokonaistavoitetta eli sen tehtävää korkealla tasolla. *Toimintoihin* keskittyvässä lauseessa kuvataan ohjelman tapahtumia matalammalla tasolla kuin tehtävässä, mutta korkeammalla tasolla kuin operaatioissa. *Operaatioita* kuvaavassa lauseessa keskitytään ohjelman matalan tason tapahtumiin ja usein yksittäisen rivin toimintaan. *Tila-korkea*-lauseissa kuvataan korkealla abstraktiotasolla ohjelman senhetkistä tilaa tietyn ehdon toteutumisen jälkeen ja *tila-matala*-lauseissa tilaa kuvataan matalammalla tasolla. *Tieto-lauseissa* kuvataan ohjelman syötteitä, tulosteita ja tietovuota kun taas *kontrolliin* liittyvissä lauseissa esitellään tietoa, joka liittyy ohjelman kontrollirakenteisiin. *Tarkentava*-lauseissa kuvataan yksityiskohtaisesti ja kerrotaan lisätietoa jo aiemmin mainittuihin asioihin liittyen. *Meta*-lauseet kuvaavat ohjelmoijan oman ajatteluprosessin etenemistä ja epäselviä lauseita taas ei voi sijoittaa mihinkään aiemmin mainituista, koska niiden merkitys on epäselvä.

Taulukossa 5 on koottu ryhmittäiset keskiarvot eri informaatiotyyppiä edustavien lauseiden prosentuaalisista määristä ohjelmakuvauksissa. Yhden koehenkilön vastaus rooliryhmästä jätettiin huomioimatta, koska se sisälsi lähinnä kopioitua ohjelmakoodia. Tämä poikkesi voimakkaasti muiden koehenkilöiden vastauksista, jolloin kyseisen vastauksen mukaan ottaminen olisi vääristänyt tilastollisesta analyysista saatavia tuloksia.

Taulukko 5: Ryhmäkohtaiset keskiarvot eri informaatiotyyppiä edustavien lauseiden prosentuaalisista määristä ohjelmakuvauksissa Sajaniemeä & Kuittista (in press) mukaillen.

Informaatiotyyppi	Ryhmä		
	Perinteinen	Rooli	Animaatio
Tehtävä	10.1	8.2	9.6
Toiminnot	20.4	18.1	21.2
Operaatiot	11.9	13.2	11.9
Tila-korkea	0.4	2.1	0.0
Tila-matala	0.2	0.6	0.5
Tieto	26.9	25.3	30.9
Kontrolli	6.0	6.0	6.0
Tarkentava	22.3	26.1	18.6
Meta	2.0	0.3	1.3

Saadut keskiarvot testattiin käyttämällä ANOVA-analyysia (ANalysis Of VAriance

between groups). Ainoa tilastollisesti merkittävä eroavaisuus havaittiin *tila-korkea*-lauseiden kohdalla ($F(2, 40) = 3.028, p = 0.0596$). Rooliryhmä käytti enemmän *tila-korkea*-lauseita kuin muut ryhmät, mutta lauseiden absoluuttinen määrä on pieni. Rooliryhmästä 31.3 % käytti *tila-korkea*-lauseita, perinteisestä ryhmästä 10 % koehenkilöistä käytti niitä, mutta animaatioryhmästä niitä ei käyttänyt yksikään koehenkilö.

Suurin absoluuttinen eroavaisuus oli *tarkentavissa* lauseissa. Rooliryhmä käytti *tarkentavia* lauseita enemmän kuin muut ryhmät, mutta ero ei ole tilastollisesti merkittävä ($F(2, 40) = 1.032, p = 0.3654$).

Toisessa luokittelutavassa eli oliokuvauksissa keskitytään siihen, miten kuvataan ohjelmassa olevia olioita. Viittaukset olioihin voidaan luokitella seuraavan tyyppisiksi: *pelkästään ohjelma* (program only), *ohjelma* (program), *ohjelma – yleinen taso* (program – real-world), *ohjelma – sovellusalue* (program – domain), *sovellusalue* (domain), *epäsuora viittaus* (indirect reference) ja *epäselvä* (unclear).

Pelkästään ohjelma -tyyppisissä olioissa on kuvattu asiaa, jolla on merkitys ainoastaan kyseisen ohjelman yhteydessä, eikä missään muussa kontekstissa, esimerkiksi ”laskuri”. *Ohjelma*-viittauksessa on kyse ohjelmatason kuvauksesta, kuten muuttujan tunnuksesta tai tietorakenteesta. *Ohjelma – yleinen taso* -viittauksessa käytetään terminologiaa, joka on ymmärrettävissä niin ohjelmatasolla kuin yleiselläkin tasolla, esimerkiksi ”numero”. *Ohjelma – sovellusalue* -viittaus sisältää niin ohjelmaan kuin sovellusalueeseen liittyvää terminologiaa, esimerkiksi "taulukko, jossa on korkeudet". *Sovellus*-viittauksessa käytetään sovellusalueeseen liittyviä termejä, kuten "sadepäivät". *Epäsuorissa viittauksissa* taas käytetään oliosta nimen mukaisesti epäsuoraa viittausta, kuten ”se”. Viimeisenä tuleva *epäselvä viittaus* on sisältönsä puolesta niin monitulkintainen, että sitä ei voida koodata.

Taulukossa 6 on esitelty ryhmäkohtaiset keskiarvot eri oliokuvaustyyppiä edustavien lauseiden määristä ohjelmakuvauksissa. Rooliryhmä ja animaatioryhmät käyttivät enemmän *pelkästään ohjelma* -viittauksia ($F(2, 40) = 1.791, p = 0,1800$; perinteinen ryhmä vs. animaatioryhmä kaksisuuntainen t-testi $t = 1.798, df = 40, p = 0.0797$). Tämä johtui enimmäkseen siitä, että heillä oli käytettävissä rikas sanasto rooleihin liittyvän opetuksen ansiosta kun taas perinteiseen ryhmään kuuluvat käyttivät ainoastaan ilmauksia ”laskuri” ja ”silmukkalaskuri”. Toisaalta *ohjelma*-viittaukset olivat harvinaisimpia animaatioryhmällä, mutta ero ei ole tilastollisesti merkittävä ($F(2, 40) = 1.012, p = 0.3725$).

Taulukko 6: Ryhmäkohtaiset keskiarvot eri oliokuvaustyyppiä edustavien lauseiden prosentuaalisista määristä ohjelman ymmärtämistehtävään liittyvissä ohjelmakuvauksissa Sajaniemeä & Kuittista (in press) mukaillen.

Oliokuvaustyyppi	Ryhmä		
	Perinteinen	Rooli	Animaatio
Pelkästään ohjelma	0.8	2.4	5.1
Ohjelma	27.1	24.8	16.0
Ohjelma - yleinen taso	0.2	1.3	0.5
Ohjelma - sovellusalue	11.3	11.9	9.6
Sovellusalue	49.9	52.6	55.2
Epäsuora viittaus	5.1	3.0	4.7
Epäselvä	5.7	3.9	9.0

Vaikka ryhmien keskiarvot taulukossa 6 ovat hyvin samantapaisia, on ryhmien sisällä merkittäviä eroavaisuuksia viittausten käytössä. Esimerkiksi perinteisessä ryhmässä ohjelmakuvaukset sisälsivät joko vähän tai paljon sovellusalueeseen liittyviä lauseita kun taas muissa ryhmissä sovellusalueeseen liittyviä lauseita käytettiin tasaisemmin. Jotta tämä eroavaisuus voitiin analysoida tarkemmin, otettiin kuvausten jaottelussa käyttöön samantapainen strategia kuin Penningtonilla (1987) oli. Ohjelmakuvaukset luokiteltiin kolmeen eri tyyppiin sen mukaan, minkä verran ne sisälsivät sovellusaluetta ja ohjelmaa koskevia oliokuvauksia. Ohjelmakuvaukset, jotka sisälsivät vähintään 67% sovellusalueeseen liittyviä lauseita (epäsuoria ja epäselviä lauseita ei otettu huomioon) kutsuttiin *sovellusaluekuvauksiksi*. Vähintään 67% ohjelma- ja pelkästään ohjelma -lauseita sisältäneet lauseet luokiteltiin *ohjelmatason kuvauksiksi* ja kaikki muut luokiteltiin *ristiinviittauskuvauksiksi*, koska niissä jakauma sovellusalueeseen ja ohjelmaan liittyvässä tiedossa oli tasaisempi.

Taulukossa 7 on koottuna koehenkilöiden käyttämät ohjelmakuvaustyypit ryhmittäin jaoteltuna. Ristiinviittauskuvausten määrä perinteisessä ryhmässä oli merkittävästi pienempi kuin muissa ryhmissä ($\chi^2 = 10.773, df = 2, p = 0.046$). Penningtonin tutkimuksessa hyvät lähdekoodin ymmärtäjät käyttivät lähes poikkeuksetta ristiinviittauskuvausta kun taas heikommat ymmärtäjät käyttivät joko ohjelmatason tai sovellusaluetason kuvauksia.

Tämä analyysi ei selittänyt, miksi oikean vastauksen ymmärtämistehtävään antaneiden henkilöiden arvosanojen välillä on niin suuria eroavaisuuksia. Tämän selvittämiseksi

Taulukko 7: Koehenkilöiden käyttämät ohjelmakuvaustyypit ohjelman ymmärtämistehtävässä ryhmittäin jaoteltuna Sajaniemeä & Kuittista (in press) sekä Sajaniemeä (2004c) mukaillen.

Ohjelmakuvauksen tyyppi	Ryhmä		
	Perinteinen	Rooli	Animaatio
Ohjelma	2 (20%)	0 (0%)	1 (5.9%)
Ristiinviittaus	1 (10%)	12 (75%)	10 (58.8%)
Sovellusalue	7 (70%)	4 (25%)	6 (35.3%)

tutkittiin korrelaatioita arvosanojen ja täydellistä tai melkein täydellistä ohjelman ymmärtämistä osoittaneiden ohjelmakuvausten välillä. Informaatiotyyppi- ja oliokuvauspohjaisten kuvausten lisäksi tarkastelun kohteeksi otettiin kuvauksen pituus ja kuvauksen oikeellisuus. Ohjelmakuvausten pituudet eri ryhmillä olivat seuraavat: 152.1 sanaa perinteisellä ryhmällä, 154.4 sanaa rooliryhmällä ja 141.4 sanaa animaatioryhmällä.

Taulukossa 8 on esitettynä tilastollisesti merkittävät korrelaatiot. Suurimmat korrelaatiot ovat negatiivisia ja liittyvät ohjelman korkeamman tason piirteiden kuvaamiseen. Kuvaukset, joissa oli painotettu muita enemmän toimintoihin ja sovellusalueeseen liittyviä lauseita, saivat huonompia tenttipisteitä kun taas kuvauksissa, joissa keskityttiin matalamman tason operaatioiden kuvauksiin ja ohjelmaterminologiaan saivat parempia tenttipisteitä. Jopa ohjelmakuvauksen pituuksien ja toiston määrät vastauksissa korreloivat enemmän arvosanan kanssa kuin kuvausten oikeellisuus.

Taulukko 8: Pearsonin korrelaatioita ohjelman ymmärtämistehtävään liittyen Sajaniemeä & Kuittista (in press) mukaillen.

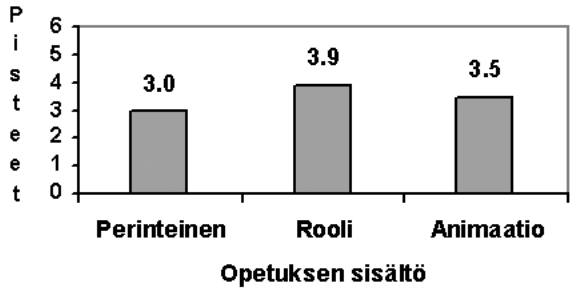
Ohjelmakuvauksen tyyppi	Korrelaatio	p	Rooliryhmä vs. animaatioryhmä
Informaatiotyyppi / Toiminnot	-0.763	0.000	<
Oliokuvaus / Sovellusalue	-0.731	0.000	<
Pituus	+0.629	0.000	>
Oliokuvaus / Ohjelma	+0.548	0.001	>
Informaatiotyyppi / Tarkentava	+0.544	0.001	>
Oliokuvaus / Epäselvä	+0.414	0.012	<
Oikeellisuus	+0.413	0.012	=
Informaatiotyyppi / Operaatiot	+0.395	0.017	>

Taulukon 8 viimeisessä sarakkeessa kerrotaan, kummalla mainituista ryhmistä oli isompi keskiarvo kyseisessä asiassa. Rooliryhmä oli animaatioryhmää parempi mil-

tei kaikissa mitatuissa arvoissa, joissa korrelaatio arvosanaan nähden oli positiivinen ja päinvastoin arvoissa, joissa korrelaatio oli negatiivinen. Vaikka korrelaatio ei välttämättä tarkoita kausaliteettia, ovat erot ryhmien välisten arvosanojen eroissa nyt selitettävissä. Ohjelmakuvauksia arvostelleet opettajat ovat antaneet hyviä pisteitä vastauksista, joissa on kuvattu ohjelmaa matalalla tasolla, ehkä myös pitempiä vastauksia on suosittu. Heikompia pisteitä edellä mainittuihin kuvauksiin verrattuna on puolestaan annettu vastauksista, joissa on keskitytty ohjelman korkeamman tason kuvaukseen.

5.4.3 Kirjoittamistehtävän tulokset

Ohjelman kirjoittamistehtävään liittyvät skaalatut keskiarvopisteet on esitelty kuvassa 31. Erot ryhmien välillä eivät ole tilastollisesti merkittäviä.



Kuva 31: Skaalatut keskiarvopisteet ohjelman kirjoittamistehtävästä Sajaniemeä & Kuittista (in press) mukaillen.

Jotta voitiin saada parempi käsitys mahdollisista eroavaisuuksista ryhmien välillä, piti koehenkilöiden tekemät virheet syntaksiin liittyviä pikkuvirheitä lukuun ottamatta luokitella jollakin tavalla. Koska joissakin ohjelmissa oli suuria puutteita, oli niistä mahdotonta laskea täsmällistä virheiden määrää. Tämän vuoksi virheiden määrään pohjautuva analysointitapa ei tullut kyseeseen, vaan sen sijaan laskettiin koehenkilöiden määriä, joilla oli ongelmia tietynlaisissa virhetyypeissä.

Ensimmäisessä käytetyssä virheiden luokittelutavassa tarkasteltiin, minkä tyyppistä oli se puuttuva tietämys, jota olisi tarvittu ohjelmassa olevan virheen poistamiseksi. Puuttuvia tietämystyyppejä oli kolmenlaisia: ohjelmointikieleen liittyvä tietämys, eli ongelmat Pascalin syntaksiin tai semantiikkaan liittyen, ohjelmointitietämys, kuten sisäkkäisten silmukoiden käyttöön liittyvät ongelmat, sekä sovellusalueeseen liittyvä tietämys, eli ongelmat esimerkiksi ohjelman tarkoituksen ymmärtämiseen liittyen. Taulu-

kossa 9 on esitettynä ryhmäkohtaisesti, kuinka suuri osa koehenkilöistä teki virheitä, jotka johtuivat tietyn tyyppisestä puuttuvasta tietämyksestä. Ainoastaan muutamalla henkilöllä oli ongelmia ohjelmointikieleen liittyen, mutta melkein kaikki koehenkilöt tekivät ohjelmointivirheitä ja heillä oli vaikeuksia ongelman yksityiskohtaisessa ymmärtämisessä. Erot ryhmien välillä eivät ole tilastollisesti merkittäviä.

Taulukko 9: Koehenkilöiden osuus, jotka tekivät virheitä ohjelman kirjoittamistehtävässä tietyn tietämystyypin puuttumisen vuoksi Sajaniemeä & Kuittista (in press) mukaillen.

Tarvittava tietämystyyppi	Ryhmä		
	Perinteinen	Rooli	Animaatio
Ohjelmointikieli	20.0	5.9	5.9
Ohjelmointi	90.0	94.1	94.1
Sovellusalue	80.0	88.2	70.6

Koska muuttujien rooleja hyödyntävän opetuksen ei oleteta vaikuttavan sovellusalueen ymmärtämiseen vaan ohjelmointitietämykseen, analysoitiin ohjelmointivirheitä yksityiskohtaisemmin. Erityisen kiinnostuksen kohteena oli, että millä ohjelmatietämyksen tasolla koehenkilöillä oli ongelmia, kun he kirjoittivat ohjelmia. Penningtonin (1987) mukaan operaatioihin ja kontrollirakenteisiin liittyvä tietämys heijastaa ohjelman pintarakenteisiin liittyvää tietämystä, joka on helposti havaittavissa. Sen sijaan ohjelman tietovuohon ja ohjelman tavoitteisiin liittyvä tietämys viittaa ohjelman syvärakenteiden ymmärtämiseen, mikä on merkki hyvästä lähdekoodin ymmärtämisestä. Ohjelmointivirheet jaoteltiin kolmeen tyyppiin, eli kontrolliin, tietoon ja tiedon erikoistarpeisiin liittyviin virheisiin. Kontrolliin liittyvät virheet ovat virheitä ohjelman kontrollirakenteiden käyttämisessä, tietoon liittyvät virheet ovat virheitä muuttujien sisäiseen ja muuttujien väliseen tietovuohon liittyen. Tiedon erikoistarpeisiin liittyvät virheet liittyvät muuttujien käyttöön poikkeuksellisissa tilanteissa, kuten virheellisten syötteiden pois rajaamisessa.

Vaikka perinteinen ryhmä teki vähiten pintarakenteisiin liittyviä virheitä (taulukko 10) ja animaatioryhmä eniten, on asia päinvastoin syvärakenteisiin liittyvässä ohjelmatietämyksessä. Ero perinteisen ryhmän ja animaatioryhmän välillä tiedon erikoistarpeissa on tilastollisesti merkittävä ($\chi^2 = 2.830, df = 1, p = 0.0998$).

Myös kaikki muuttujiin liittyvät virheet analysoitiin tutkimalla muuttujien roolit ja niiden käyttämiseen liittyvät virheet. Taulukossa 11 on lueteltuna kaikki virhetyypit

Taulukko 10: Niiden koehenkilöiden määrä ja osuus, jotka tekivät ohjelmointivirheitä ohjelman kirjoittamistehtävässä ohjelmatietämyksen syvyyden mukaan jaoteltuna Sajaniemeä & Kuittista (in press) sekä Sajaniemeä (2004c) mukaillen.

Puuttuvan ohjelmatietämyksen tyyppi	Ryhmä		
	Perinteinen	Rooli	Animaatio
Kontrolli	4 (40%)	8 (47.1%)	9 (52.9%)
Tieto	7 (70%)	10 (58.8%)	9 (52.9%)
Tiedon erikoistarpeet	8 (80%)	10 (58.8%)	8 (47.1%)

muuttujien roolien ja niiden käytön vaativuuden mukaan jaoteltuna. Suurimmat erot ryhmien välillä ovat monimutkaisen *askeltajan* käytön yhteydessä, jossa rooliryhmä selviää huonoiten, sekä monimutkaisen *tuoreimman säilyttäjän* käyttämisen yhteydessä, jossa animaatioryhmä selviää parhaiten. Mikään eroavaisuus ryhmien välillä ei kuitenkaan ole tilastollisesti merkittävä.

Taulukko 11: Muuttujiin liittyvät virheet Sajaniemeä & Kuittista (in press) mukaillen. Numerot ovat prosentuaalisia osuuksia ryhmiin kuuluvien koehenkilöiden kokonaismääristä.

Rooli	Perinteinen	Rooli	Animaatio	Tavallisimmat virheet
Yksinkertainen tuoreimman säilyttäjän käyttö	10.0	23.5	29.4	Rooliryhmä ja animaatioryhmä: puuttuva päivitys (a)
Yksinkertainen askeltajan käyttö	40.0	29.4	35.3	Kaikki ryhmät: väärin sijoitettu alustus
Monimutkainen askeltajan käyttö	30.0	52.9	23.5	Rooliryhmä: puuttuva päivitys (b), puuttuva alustus
Monimutkainen tuoreimman säilyttäjän käyttö	70.0	64.7	41.2	Perinteinen: käytetty taulukkoa; Kaikki: puuttuva päivitys (c)
Monimutkainen kokoojan käyttö	70.0	70.6	64.7	Perinteinen ja animaatioryhmä: väärin sijoitettu alustus; Rooliryhmä: puuttuva alustus

Virheiden potentiaalisia syitä on monia. Esimerkiksi väärässä paikassa olevat muuttujien alustukset ja arvon päivitykset viittaavat ongelmiin kontrollirakenteissa, kun taas

tarpeeton taulukon käyttäminen on merkki puuttuvasta (Détienne, 2002) tai hauraasta (fragile) (Perkins & Martin, 1986) menetelmätietämyksestä. Tällöin ohjelmoija ei tiedä, kuinka käyttää tuoreimman säilyttäjää eri arvojen läpikäymiseen. Puuttuva muuttujan alustaminen tarkoittaa, että ohjelmoija on käyttänyt fokaalista laajennusta (Rist, 1989), eli on kirjoittanut muuttujan arvoa päivittävän lauseen ennen sen alustamista. Tämä on merkki puuttuvasta menetelmätietämyksestä.

On syytä käydä läpi taulukossa 11 esiintyvät muuttujien arvojen puuttuvien päivitysten mahdolliset syyt. Ristin (1989) käyttämän terminologian mukaan muuttujan arvon päivitys muodostaa fokaalirivin, joka on osa muuttuja-menetelmää. Mikäli kyseinen menetelmä on hallinnassa, ei kyseinen rivi, eli tässä tapauksessa yksinkertaisen tuoreimman säilyttäjän arvon päivitys, unohtuisi pois ohjelmasta. Tapauksessa (a) on kyse ohjelman sovellusalueen ymmärtämiseen ja tapauksessa (c) kontrollirakenteisiin liittyvästä ongelmasta. Tapaukselle (b) ei ole selkeää selitystä. Eniten virheitä tässä suhteessa teki rooliryhmä, joka saavutti heikoimmat tulokset kahdessa ensimmäisessä muuttujiin liittymättömissä tehtävissä, joiden perusteella ryhmien taitotaso arvioitiin. On mahdollista, että ohjelman kirjoittamistehtävä oli kyseisen ryhmän kykyihin nähden niin vaativa, että syy päivityksen puuttumiselle oli lyhytkestoisen muistin ylikuormitus. Taulukossa 12 on koottu potentiaaliset syyt ohjelman kirjoittamisessa tehdyille virheille.

Taulukko 12: Muuttujiin liittyvien virheiden syyt ohjelman kirjoittamisessa Sajaniemeä & Kuittista (in press) mukaillen.

Muuttujan käyttöyhteys	Ryhmä		
	Perinteinen	Rooli	Animaatio
Yksinkertainen	Kontrolli-rakenteet	Kontrolli-rakenteet Sovellus-alueen ym-märtäminen	Kontrolli-rakenteet Sovellus-alueen ym-märtäminen
Monimutkainen	Kontrolli-rakenteet Menetelmä-tietämys	Kontrolli-rakenteet Menetelmä-tietämys	Kontrolli-rakenteet

Kaikilla ryhmillä oli ongelmia kontrollirakenteiden suhteen niin yksinkertaisissa kuin monimutkaisissa käyttötapauksissa. Animaatioryhmä vaikutti omaavan parhaan mene-

telmätietämyksen, sillä heillä oli hallussaan tarpeellinen muuttujiin liittyvät menetelmät (eli roolit) ja he sovelsivat omaamaansa tietämystä myös käytännössä. Sekä rooliryhmällä että animaatioryhmällä oli kuitenkin ongelmia sovellusalueen ymmärtämisessä, eikä sille ole löytynyt selkeää syytä.

5.4.4 Roolinimien käyttö

Myös roolinimien käyttö tenttivastauksissa tutkittiin. Kuten oli odotettavissa, perinteinen ryhmä ei käyttänyt roolien nimiä koevastauksissaan. Niin rooliryhmään kuin animaatioryhmään kuuluvista koehenkilöistä 35 % käytti tenttivastauksissaan roolien nimiä. Muuttujien roolit tunnistettiin tavallisesti oikein lukuun ottamatta kahden rooliryhmään kuuluneen koehenkilön tekemiä virheitä.

5.5 Pohdintaa

Lopputentissä olleiden kysymysten ratkaisemiseksi piti osata paitsi simuloida ohjelman toimintaa, myös ymmärtää lähdekoodia sekä kirjoittaa ohjelma itse. Vaikka eroavaisuudet ryhmien välillä eivät useimmiten olekaan tilastollisesti merkittäviä, vaikuttaa siltä, että muuttujien roolien vaikutus heijastuu eri tavoin ohjelmoinnin eri osa-alueilla. Seuraavassa on pohdittu mahdollisia eroavaisuuksien syitä näiden osa-alueiden eli ohjelman toiminnan simuloimisen, ymmärtämisen ja kirjoittamisen välillä.

5.5.1 Ohjelman toiminnan simulointi

Ohjelman toiminnan simulointitehtävässä eroavaisuudet ryhmien välillä ovat pienemmät kuin kahdessa muussa tehtävätyypissä. Perinteinen ryhmä ja rooliryhmä selvisivät tehtävästä yhtä hyvin, kun taas animaatioryhmä menestyi aavistuksen verran heikommin. Koska simulointitehtävässä olleen ohjelman toiminta oli tarkoituksella vaikeaselkoinen, oli käytännössä ainoa keino selvittää sen antamat tulosteet simuloimalla sen toimintaa askel askeleelta. Vaikka muuttujien roolit on helppo löytää kyseisessä ohjelmassa, ei muuttujien arvoissa tapahtuvia päivityksiä ollut helppo ennustaa. Tämän vuoksi on luonnollista, että muuttujien rooleista ei tämän tehtävän yhteydessä ollut apua. Kuittisen & Sajaniemen (2003) mukaan Jehng & al. (1999) ovat tutkineet visualisoinnin vaikutusta rekursion oppimisessa. He havaitsivat pienempiä eroja tehtävissä,

joissa koehenkilöiden piti ennustaa ohjelmien toiminnan tulosta kuin tehtävissä, jossa piti kirjoittaa ohjelma. Muuttujien rooleihin liittyvä koe päätyi myöskin samaan lopputulokseen.

5.5.2 Ohjelman ymmärtäminen

Ohjelman ymmärtämistehtävässä, jossa koehenkilöiden tuli selittää ohjelman toimintaa, rooliryhmä selviytyi parhaiten, animaatioryhmän jäädessä kolmannelle sijalle. Sama suuntaus oli yllättäen nähtävissä myös tenttivastauksista, joissa koehenkilöt olivat osoittaneet täydellistä ohjelman toiminnan ymmärtämistä. Ohjelmakuvausten analyysin perusteella havaittiin, että rooliryhmän antamat kuvaukset olivat yksityiskohtaisempia ja sisälsivät matalamman tason tietoutta kuin animaatioryhmän kuvaukset. Matalan tason kuvausta sisältävien lauseiden osuus vastauksissa korreloi tenttipisteiden kanssa positiivisesti kun taas korkean tason kuvaukset korreloivat negatiivisesti. Opettajat antoivat siis parempia tenttipisteitä lähdekoodin toimintaa rivi riviltä kuvaavista vastauksista verrattuna vastauksiin, joissa tarkasteltiin ohjelman korkeamman tason toimintaa ja ei suoraan ohjelman lähdekoodista näkyviä asioita. Koska animaatioryhmä painotti vastauksissaan korkeamman tason informaatiota, heidän tehtävästä saamansa pisteet olivat vastaavasti matalia. Näin siitä huolimatta, että korkeamman tason kuvausten on todettu liittyvän parempaan lähdekoodin ymmärtämiseen (Pennington, 1987) ja esimerkiksi Clancy ja Linn (1999) ovat todenneet, että koodin uudelleenkäyttö oli merkittävästi yleisempää opiskelijoilla, jotka käyttivät ohjelmia kuvatessaan korkeamman abstraktiotason kuvauksia.

Eroavaisuudet ohjelman toiminnan kuvauksessa voidaan selittää toisistaan poikkeavilla ohjelmistoilla, joita koehenkilöt käyttivät harjoitusten yhteydessä ohjelmien animoinnissa. Rooliryhmän ja animaatioryhmän harjoituksia vetäneen opettajan mukaan PlanAnia käyttäneet koehenkilöt keskittyivät enemmän muuttujien tarkasteluun, kun taas debuggerin käyttäjät keskittyivät enimmäkseen ohjelman lähdekoodin seuraamiseen (Sajaniemi & Kuittinen, 2003). Vaikka PlanAni-animaattorikin korostaa sillä hetkellä animoimansa kooodinpätkän, eivät koehenkilöt tuntuneet seuraavan lähdekoodia. Tämän seurauksena debuggerin käyttäjät ymmärsivät koodia yksityiskohtaisesti kun taas PlanAnia käyttäneet koehenkilöt keskittyivät ohjelman kokonaistoiminnan ymmärtämiseen ja siihen, kuinka yksittäiset muuttujat vaikuttivat kokonaisuuteen. Tämä on saattanut vaikuttaa siihen, miten PlanAnia käyttäneet koehenkilöt ajattelevat oh-

jelmien toimintaa: he ehkä pitävät muuttujien elinkaaria tärkeämpänä kuin ohjelman yksittäisiä toimintoja.

Ohjelmakuvausten analysoinnin perusteella oli nähtävissä, että rooliryhmällä ja animaatioryhmällä oli kattavampi sanasto kuvaamaan ohjelman toimintaa kuin perinteisellä ryhmällä. Lisäksi rooleihin liittyvää opetusta saaneet ryhmät antoivat enemmän ristiinviittauskuvauksia, kun perinteinen ryhmä painotti sovellusaluetason ja ohjelmatason kuvauksia. Vaikuttaa siltä, että rooleihin liittyvä tietämys on noviiseille tarpeen paremman ohjelmatietämyksen kehittämiseksi.

5.5.3 Ohjelman kirjoittaminen

Ohjelman kirjoittamistehtävässä rooliryhmä ja animaatioryhmä olivat parempia kuin perinteinen ryhmä. Myös tällä kertaa rooliryhmän pisteet olivat paremmat kuin animaatioryhmän, mutta lopputentistä saatujen pisteiden skaalaaminen saattaa osittain vaikuttaa tähän, sillä rooliryhmä hyötyi kyseisestä toimenpiteestä eniten.

Virheanalyysin mukaan animaatioryhmällä oli vähiten ongelmia muuttujiin liittyen ja heillä oli siihen liittyen paras menetelmätietämys. Animaatioryhmä osoitti parasta tietämystä myös ohjelmien syvärakenteisiin liittyen perinteisen ryhmän ollessa tässä suhteessa heikoin. Pintarakenteisiin liittyvässä tietämyksessä tilanne oli päinvastainen, mutta ryhmien väliset erot olivat pieniä. Vaikuttaa siltä, että perinteisellä ryhmällä oli pinnallisin mentaalinen malli kun taas animaatioryhmällä oli paras mentaalinen representaatio.

Mentaalisissa representaatioissa olevat erot voidaan selittää muuttujien rooleihin liittyvällä tietämyksellä ja roolipohjaisen animaattorin PlanAnin käytöllä. Muuttujien rooleihin liittyvä tietämyksen avulla noviisi voi saada ohjelmoinnista toisenlaisen kuvan perinteiseen opetustapaan verrattuna, jossa muuttujilla ja niiden käyttötavoilla ei itsessään ole mitään erityismerkitystä. Muuttujat saatetaan perinteisessä opetuksessa kokea ainoastaan epämääräisiksi tiedon säilytyspaikoiksi, joiden arvoja voidaan muokata. Muuttujien roolien avulla muuttujat nähdään aktiivisina toimijoina, johon ohjelman toiminta perustuu. PlanAnia käyttämällä roolien omaksuminen on helpompaa, koska tällöin rooleilla voidaan ajatella olevan konkreettinen visuaalinen ulkoasu ja käyttäytymistapa. Tämän voidaan olettaa helpottavan ohjelmien syvärakenteiden ymmärtämistä.

5.6 Johtopäätökset

Empiirisessä kokeessa tutkittiin muuttujien roolien ja roolipohjaisen animoinnin vaikutusta ohjelmoinnin alkeiden oppimiseen. Keskeisimpiä kysymyksiä ovat olleet, kuinka roolitietämyksen opettaminen vaikuttaa noviisien ohjelmointitaitoihin ohjelmien toiminnan simuloinnissa, ohjelmien ymmärtämisessä sekä ohjelmien kirjoittamisessa.

Tulokset osoittivat, että koehenkilöt pystyivät omaksumaan muuttujien roolit käsitteenä ja soveltamaan niitä uusissa tilanteissa. Kurssin jälkeen 35 % koehenkilöistä käytti muuttujien roolinimiä tenttivastauksissaan, vaikka kysymyksissä ei mainittu muuttujien rooleja millään tavalla. Muuttujien roolien avulla koehenkilöt saivat paitsi käyttöönsä sanaston ohjelman toiminnasta keskustelemiseen, myös välineet käsitellä ohjelmaan liittyvää tietoa tavalla, joka on tyypillistä hyville lähdekoodin ymmärtäjille.

Vaikuttaa siltä, että roolipohjaisen animaattorin PlanAnin käyttö edistää roolitietämyksen kehittymistä, sillä animaatioryhmän koehenkilöillä oli vähemmän vaikeuksia muuttujien käytössä muihin ryhmiin verrattuna ohjelman kirjoittamistehtävässä. Lisäksi PlanAnia käyttäneet henkilöt käyttivät ohjelmakuvauksissaan paljon ohjelman syväarakenteita kuvaavia lauseita, mikä on merkki hyvästä lähdekoodin ymmärtämisestä.

Ristiriita opettajien antamien tenttipisteiden ja koehenkilöiden todellisten ohjelmointitaitojen kanssa viittaa siihen, että opetuskokeilujen tulosten arviointi ei voi pohjautua pelkästään opiskelijoiden saamien tenttitulosten tarkasteluun. Lisäksi kannattaa keskittyä tenttivastausten ja mahdollisen muun kokeen yhteydessä kerätyn materiaalin tutkimiseen, sillä näistä voivat käydä ilmi erot koehenkilöiden mentaalisissa malleissa.

Ohjelman simulointitehtävässä roolien käyttämisellä ei näyttänyt olevan vaikutusta tenttituloksiin, mutta toisaalta simulointitehtävästä saatujen vastausten muoto oli sellainen, että niistä oli mahdotonta arvioida koehenkilöiden omaamia mentaalisia malleja. Kerätyn aineiston analysointia jatketaan (protokollatehtävät), ja tämän toivotaan edelleen selventävän sitä, millä tavalla muuttujien roolit vaikuttavat ohjelmointitaitojen kehittymiseen.

6 Yhteenveto

Tässä työssä on tarkasteltu oppimisprosessia, ohjelmoinnin alkeiden oppimista ja opettamista sekä esitelty uusi ohjelmoinnin alkeiden opettamiseen soveltuva käsite muuttujien roolit, roolipohjainen animaattori PlanAni sekä näiden hyödyntämiseen ohjelmoinnin alkeiden opetuksessa liittyvä empiirinen koe.

Behavioristisen oppimiskäsityksen mukaan oppimisessa on kyse ennen kaikkea valmiin tiedon siirtämisestä. Tämä saattaa olla riittävä lähtökohta opeteltaessa esimerkiksi eri maiden pääkaupunkeja, mutta jos opittavaa asiaa (kuten vaikkapa ohjelmointia) pitää pystyä aktiivisesti soveltamaan uusissa tilanteissa, ei tiedon ulkoa opettelu ole toimiva oppimistapa. Luonnontieteellisten aineiden ja myös ohjelmoinnin opettamisessa ja oppimisessa konstruktivistinen lähestymistapa tarjoaakin oppimiselle paremman lähtökohdan. Konstruktivistisessa oppimiskäsityksessä ihminen nähdään aktiivisena toimijana, joka hankkimansa tietämyksen ja kokemuksen myötä muokkaa omia sisäisiä mallejaan opittavasta asiasta.

Ihmisen ulkopuolelta tuleva informaatio päätyy alustavan käsittelyn jälkeen sensorisiin muisteihin, joissa se odottaa mahdollista jatkoprosessointia. Jotta kyseinen informaatio pääsisi lyhytkestoiseen muistiin, tarvitaan tarkkaavaisuutta. Tarkkaavaisuutta käyttämällä informaatio saadaan käyttöön lyhytkestoiseen muistiin, jossa sitä voidaan käsitellä tietoisesti. Lyhytkestoisessa muistissa voi olla kerralla tietoa rajallinen määrä, arvioitu maksimimäärä vaihtelee kolmen ja yhdeksän mieltämisyksikön välillä.

Ihmisellä on pitkäkestoisessa muistissa tallennettuna siihen mennessä koetun pohjalta joukko hierarkkisia tietämysrakenteita eli mentaalisia malleja (tai skeemoja) erilaisista asioista. Pitkäkestoisessa muistissa olevaan tietoon voidaan tehdä hakuja ja sinne voidaan myös tallentaa uutta tietoa. Mikäli ongelmanratkaisun yhteydessä pitkäkestoisessa muistissa ei ole tarjolla valmista tai tilanteeseen soveltuvaa ratkaisumallia (=skeemaa) tai muistihaku epäonnistuu, voidaan apuna käyttää ongelmanratkaisuheuristiikkoja. Ulkopuolelta tulevaa informaatiota ja pitkäkestoisessa muistissa olevaa tietoa voidaan käsitellä yhtä aikaa lyhytkestoisessa muistissa tarkkaavaisuutta apuna käyttäen. Harjoittelun avulla taidot automatisoituvat, kyky käyttää tietämystä paremmin hyväksi kasvaa ja asiaan liittyvät tietämysrakenteet muistissa paranevat. Samalla tieto muuttuu deklaratiivisesta tiedosta vähitellen proseduraaliseksi tiedoksi. Kun asiaa tarkastellaan taidon oppimisen kannalta, sama prosessi voidaan jakaa kognitiiviseen,

assosiatiiviseen ja autonomiseen vaiheeseen.

Monimutkaisten asioiden omaksuminen ja oppiminen vaatii aikaa, vaivaa, kiinnostusta ja tietoista harjoittelua. On kyseessä sitten shakin pelaaminen, uimahyppyjen tekeminen, taitoluistelu, ohjelmointi tai mikä tahansa muu ala, ei oikotietä noviisista ekspertiksi ole olemassa. Tosiasia on, että esimerkiksi Java-ohjelmointikieltä ei voi oppia 24 tunnissa, vaikka tämän tyyppinen kirjallisuus ja ennen kaikkea taustalla vaikuttava lyhytjänteinen ajattelutapa on nykyään pelottavan yleistä.

Ohjelmointi on taito, joka jakautuu moniin eri osa-alueisiin. Käytännön työelämässä lähdekoodin suunnittelu, kirjoittaminen, jo olemassa olevan koodin ymmärtäminen ja tarvittavien muutosten tekeminen limittyvät usein toinen toisiinsa. Ohjelmoinnin eri osa-alueita ja ohjelmoinnin oppimista on tutkittu laajalti. Keskeisiä esiteltyjä malleja ja tutkimuksia ohjelmointiin liittyen ovat tavoite/menetelmä-malli ja sen laajennos fokaaliriveistä ja menetelmien luomisesta, parsing–gnisrap-malli, Brooksin malli sekä Penningtonin tekemät havainnot ohjelmatietämyksen lajeista.

Noviisien ongelmat ohjelmoinnissa eivät enimmäkseen johdu siitä, että he eivät tuntisi ohjelmointikielen syntaksia, vaan puuttuvasta kyvystä soveltaa tätä tietämystä eri tilanteissa. Koska noviiseilla ei ole ehtinyt kehittyä tarvittavaa menetelmätietämystä, on ongelmanratkaisu aloitettava useimmiten puhtaalta pöydältä kun taas ekspertti voi käyttää apunaan jo omaamaansa tietämystä ja soveltaa sitä uusiin tilanteisiin. Ohjelmoinnissa tarvittava tietämys voidaan saavuttaa vain harjoittelemalla erilaisia ohjelmointiin liittyviä taitoja. Harjoittelun tuloksena ovat aiempaa kehittyneemmät mentaaliset mallit.

Ohjelmoinnin alkeita voidaan opettaa hyvin monella eri tavalla. Periaatteessa mikä tahansa looginen ja järjestelmällinen opetustapa, joka pakottaa opiskelijan aktiivisesti kehittämään ohjelmointiin liittyviä mentaalisia mallejaan, voi olla opiskelijalle avuksi. Sisällölliset uudistukset ohjelmoinnin alkeiskursseihin liittyen ovat olleet viime aikoina harvinaisia, sillä useimmiten pääpaino on ollut sopivan tiedon esittämistavan miettimisessä. Tästä poikkeuksena ovat suunnittelumallit, joita voidaan hyödyntää myös ohjelmoinnin alkeiden opettamisessa. Suunnittelumallien käyttö ohjelmoinnin alkeiden opettamisessa vaatii kuitenkin huolellista suunnittelua. On syytä muistaa, että ketään ei voi pakottaa oppimaan. Ohjelmoinnin opettamisen ja oppimisen yhteydessä tämä näkyy siten, että mikäli motivaatio oppia puuttuu, eivät odotettavissa olevat tulokset ole hyviä, on opetustapa millainen hyvänsä.

Muuttujien roolit on kognitiivinen apukeino, jota hyödyntämällä voidaan välittää noviiseille eksperttien piilevässä muodossa olevaa ohjelmointiin liittyvää tietämystä. *Muuttujan roolilla tarkoitetaan muuttujan dynaamista luonnetta, joka ilmenee sarjana muuttujan perättäisiä arvoja suhteessa muihin muuttujiin ja ulkoisiin tapahtumiin*. Muuttujien rooleja on kymmenen kappaletta: *kiintoarvo*, *askeltaja*, *tuoreimman säilyttäjä*, *sopivimman säilyttäjä*, *kokooja*, *yksisuuntainen lippu*, *seuraaja*, *tilapäissäilö*, *järjestelijä* ja *muunnos*. Muuttujat, jotka eivät kuulu yllämainittuihin rooleihin, luokitellaan roolijaossa kategoriaan muu, mutta noviisitason Pascal-kielellä kirjoitetuissa ohjelmissa olevista muuttujista 99% voidaan luokitella kymmenen edellä mainitun roolin avulla. Muuttujien roolit eivät ole teknisiä, vaan kognitiivisia käsitteitä, minkä seurauksena kaksi eri ihmistä voi pitää saman ohjelman samaa muuttujaa eri roolin edustajana, ja molemmat saattavat olla oikeassa. Muuttujan rooli voi myös vaihtua ohjelman suorituksen aikana esimerkiksi *tuoreimman säilyttäjästä askeltajaksi*.

PlanAni on roolipohjainen animaattori, jota voidaan hyödyntää ohjelmoinnin alkeiden opettamisessa, kun halutaan visualisoida muuttujien rooleja ja muuttujien käyttöön liittyvää ohjelman sisäistä toimintalogiikkaa. PlanAnia käyttämällä voidaan konkretisoida muuttujien rooleja ja niiden käyttäytymistä, ja antaa hieman luennoista poikkeavia kognitiivisia ärsykkeitä, mikä osaltaan helpottaa muuttujien roolien ymmärtämistä käsitteenä.

Muuttujien rooleihin liittyvää tietämystä tulee opettaa konstruktivistiseen oppimiskäsitykseen pohjautuvalla tavalla, eli kasvattamalla tietämystä vähitellen jo ennestään olemassa olevan tiedon päälle. Tapoja rooleihin liittyvä tietämyksen vahvistamiseksi on monia, mutta parhaimmat tulokset saavutetaan, kun opetettavaan asiaan liitetään oppijan kannalta merkityksellistä lisätietoa. Luennoilla läpi käytävien esimerkkiohjelmien yhteydessä ei esimerkiksi riitä, että luetellaan ohjelmassa olevien muuttujien roolit, vaan on syytä tuoda esille, kuinka ohjelmassa oleva muuttuja ilmentää tietyn roolin käyttäytymistä. Muita keskeisiä tapoja ovat roolien mainitseminen toistuvasti erilaisissa, merkityksellisissä asiayhteyksissä, muuttujien rooleihin liittyvät keskustelut opiskelijoiden kanssa sekä PlanAni-animaattorin käyttäminen.

Muuttujien roolit tarjoavat ohjelmoinnin alkeiden opettamiseen perinteisestä opetustavasta poikkeavan lähestymistavan. Muuttujien roolien ansiosta noviisit saavat lisäksi käyttöönsä kompaktin sanaston ohjelman toimintaperiaatteiden kuvaamiseen ja ymmärtämiseen. PlanAni-animaattoria käyttämällä roolien käyttäytymistä ja niiden toimintaa voidaan konkretisoida visuaalisesti. Yhdessä käytettyinä ne näyttävät helpotta-

van ohjelmien syvärakenteiden ja muuttujien toiminnan ymmärtämistä. Moniin muihin ohjelmoinnin opetustapoihin nähden muuttujien rooleilla on vielä yksi etu: niiden hyödyntäminen opettamisessa ei tarkoita sitä, että olemassa oleva opetusmateriaali pitäisi laatia kokonaan uudelleen. Muuttujien rooleja voidaan käyttää apuna tekemällä materiaaleihin ainoastaan pieniä muutoksia, ja opetettavia asioita läpi käytäessä tulee muistaa mainita muuttujien rooleihin liittyvä näkökulma.

Empiirinen koe osoitti muuttujien roolien ja PlanAnin soveltuvan hyvin ohjelmoinnin alkeiden opetukseen. Opiskelijat pystyivät ymmärtämään muuttujien roolit käsitteenä ja hyödyntämään sitä uusissa tilanteissa. Roolien avulla opiskelijat saivat paitsi käyttöönsä sanaston ohjelmien toiminnan kuvaamiseen, myös tarvittavat välineet prosessoida ohjelmiin ja niiden toimintaan liittyvää informaatiota tavalla, joka on tyypillinen hyville lähdekoodin ymmärtäjille. Roolipohjaisen animaation käyttö näyttää helpottavan roolitietämyksen omaksumista, sillä PlanAnin käyttäjillä oli vähiten vaikeuksia muuttujien käyttämisessä ohjelmien kirjoittamistehtävän yhteydessä. Lisäksi PlanAnin käyttäjät painottivat ohjelmakuvauksissaan ohjelman syvärakenteita, mikä on merkki hyvästä lähdekoodin ymmärtämisestä. Roolipohjaisen animaation käyttäjien syvemmän tason tietämys ohjelmoinnista ei näkynyt heidän ohjelmakuvauksissa saaduista tenttipisteissä. Tämä johtui siitä, että arvostelussa suosittiin matalan tason kuvauksia, joissa keskityttiin ohjelman pintarakenteiden selittämiseen.

Ohjelmointi ja sen alkeiden opetus on aiheena erittäin laaja-alainen ja se sivuaa monia eri tieteenaloja. Tästä oli seurauksena runsaudenpula sen suhteen, mitä asioita ja missä laajuudessa tähän työhön tulisi sisällyttää. Periaatteellisella tasolla on tarjolla kaksi ääripäätä: kuvataan tarkasti muutamaa, tarkoituksella rajattua asiaa tai sitten tarkastellaan hieman kaikkia aiheeseen liittyviä asioita, mutta ei niin tarkasti. Molemmissa ääripäissä on omat hyvät ja huonot puolensa. Tekijän mielestä oli olennaista paitsi kertoa muuttujien rooleista, myös saada selkeä kokonaiskuva oppimiseen käytettävissä olevista resursseista sekä ohjelmoinnin oppimiseen ja opettamiseen liittyvistä asioista. Kaikkia työssä olevia asioita voitaisiin käsitellä paljon laajemminkin, ja samoin on mahdollista, että tärkeitä asioita on jäänyt pois tai liian vähälle huomiolle. Tarkemmin olisi voitu käsitellä esimerkiksi visualisointiin liittyviä asioita, mutta syvällisempi käsittely olisi vienyt melkoisen määrän tilaa.

Ohjelmoinnin psykologia ja ohjelmoinnin empiirinen tutkimus tarjoavat valtavan määrän potentiaalisia tutkimuskohteita. On syytä huomauttaa, että useimmat ohjelmointiin liittyvät tutkimukset, kuten myös tässä työssä esitelty muuttujien rooleihin liittyvä

tutkimus, tarkastelevat ohjelmoinnin oppimista proseduraalisen ohjelmoinnin kannalta. Pääasiassa tutkimusten yhteydessä on ohjelmointikielenä ollut Pascal. Vaikka esimerkiksi oliopohjainen paradigma on jo vakiinnuttanut asemansa, on syvällistä tutkimustietoa sen käyttämisestä ohjelmoinnin alkeiden opettamisessa ja oppimisessa vieläkin melko vähän saatavilla. Tässä mielessä muuttujien roolien hyödyntäminen olio-ohjelmoinnin opettamisen yhteydessä on potentiaalinen tutkimuksen kohde.

Viitteet

Anderson, J. R. (1983) *The Architecture of Cognition*. Harvard University Press, Cambridge, Massachusetts.

Anderson, J. R. (1985) *Cognitive Psychology and Its Implications, Second Edition*. W.H.Freeman and Company, New York.

Astrachan, O., Reed, D. (1995) The Applied Apprenticeship Approach to CS 1. *Proc. of the twenty-sixth SIGCSE technical symposium on computer science education* ACM Press, New York, 1-5.

Baecker, R. (1988) Enhancing Program Readability and Comprehensibility with Tools for Program Visualization. *Proc. of the 10th international Conference on Software Engineering* IEEE Computer Society Press, Los Alamitos, California., 356-366

Ben-Ari, M. (2001) Constructivism in Computer Science Education. *Journal of Computers in Mathematics and Science Teaching* **20**, 45-73.

Bonar, J., Soloway, E. (1985) Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human-Computer Interaction* **1**, 133-161.

Broadbent, D. E. (1975) The magic number after fifteen years. *Studies in Long-Term Memory* (toim. Kennedy, R. A., Wilkes, A.), Wiley, New York.

Brooks, R. E. (1983) Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies* **18**, 543-554.

Clancy, M. J., Linn, M. C. (1999) Patterns and Pedagogy. *Proc. of the 30th SIGCSE Technical symposium on CS Education, ACM SIGCSE Bulletin.* **31** (1), 37-42.

Davies, S. P. (1993) The Structure and Content of Programming Knowledge: Disentangling Training and Language Effects in Theories of Skill Development. *International Journal of Human-Computer Interaction* **5** (4), 325-346.

Détienne, F. (2002) *Software Design - Cognitive Aspects*. Springer-Verlag.

Ellis, A., Tarhio, J., Carswell, L., Bernat, A., Deveaux, D., Frison, P., Meisalo, V., Meyer, J., Nulden, U., Rugelj, J. (1998) Resources, tools, and Techniques for Problem Based Learning in Computing. *ACM SIGCUE Outlook Special issue on the working*

group reports of the 3rd annual SIGCES/SIGCUE ITiCSE conference ACM Press, New York, 41-56.

Eteläpelto, A. (1996) Asiantuntijuuden kehittyminen; miten tullaan noviisista ekspertiksi? *Psykologia, johdantokurssi* (toim. Sarkkinen, M.), Gummerus Kirjapaino Oy, Jyväskylä, 48-83.

Fincher, S. (1999) What are we doing when we teach programming? *Proc. of the 29th ASEE/IEEE Frontiers in Education Conference* San Juan, Puerto Rico, Session 12a4, 1-5.

Fitts, P. M., Posner, M. I. (1967) *Human Performance* Brooks Cole, Belmont, California.

Gellenbeck, E. M., Cook, C. R. (1991) An Investigation of Procedure and Variable Names as Beacons during Program Comprehension. *Empirical Studies of Programmers: Fourth Workshop* (toim. Koenemann-Belliveau, J., Moher, T. G., Robertson, S. P.), Ablex Publishing Company, Norwood, New Jersey, 65-81.

Glaser, R., Chi, M. T. H. (1988) Overview. Chi, M. T. H., Glaser, R., Farr, M. J. (toim.) (1988) *The Nature of Expertise*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, xv-xxviii.

Good, J., Brna, P. (2003) Towards authentic measures of program comprehension. *EASE & PPIG 2003* Papers from the Joint Conference at Keele University 8th-10th April 2003. Keele, U.K., 29-49.

Green, T. R. G., Bellamy, R. K. E., Parker, J. M. (1987) Parsing and Gnisrap: A Model of Device Use. *Empirical Studies of Programmers: Second Workshop* (toim. Olson, G. M., Sheppard, S., Soloway, E.), Ablex Publishing Company, Norwood, New Jersey, 132-146.

Hillside.net (2004) Home of the Patterns Library. WWW-sivusto, `http://hillside.net` (6.3.2004)

Hundhausen, C. D., Douglas, S. A., Stasko, J. D. (2002) A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing* **13**, 259-290.

Jehng, J-C. J., Tung, S-H. S., Chang, C-T. (1999) A Visualization Approach to Lear-

ning the Concept of Recursion. *Journal of Computer Assisted Learning* **15**, 279-290.

Jenkins, T. (2001) Teaching Programming - A Journey from Teacher to Motivator. *Proc. of the 2nd annual LTSN-ICS conference* LTSN Centre for Information and Computer Sciences.

Kids' Club (2004) *Tools in Kid's Club*. WWW-sivusto, `http://cs.joensuu.fi/~kidsclub/pages/tools.htm` (20.2.2004).

Kuittinen, M., Sajaniemi, J. (2003) First Results of An Experiment on Using Roles of Variables in Teaching. *EASE & PPIG 2003* Papers from the Joint Conference at Keele University 8th-10th April 2003. Keele, U.K., 347-357.

Kuittinen, M., Sajaniemi, J. (2004) Teaching Roles of Variables in Elementary Programming Courses. Proc. of the 9th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2004), Leeds UK, 28-30 June 2004., 57-61.

Lobo, A. F., Baliga, G. R., Bergmann, S., Stone, D., Shah, A. (2000) Using Real-world Objects to Motivate OOP in a CS1 Lab. *Proc. of the fifth annual CCSC northeastern conference on The journal of computing in small colleges* The Consortium for Computing in Small Colleges, USA, 144-156.

Microsoft MSDN (2003) *Hungarian Notation*. WWW-sivusto, `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvsgen/html/hunganotat.asp` (10.12.2003).

Miller, G. A. (1956) The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review* **63**, 81-97.

Nguyen, D. (1998) Design patterns for data structures. *Proc. of the 29th SIGCSE Technical Symposium on CS Education*. ACM SIGCSE Bulletin, **30** 336-340.

Norman, D. A. (1983) Some Observations on Mental Models. *Mental Models* (toim. Gentner, D., Stevens, A. L.), Erlbaum, Hillsdale, New Jersey, 7-14.

Pennington, N. (1987) Comprehension Strategies in Programming. *Empirical Studies of Programmers: Second Workshop* (toim. Olson, G. M., Sheppard, S., Soloway, E.), Ablex Publishing Company, Norwood, New Jersey, 100-113.

Perkins, D. N., Martin, F. (1986) Fragile Knowledge and Neglected Strategies in No-

vice Programmers. *Empirical Studies of Programmers* (toim. Soloway, E., Iyengar, S.), Ablex Publishing Company, Norwood, New Jersey, 213-229.

Petre, M., Blackwell, A. F. (1999) Mental Imagery in Program Design and Visual Programming. *International Journal of Human-Computer Studies* **51** (1), 7-30.

Poon, J. (2000) Java Meets Teletubbies: An Interaction Between Program Codes and Physical Props. *Proc. of the Australasian conference on Computing education* ACM Press, New York, 195-202.

Rist, R. S. (1989) Schema Creation in Programming. *Cognitive Science* **13**, 389-414.

Sajaniemi, J. (2002) An Empirical Analysis of Roles of Variables in Novice-Level Procedural Programs. *Proc. of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)* IEEE Computing Society, 37-39.

Sajaniemi, J. (2003) *Roles of Variables Home Page*. WWW-sivusto, `http://www.cs.joensuu.fi/~saja/var_roles/` (15.12.2003).

Sajaniemi, J. (2004a) Sähköpostiviesti 19.2.2004.

Sajaniemi, J. (2004b) Sähköpostiviesti 12.3.2004.

Sajaniemi, J. (2004c) Sähköpostiviesti 6.5.2004.

Sajaniemi, J., Kuittinen, M. (2003) Program Animation Based on the Roles of Variables. *Proceedings ACM 2003 Symposium on Software Visualization (SoftVis 2003)* San Diego, CA, June 2003. Association for Computing Machinery, 7-16.

Sajaniemi J., Kuittinen M. (in press) An Experiment on Using Roles of Variables in Teaching Introductory Programming. Computer Science Education.

Schneider, W., Shiffrin, R. M. (1977) Controlled and Automatic Human Information Processing: 1. Detection, search, and attention. *Psychological Review* **84** (1), 1-66.

Seel, N. (2001) Epistemology, situated cognition, and mental models: "Like a bridge over troubled water". *Instructional Science* **29** (4/5), 403-427.

Shih, Y., Alessi, S. M. (1994) Mental Models and Transfer of Learning in Computer Programming. *Journal of Research on Computing In Education* **26** (2), 154-175.

Simon, H. A. (1974) How big is a chunk? *Science* **183**, 482-488.

Soloway, E., Ehrlich K., Bonar J., Greenspan, J. (1982) What Do Novices Know about Programming? (toim. Badre, A., Shneiderman, B.), *Directions in Human/Computer Interaction* Ablex Publishing Company, 27-54.

Spohrer, J. C., Pope, E., Lipman, M., Sack, W., Freiman, S. Littman, D., Johnson, L., Soloway, E. (1985) Bugs in Novice Programs and Misconceptions in Novice Programmers. *Computers in Education* Elsevier Science Publishers, 543-552.

Spohrer, J. C., Soloway, E. (1986) Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM* **29** (7), 624-632.

Tynjälä, P. (2000) *Oppiminen tiedon rakentamisena: konstruktivistisen oppimiskäsityksen perusteita* Tammer-Paino Oy, Tampere.

Wallingford, E. (2004) The elementary patterns home page. WWW-sivusto, `http://www.cs.uni.edu/~wallingf/patterns/elementary/` (5.3.2004.)

Wilson, J. D. (1987) A Socratic Approach to Helping Novice Programmers Debug Programs. *Proc. of the eighteenth SIGCSE technical symposium on Computer science education* ACM Press, New York, 179-182.

Liite 1: Esitietolomake

ESITIETOLOMAKE

Ohjelmoinnin alkeet

3. lokakuuta 2002

Nimi: _____

Opiskelijanumero: _____

Sähköpostiosoite: _____

Sukupuoli: Mies Nainen

Kouluarvosanat lukion päästötodistuksesta tai vastaavista (merkitse viiva, jos arvosanaa ei ole).

	lukion päästö-todistuksen arvosana	TAI muun päästötodistuksen arvosana
äidinkieli		
matematiikka		
tietotekniikka		
kuvaamataito		

Jos arvosanasi ovat muualta kuin lukiosta, niin mistä?

Oletko käyttänyt taulukkolaskentaa?
Jos kyllä, niin montako taulukkoa olet tehnyt?

Oletko osallistunut yhdelle tai useammalle ohjelmointia käsittelevälle kurssille?
Jos kyllä, niin mille kursseille ja kuinka pitkiä kurssit ovat olleet?

Vastaa seuraaviin kysymyksiin vain, mikäli olet ohjelmoinut aiemmin.
Mitä ohjelmointikieliä olet käyttänyt?

Kuinka monta vuotta olet tehnyt ohjelmia?

Kuinka monta ohjelmaa olet tehnyt seuraavissa ohjelmien kokoluokissa? Merkitse yksi rasti kullekin riville.

	ei yhtään	1-10 ohjelmaa	yli 10 ohjelmaa
alle 50 rivisiä			
50-300 rivisiä			
yli 300 rivisiä			

Liite 2: Lopputenttikysymykset

OHJELMOINNIN ALKEET, LOPPUKUULUSTELU 11.12.2002

0. Arvioi asteikolla 0-40 %, 40-70 %, 70-100 %, kuinka usein olit läsnä a) luennoilla ja b) harjoituksissa. (Tästä tehtävästä saa 1 pisteen.)

1. Kerro, millaisia eri rakenteita Pascal-kielessä on toimintojen toistamiseen ja millaisissa tilanteissa kutakin rakennetta on sopivinta käyttää.

2. Tarkastellaan seuraavia syntaktisia sääntöjä:

```
juttu = [ a | b ] a
a = ”)” | ”:”
b = { k n }
k = ”x” | ”y” | ”z”
n = ”1” | ”2” | ”3”
```

   Mitkä seuraavista saadaan välikesymbolista `juttu` käyttämällä edellä olevia sääntöjä? Kirjoita lyhyt perustelu kullekin vastauksellesi.

   a) `:`
   b) `:)`
   c) `:::`
   d) `xyz:`
   e) `x1:`
   f) `x1y2z3)`
   g) `x1y2z2:`
   h) `x1y2z3):`

3. Mitkä luvut seuraava ohjelma tulostaa, kun sille annetaan syötteeksi luku 18?

```
program tehtava3 (input,output);
var j,k,n,s: integer;
    a: array [2..100] of Boolean;
begin
     write('Mihin saakka: '); readln(n);
     s:=0;
     for j:=2 to n do a[j]:=true;
     for j:=2 to n do
         if a[j] then
            begin
            s:=s+j; writeln(j, ' ', s);
            k:=j;
            while k <= n do begin a[k]:=false; k:=k+j end
            end
end.
```

4. Selitä, mikä on seuraavan ohjelman tarkoitus ja kuinka ohjelma toimii.

```
program tehtava4 (input, output);
var paino,p,a,m: integer;
begin
     m := 0;
     write('Anna potilaan paino (kg): '); readln(paino);
     for p := 1 to 3 do
         begin
         a := paino * 3;
         writeln('Päivän ', p, '. aamulla ja illalla ', a, ' ml.');
         m := m + 2*a
         end;
     for p := 4 to 7 do
         begin
         a := paino * 4;
         writeln('Päivän ', p, '. aamulla ', a, ' ml.');
         m := m + a
         end;
     writeln(m)
end.
```

5. Kirjoita Pascal-kielinen ohjelma, joka tulostaa saamiensa syötteiden perusteella tiedon oppilaan kelpoisuudesta osallistua välikokeeseen. Ohjelma saa syötteinä harjoituskertojen lukumäärän ja tehtävien kokonaismäärän sekä kullekin oppilaalle kunkin harjoituskerran tehtyjen tehtävien määrän. Ohjelma tulostaa kunkin oppilaan jälkeen tiedon tämän oikeudesta osallistua välikokeeseen. Välikokeeseen saa osallistua, jos vähintään 1/3 annetuista tehtävistä on tehty harjoituksissa.
   Seuraavassa on esimerkki ohjelman tulostuksesta.

```
Välikoekelpoisuuden tarkistaminen.

Anna harjoituskertojen lukumäärä ja tehtävien kokonaismäärä: 0 4
Syötteissä on virhe.
Anna harjoituskertojen lukumäärä ja tehtävien kokonaismäärä: 5 4
Syötteissä on virhe.
Anna harjoituskertojen lukumäärä ja tehtävien kokonaismäärä: 5 24

Anna 1. oppilaan tiedot:
   Anna 1. kerralla tehtyjen tehtävien määrä: 3
   Anna 2. kerralla tehtyjen tehtävien määrä: 0
   Anna 3. kerralla tehtyjen tehtävien määrä: 1
   Anna 4. kerralla tehtyjen tehtävien määrä: 0
   Anna 5. kerralla tehtyjen tehtävien määrä: 2
Oppilas ei saa osallistua välikokeeseen.

Anna 2. oppilaan tiedot:
   Anna 1. kerralla tehtyjen tehtävien määrä: 3
   Anna 2. kerralla tehtyjen tehtävien määrä: -2
Syötteessä on virhe.
   Anna 2. kerralla tehtyjen tehtävien määrä: 2
   Anna 3. kerralla tehtyjen tehtävien määrä: 2
   Anna 4. kerralla tehtyjen tehtävien määrä: 3
   Anna 5. kerralla tehtyjen tehtävien määrä: 4
Oppilas saa osallistua välikokeeseen.

Anna 3. oppilaan tiedot:
   Anna 1. kerralla tehtyjen tehtävien määrä: 0
   Anna 2. kerralla tehtyjen tehtävien määrä: 0
   Anna 3. kerralla tehtyjen tehtävien määrä: 0
   Anna 4. kerralla tehtyjen tehtävien määrä: 0
   Anna 5. kerralla tehtyjen tehtävien määrä: 0
Ohjelman suoritus loppuu.
```