

hyväksymispäivä

arvosana

arvostelija

Ohjelmointikieli - ajattelun apuväline

Alexi Kallio

Helsinki 30.4.2003

Tutkielma

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Pelkkä tietotekniikka ei vastaa ohjelmointikielten rakenteen kannalta tärkeisiin kysymyksiin. Ohjelmointikielten suunnittelu on kehittynyt järkeilyn ja hyväksi havaittujen käytäntöjen avulla, joilla kuitenkin harvemmin on kognitiotieteellistä taustaa.

Luonnollinen tapa lähestyä ohjelmointikielten suunnittelua onkin etsiä ohjaavia periaatteita ihmisen ajattelun toiminnasta, koska ohjelmointikieli on kuitenkin aina inhimillisen ajattelun työkalu. Tässä tutkielmassa tarkastellaan klassisia ohjelmointikielten suunnitteluun liittyviä periaatteita ja uudempia ohjelmointiin liittyviä kysymyksiä, sekä sitä miten ihmisen ajattelua tutkivien tieteiden tuloksia voidaan soveltaa ohjelmointikielten suunnitteluun.

Lopuksi esitetään karkea näkemys siitä, millainen tarkastelun tuloksiin perustuva tulevaisuuden ohjelmointikieli voisi olla.

Sisältö

1	Johdanto	1
2	Tietokoneen arkkitehtuuri ja ohjelmointikielet	2
3	Yleisesti hyväksytyt suunnitteluperiaatteet	4
3.1	Yksinkertaisuuden ja geneerisyyden suhde	4
3.2	Abstraktioiden merkitys ohjelmoijalle	5
3.3	Ohjelman dynaaminen ja staattinen muoto	7
4	Olio-ohjelmointi	9
4.1	Olioparadigma hahmottamisen välineenä	9
4.2	Modularisointi vaatii ilmaisuvoimaa	10
5	Ohjelmointi ja ihmisen ajattelu	12
6	Monimutkaisuuden hallinta	15
6.1	Monimutkaisuus ja ihmisen muisti	15
6.2	Ohjelma on ohjelmointitehtävän yleistys	17
6.3	Ohjelmointityö dialogina	18
7	Visio hyvästä ajattelun apuvälineestä	20
8	Yhteenveto	23
	Lähteet	24

1 Johdanto

Ohjelmointikielten kehitys poikkeaa muun tieto- ja viestintäteknikan kehityksestä. Kymmenen vuotta vanha tietokone on käytännöllisesti katsoen käyttökelvoton, kun taas suurin osa käytössä olevista ohjelmointikielistä on yli kymmenen vuotta vanhoja. Monet tietokoneiden alkuaikoina 1950- ja 1960-luvulla kehitetyt kielet - kuten Fortran, Lisp, Cobol ja Basic - ovat nykyäänkin käytössä¹.

Henkilökohtainen tyytymättömyyteni nykyisiin ohjelmointikieliin sai minut perehtymään tarkemmin korkean tason ohjelmointikielten suunnitteluun. Kirjoitus tarkastelee korkean tason ohjelmointikielten suunnittelun taustalla olevia tavoitteita, kielten rakenteita ja ominaisuuksia sekä sitä miten hyvin kielet täyttävät tavoitteensa, siis toimivat ohjelmoijan työkaluina.

Aluksi perehdytään tietokoneen arkkitehtuurin ja ohjelmointikielten väliseen suhteeseen. Tarkastelussa havaitaan, että tietokoneen arkkitehtuurista ja matalan tason konekieliohjelmoinnista ei voida löytää ratkaisua korkean tason kielten suunnitteluun liittyviin ongelmiin. Aiheen käsittelyä jatketaan klassisten lähteiden pohjalta, joiden avulla löydetään neljä merkittävää hyvän ohjelmointikielen piirrettä.

Kun löydettyjä piirteitä tarkastellaan lähemmin, huomataan kuinka merkittävässä roolissa ohjelmoijan ajatusmaailma on ohjelmointikielen laatua arvioitaessa. Sama ohjelmoijan ajatusmaailman tärkeys havaitaan myös nykyaikaista olio-ohjelmointia tarkasteltaessa. On siis perusteltua tarkastella ohjelmointia luovan ajattelun prosessina.

Kognitiotieteen tulosten pohjalta pohditaan ohjelmointiin liittyviä inhimillisen ajattelun ulottuvuuksia: muistin toimintaa, näkökulmien huomioonottoa ja yleistysten rakentamista. Lopuksi esitän tehtyjen havaintojen pohjalta karkean näkemykseni siitä, millainen tulevaisuuden ohjelmointikieli voisi olla.

¹Ks. http://www.wikipedia.org/wiki/Programming_language_timeline (tarkistettu 24.2.2003).

2 Tietokoneen arkkitehtuuri ja ohjelmointikieliet

Suurin osa korkean tason kielillä ohjelmoitavista tietokoneista perustuu von Neumannin arkkitehtuuriin² [Ril87]. Vaikka tietokoneen arkkitehtuuri asettaa tekniset puitteet ja rajoitukset ohjelmoinnille, korkean tason ohjelmointikielen suunnittelun ei tule kuitenkaan perustua samoihin tavoitteisiin kuin tietokoneen arkkitehtuurin suunnittelun [Dij65].

Von Neumannin arkkitehtuurin keskeisimmät piirteet ovat nykyaikaisen tietokoneen perusominaisuuksia [Ril87]. Arkkitehtuurissa käskyt ja data sijaitsevat samassa muistialueessa, jolloin käskyjä voidaan käsitellä kuten muutakin dataa [BGvN61]. Näin ollen voidaan käyttää hyödylliseksi havaittuja tekniikoita, kuten virtuaalimuistia, dynaamisesti ladattavia kirjastoja ja käynnissä olevia ohjelmia valvovia viruk-sensorijuntaohjelmia. Ennen von Neumannin hahmottelemaa arkkitehtuuria käskyt oli tallennettu erilliseen muistitilaan ja tietokoneet olivat huomattavasti joustamattomampia [Ril87].

Sen sijaan korkean tason kielessä muistin ja käskyjen yhtäläinen kohtelu johtaa ongelmiin. Mikäli ohjelma muokkaa itseään suorituksen aikana, se on huomattavasti alttiimpi virheille. Koska ohjelmakoodi voi muuttua, sen hahmottaminen on monimutkaisempaa ja virheiden riski kasvaa. Samoin ohjelmavirheiden selvittäminen on vaikeampaa, koska virhetilanteeseen ei vaikuta pelkästään ohjelman tila, vaan myös ohjelmakoodin tila. Esimerkiksi monissa nykyaikaisissa kielissä poikkeuksen mukana seuraa tieto siitä, mitä lähdekoodin riviä vastaavasta kohdasta se on heitetty. Mikäli ohjelma muokkaa itseään vapaasti, ei vastaavuuden selvittäminen ole mahdollista. Lisäksi monet hyödylliset korkean tason kieliin liittyvät rajoittavat ominaisuudet - kuten muuttujien näkyvyyden rajoittaminen, rinnakkaisten operaatioiden synkronoiminen ja taulukkojen tarkastettu indeksointi - eivät toimi, jos ohjelma voi itseään muokkaamalla kiertää rajoitukset. Toki ohjelmoija voi harjoittaa itsekuria

²John von Neumann julkaisi kollegoidensa kanssa 1940-luvulla ensimmäiset artikkelit, joissa kuvattiin yleiskäyttöisen tietokoneen arkkitehtuuri [Ril87].

ja käyttää itseään muokkaavaa koodia vain varoen ja harkitusti. Siitä huolimatta välinpitämättömyydestä tai huolimattomuudesta johtuvien virheiden riski kasvaa.

Toinen von Neumannin arkkitehtuurille ominainen piirre on tietokoneen toiminnan jakaminen neljään yksikköön³: kontrolli, aritmetiikka, muisti ja syöttö- ja tulostuslaitteisto [BGvN61]. Vastaava jako ei kuitenkaan ole toimiva korkean tason kielessä, koska monet hyvänä pidetyt ominaisuudet yhdistävät useisiin yksiköihin liittyviä käskyjä. Esimerkiksi normaalin indeksoidun taulukon iterointi, toteutettuna korkean tason rakenteella kuten iteraattorilla tai for-lauseella, yhdistää prosessorin kontrolliin, muistinkäsittelyyn ja aritmetiikkaan kuuluvia käskyjä. Tietokoneen arkkitehtuurin piilottaminen onkin usein hyödyllistä korkean tason kielessä.

Von Neumannin arkkitehtuurille ominaisia piirteitä tarkasteltaessa havaittiin, että ne eivät välttämättä ole toivottavia ohjelmointikielissä. Tietokoneen arkkitehtuurista ei voida suoraan johtaa hyvää ohjelmointikieltä [Dij65]. Von Neumannin arkkitehtuuri on hyödyllinen lähtökohta tietokoneen arkkitehtuurin suunnittelussa, mutta ohjelmointikielen suunnittelun tavoitteet tulee kuitenkin etsiä muualta.

³Von Neumann kutsui itse yksiköitä "elimiksi" - hänelle ihmisen anatomia oli innoittaja tietokoneen arkkitehtuurin suunnittelussa [BGvN61].

3 Yleisesti hyväksytyt suunnitteluperiaatteet

Mielestäni tärkeimmät klassisista lähteistä löytyvät hyvän ohjelmointikielen tunnusmerkit ovat:

- kieli on yksinkertainen [Hoa89]
- kieli ei ole liian geneerinen [Wir74]
- kielen ominaisuudet ovat samalla abstraktiotasolla keskenään [Wir74, Hoa89]
- ohjelman staattinen ja dynaaminen rakenne ovat mahdollisimman samankaltaisia [Dij68, WH89].

3.1 Yksinkertaisuuden ja geneerisyyden suhde

Ohjelmointikielen syntaksin yksinkertaisuutta voidaan pitää tärkeimpänä ohjelmointikielten suunnittelun tavoitteena [Hoa89]. Yksinkertainen syntaksi ei sisällä paljon rakenteita tai muistettavia sääntöjä, joten se on helppo oppia. Oppimisen helppous ei ole sivuseikka: monimutkaisen kielen opiskelu on hidasta ja voi johtaa siihen, että ohjelmoija ei hallitse kunnolla työkaluaan. Huonosti hallitun kielen käyttö taas johtaa helpommin virheisiin, mikä vähentää ohjelmoijan työtehoa.

Yksinkertaiselle kielelle on helppo toteuttaa kääntäjä, joka puolestaan yksinkertaisena ohjelmanä ei ole altis virheille ja toimii tehokkaasti. Samoin muiden kieltä analysoivien työkalujen tekeminen on helppoa. Helposti käännettävän kielen syntaksi on myös vaivattomasti ihmisen ymmärrettävissä. Vaikka monimutkaiselle syntaksille olisikin mahdollista toteuttaa toimiva ja tehokas kääntäjä, olisi monimutkaisen lähdekoodin ymmärtäminen ihmiselle silti vaikeaa [Hoa89]. Ohjelmointikieli ei ole pelkästään työväline tietokoneen “käskyttämiseen”, vaan myös tapa kommunikoida ja dokumentoida ajatuksia.

Geneerisyys on tärkeä käsite ohjelmointikielten suunnittelussa. Geneerisyydellä tarkoitetaan tässä yhteydessä sitä, että hyvin pientä operaatioiden joukkoa voidaan

soveltaa monenlaisiin tehtäviin [Wir74]. Hyvin geneerinen ohjelmointikieli on esimerkiksi konekieli, jossa yleensä toimitaan kohtuullisen pienellä käskyjen joukolla. Käykyjä voi soveltaa lähes kaikkeen dataan: yhteen voi laskea lukujen lisäksi esim. kirjaimia tai jopa toisia käskyjä. Merkittävä osa kaikkien mahdollisten operaatioiden joukosta on sallittu, vaikka vain murto-osa niistä tuottaa ohjelmointitehtävän kannalta järkeviä tuloksia. Toinen esimerkki liiallisen geneerisyyden haitallisuudesta on Wirthin kehittämä Euler-kieli. Se on hyvin yksinkertainen, geneerinen ja erittäin joustava, mutta sillä kirjoitettu lähdekoodi on jopa kehittäjänsä mielestä lähes käsittämätöntä [Wir74].

Konekielen ilmaisun vapaus on haitallista ohjelmoijalle, joka ratkoo korkean tason ohjelmointiongelmia. Yksinkertaisuutta ei tulisi saavuttaa liiallisella geneerisyydellä, vaikka geneerinen ratkaisu voikin olla joustava [Wir74].

Yleensä ohjelmointikielen geneerisyyttä pidetään hyvänä ominaisuutena [Mey86]. Geneerisyys ei ole kuitenkaan itseisarvo: jotta ohjelmointikielellä voidaan toteuttaa selkeitä ohjelmia, tulee sen asettaa järkeviä rajoitteita sallituille operaatioille [Wir74]. Ohjelmoijan tekemät ajatusvirheet käyvät helpommin ilmi, kun lähdekoodin tulee noudattaa annettua rajoitettua esitysmuotoa.

3.2 Abstraktioiden merkitys ohjelmoijalle

Ohjelmointikieli asettaa rajoitteita ohjelmien toteutukselle. Rajoitteiden tulee estää ohjelmoijaa tekemästä virheitä, mutta ei hankaloittaa ohjelmointityötä. Toisin sanoen niiden tulee tukea ohjelmoijan virheetöntä ajattelua - antaa puitteet ongelmanratkaisulle.

Millaiset rajoitukset tukevat ohjelmoijan ajattelua? Tarkastellaan kahta ominaisuutta, joita pidetään haitallisina korkean tason ohjelmointikielessä: hyppykäsky [Dij68] ja muistin vapaa käsittely, jolla tarkoitetaan osoitinaritmetiikkaa ilman tyyppitarkistuksia [Wir74]. Molemmat ovat tyyppillisiä konekieliohjelmoinnille.

Proseduraalisessa kielessä ohjelman suoritus voidaan ajatella proseduurien kutsupinona, jossa suoritettava proseduuuri on päällimmäisenä. Mahdollisuus ajatella ohjelma proseduurien rakennelmana on ohjelmoijalle arvokas abstraktio, joka helpottaa ohjelman hahmottamista. Korkean tason rakenteet voidaan suunnitella niin, että ne tukevat tätä hahmottamisen mallia. Alemman abstraktiotason rajoittamaton hyppykäsky ei kuitenkaan tue piirrettä, vaan mahdollistaa hypyn kesken proseduurin johonkin sattumanvaraiseen ohjelman kohtaan. Vaikka rajoittamattoman hyppykäskyn käyttö onkin mahdollista von Neumannin arkkitehtuurin mukaisissa tietokoneissa, on se lähes poikkeuksetta haitallista ohjelman rakenteen selkeydelle [Dij68]. Kun kutsupino ohitetaan hyppykäskyn avulla, tulee ohjelman toiminnan ymmärtäminen paljon vaikeammaksi ja ohjelmointivirheiden riski kasvaa.

Myös osoitinaritmetiikalla voidaan tuhota korkean tason ohjelmointikielen hyödyllisiä abstraktioita. Tyypittämättömän osoitinaritmetiikan avulla voidaan kaikkea muistin sisältöä käsitellä vapaasti, myös käskykoodeja. Mikäli ohjelma muuttaa ohjelmointivirheen vuoksi omia käskykoodejaan, aiheuttaa muuttunut käskykoodi toimintavirheen, joka tulee ilmi ehkä jossain toisessa ohjelman suorituksen kohdassa ja on näin ollen erittäin hankala korjata. Jo pelkkää osoitinaritmetiikkaa - myös tyypitettyä - voidaan pitää liian matalan tason ominaisuutena nykyisiin ohjelmointikieliin, joissa muisti vapautetaan roskienkeruun avulla - kuten Javassa tai C#:ssa. Koska osoittimien arvoja voidaan käsitellä aritmeettisesti, voidaan aina tuottaa uusi osoitin tietoalkioon, johon ei muuten enää olisi osoittimia. Näin ollen roskienkerääjä ei koskaan voi olla varma, että voidaanko tietoalkio vapauttaa. Osoitinaritmetiikka tekee luotettavasta ja ohjelmoijalle helppokäyttöisestä roskienkeruusta hyvin vaikean toteuttaa.

Tarkasteltaessa kahta konekieliohjelmoinnille keskeistä ominaisuutta korkean tason kielten yhteydessä havaittiin, että matalan abstraktiotason ominaisuudet eivät sovellu korkean tason kieleen. Ohjelmointikielen rakenteiden tulisi siis olla samalla abstraktiotasolla keskenään [Wir74, Hoa89]. Matalan tason ominaisuuksilla tehdyt ratkaisut voivat olla yksinkertaisia ja elegantteja. Yksittäisen ohjelmointiratkaisun liiallinen yksinkertaistaminen voi kuitenkin johtaa ohjelman kokonaisrakenteen mo-

nimutkaistumiseen. Kuten Einstein on sanonut: “Tee kaikesta mahdollisimman yksinkertaista, mutta ei yhtään sen yksinkertaisempaa”⁴. Vakavasti otettava ohjelmistosuunnittelu sisältää pääasiassa monimutkaisia ohjelmointiongelmia [FPB87], joten yksinkertaiset ja elegantit ratkaisuvat eivät riitä.

3.3 Ohjelman dynaaminen ja staattinen muoto

Hyppykäskyjen avulla tuotettua koodia kutsutaan “spagettikoodiksi”. Sillä tarkoitetaan ohjelmakoodia, jossa suorituskohdan siirtymät ovat niin monimutkaisia, että ohjelmoijan on erittäin vaikea ymmärtää ohjelman ajonaikaista toimintaa lukemalla sen lähdekoodia [Dij68]. Siis toisin sanoen ohjelman dynaaminen (ajonaikainen) ja staattinen (lähdekoodista luettavissa oleva) muoto ovat hyvin erilaiset. Korkean tason kielet voidaan suunnitella niin, että nämä kaksi muotoa ovat lähempänä toisiaan ja ohjelman dynaaminen toiminta voidaan hahmottaa huomattavasti helpommin lähdekoodia lukemalla.

Jotta lähdekoodi kuvaisi hyvin ohjelman ajonaikaista toimintaa, tulee sen kirjoitusasu sisältää konekoodia enemmän informaatiota. Esimerkiksi iteraatio toteutetaan useimmissa konekielissä ehdollisella hyppykäskyllä, joka sisältää minimaalisen määrän informaatiota: hyppyehdon ja hyppyosoitteen. Korkean tason kielet sisältävät useita erilaisia rakenteita iteraation toteuttamiseksi, kuten erityyppiset silmukat ja iteraattorit. Näin ollen korkean tason kielellä voidaan kuvata tarkemmin iteraation tarkoitusta ja sen ajonaikaista käyttäytymistä. Vaikka lisäinformaatio kasvattaa ohjelman lähdekoodin kokoa ja hidastaa sen kirjoittamista, kirjoittamiseen liittyvä vaiva kompensoituu kuitenkin ohjelman hyvällä luettavuudella. Lähdekoodin luettavuus on tärkeämpää kuin kirjoittamisen helppous [Hoa89].

Tässä kappaleessa klassisia lähteitä tarkasteltaessa havaittiin, kuinka tärkeässä roolissa ohjelmoija ja hänen ajatusmaailmansa on. Ohjelmointikielen tulee asettaa sellaisia rajoitteita, jotka tukevat ohjelmoijan ajattelua ja estävät ajatusvirheitä. Ohjelmoijan tulee pystyä hahmottamaan ohjelman ajonaikainen toiminta mahdollisim-

⁴Ks. <http://www.einsteinalive.com> (tarkistettu 24.2.2003).

man hyvin lähdekoodista - hänen siis tulee lähdekoodin perusteella rakentaa mielessään malli, joka kuvaa ohjelman toimintaa. Mallin rakentaminen ei onnistu, mikäli kieli sisältää matalan tason rakenteita, jotka rikkovat hahmottamiselle hyödyllisiä korkean tason abstraktioita. Samoin mikäli kieli on liian monimutkainen, ohjelmoija joutuu käyttämään henkiset voimavaransa kielen hahmottamiseen, ja itse ohjelmointiongelman hahmottamiseen ja hyvän ratkaisumallin luomiseen ei ole riittäviä mahdollisuuksia.

4 Olio-ohjelmointi

Olio-ohjelmointi on paradigma, jonka tarkoitus on parantaa lähdekoodin modulaarisuutta ja siten hallittavuutta, joustavuutta ja ymmärrettävyyttä, sekä edistää ohjelmakoodin uudelleenkäyttöä. Lisäksi oliomalli vastaa perinteistä proseduraalista paradigmaa paremmin ohjelmointitehtävän määrittelyssä käytettäviä käsitteitä [Ber93, Par72].

4.1 Olioparadigma hahmottamisen välineenä

Oliot ovat kokonaisuuksia, jotka sisältävät tilan (dataa) ja toimintoja (ohjelmakoodia). Olio kapseloi omat toteutusyksityiskohtansa ja esittää muille rajatun liittymän, jonka kautta oliota voidaan käyttää.

Oliomalli tukee ohjelmoijan ajattelua tekemällä koodin toteutusyksityiskohdisti paikallisempia, jolloin kerralla hahmotettavan tiedon määrä on pienempi [LHR88]. Paikallisuus on tärkeä käsite kaikissa ohjelmointiparadigmoissa. Paradigmojen erot tulevat sen sijaan esiin siinä, mistä näkökulmasta ohjelma jäsennetään, eli millä tavalla asiat ryhmitellään keskenään. Esimerkiksi funktionaalisessa ohjelmoinnissa ohjelma rakennetaan funktioiden avulla. Niinpä ohjelman informaatio on jäsentynyt funktioiksi. Olio-ohjelmoinnissa sen sijaan ohjelma rakennetaan olioiden avulla, jolloin informaatio on jäsentynyt olioksi [Ber93, s. 10]. Molemmat paradigmat mahdollistavat ohjelman modularisoinnin, mutta eri näkökulmasta.

Ei ole olemassa yleispätevää mittaria, jolla voitaisiin arvioida mikä paradigma on paras ohjelman jäsentämiseen. On kuitenkin olemassa näyttöä, että ihmisen on helpompi hahmottaa todellisen maailman käsitteitä olioina kuin funktioina [Ber93, s. 21-22].

Olio-ohjelmointi tukee koodin uudelleenkäyttöä, joka varsinkin konkreettisesti tehostaa ohjelmointityötä: jos kerran ohjelmoituja ratkaisuja ei tarvitse toteuttaa uudelleen. Uudelleenkäyttö ei koske pelkästään konkreettista ohjelmakoodia, vaan myös hyviä ajatuksia ja ratkaisumalleja. Ratkaisujen uudelleenkäytöstä hyvä esimerkki ovat

suunnittelumallit (engl. design patterns), jotka ovat nousseet 90-luvun loppupuolella hyvin suosituiksi olio-ohjelmoijien keskuudessa [GHJV95].

4.2 Modularisointi vaatii ilmaisuvoimaa

Hyvässä oliorakenteessa oliot keskustelevat vain “naapureidensa” kanssa. Tämä ajatus on pohjana nk. Demeterin laissa: metodi saa käyttää toista oliota vain, jos olio kuuluu metodin argumenttiluokkiin tai metodin sisältävän olion jäsenmuuttujien luokkiin⁵ [LHR88]. Demeterin laki korostaa ohjelman paikallisuutta. Paikallisuudella on monia suotuisia vaikutuksia: lähdekoodi on helpommin hallittavissa, virheet helpommin paikallistettavissa ja koodi on paremmin uudelleenkäytettävissä [LHR88]. Paikallisuus auttaa merkittävästi ohjelman monimutkaisuuden hallinnassa [LH02, Str94].

Osa ohjelman toiminnalle asetetuista vaatimuksista on nk. risteäviä vaatimuksia⁶ (engl. crosscutting concern), eli ohjelmalle asetettuja vaatimuksia, jotka eivät noudata muun ohjelman rakennetta, vaan esim. koskettavat useita toisiinsa liittymättömiä luokkia. Risteävien vaatimusten vuoksi monissa tilanteissa oliomalli ei voi vastata täysin mallinnettavan ongelmakentän käsitteitä [KHH⁺01].

Olio-ohjelma voidaan muuntaa aina Demeterin lakia noudattavaksi [LHR88]. Risteävien vaatimusten vuoksi Demeterin lain noudattaminen voi johtaa siihen, että ohjelman rakenne ei enää vastaa todellisen maailman mallia [KHH⁺01]. Esimerkiksi jos vaaditaan, että sovelluksen kaikki metodit kirjoittavat merkinnän lokitiedostoon, vaikuttaa se myös sovelluksen luokkiin, jotka eivät liity käsitteellisesti lokitoiminnallisuuteen. Jos useita toisiinsa liittymättömiä olioita koskeva ominaisuus toteutetaan Demeterin lakia noudattaen, eli paikallisuus säilyttäen, vaatii se väliolioiden lisäämistä. Välioliot ovat täysin keinotekoisia, joten niillä ei ole vastaavuutta ohjelmointitehtävän määrittelyssä annettuihin käsitteisiin. Mikäli olio-ohjelmointikieli tukee

⁵Katso tarkempi formalisointi: [LHR88].

⁶Olen nähnyt myös käytettävän suomennosta “yhteiset aiheet”. Pidän omaa suomennostani “risteävät vaatimukset” kuitenkin tähän asiayhteyteen soveltuvampana.

aspekteja eli modularisoituja risteäviä vaatimuksia - kuten Java AspectJ:n avulla [KHH⁺01] - voidaan modularisointi tehdä rikkomatta vastaavuutta todellisen maailman kanssa ja silti Demeterin lakia noudattaen.

Demeterin lain noudattaminen voi vähentää ohjelman rakenteen vastaavuutta todellisen maailman kanssa, mikä vaikeuttaa ohjelman hahmottamista. Lisäksi lain vaatima muutos voi johtaa useampiin metodeihin ja metodien argumentteihin, siis monimutkaistaa ohjelman lähdekoodia [LHR88], mikä on ristiriidassa yksinkertaisuuden periaatteen kanssa. Lisäämällä kielen ilmaisuvoimaa - kuten lisäämällä aspektit kielen rakenteiksi - voidaan säilyttää modulaarisuus, paikallisuus ja vastaavuus mallinnettavan maailman kanssa. Kuitenkin mikäli ohjelmointikielen ilmaisuvoimaa lisätään, lisää se myös kielen monimutkaisuutta, koska muistettavien ja hahmotettavien kielen yksityiskohtien määrä lisääntyy.

Yksi merkittävä piirre ohjelmointikielten kehityksessä on ollut modulaarisuuden lisääntyminen. Modulaarisuus ei ole kuitenkaan itseisarvo, vaan sillä havaittiin tässä tarkastelussa myös haitallisia piirteitä. Suoraviivainen modulaarisuuden parantaminen voi vaikeuttaa ohjelman hahmottamista. Onkin tärkeää, että ohjelmointikielen tarjoamat ominaisuudet modularisointiin mahdollistavat ohjelman jakamisen samalla tavalla kuin ohjelmoija jakaa ratkaistavan ongelman omassa ajatusmaailmassaan [Wir74]. Tässä yhteydessä havaitaan sama ilmiö kuin ohjelmointikielen rajoitteita tarkasteltaessa: kaikista mahdollisista rakenteista tulee korkean tason kieleen valita ne, jotka yhdessä tukevat parhaiten ohjelmoijan ajattelua.

5 Ohjelmointi ja ihmisen ajattelu

Kognitiopsykologi Neisserin mukaan ihmisen ajattelussa on kolme merkittävää eroa tietokoneen toimintaan [Nei63]:

- ihmisen ajattelumaailma on jatkuvasti rakentuva
- ajatteluun liittyvät aina myös tunteet
- ajattelulla on useita päämääriä.

Neisserin havaitsemien erojen pohjalta ohjelmoinnin ei voida ajatella olevan suoraviivaista ongelmanratkaisua. Monissa vanhemmissa ohjelmistotuotannon malleissa, kuten vesiputousmallissa [Pre00, s. 26-29], ohjelmistotuotantoprosessi on lineaarinen. Lineaarinen malli ei huomioi sitä, että ohjelmoijan ajattelu on jatkuvasti kehittyvää. Kun prosessi on suunnitteluvaiheessa, ohjelmoija ei voi tehdä hyvää ja kattavaa suunnitelmaa valmiista ohjelmasta, koska hänen käsityksenä itse ratkaistavasta ongelmasta kehittyy jatkuvasti. Uudemmat kevyet ohjelmistotuotannon mallit, kuten Extreme Programming [Bec00], huomioivatkin paremmin ihmisen kehittyvän ajattelumaailman.

Ongelmanratkaisulla on yksi ainoa päämäärä: ratkaista annettu ongelma. Sen sijaan ongelmaa ratkaisevalla ohjelmoijalla - inhimillisenä tekijänä - on useita päämääriä [Nei63]. Sellaisia voivat olla esimerkiksi: kirjoittaa koodia, joka on selkeää, helposti ymmärrettävää, jopa esteettistä; käyttää yleisesti hyväksytyjä ratkaisumenetelmiä, käyttää tekniikoita joita ohjelmoija haluaisi osata paremmin; kirjoittaa koodia joka vaikuttaa ammattimaiselta ja tehdä nokkelia ratkaisuja, jotka lisäävät arvostusta kollegoiden piirissä; ja tietenkin myös ratkaista annettu ongelma. Mekaaninen ongelmanratkaisu ja luova toiminta eroavat suuresti päämääriltään [Sim79b]. Jotta ohjelmointikieli olisi hyvä työkalu, sen tulisi huomioida mahdollisimman laajasti ohjelmoijan ajattelun monenlaiset päämäärät.

Tunteiden ja ohjelmointikielten suunnittelun yhdistäminen vaikuttaa Neisserin kolmesta kohdasta haastavimmalta. Inhimillisten tunteiden toimintaa ei tunneta kuin

suppeasti. Tunteiden ja muun ajattelun suhteen hahmottamista helpottaa kuitenkin kognitiotieteilijä Herbert Simonin ehdottama tapa ajatella tunteita ajattelun toimintona, joiden avulla ihminen vastaa yllättäviin tilanteisiin [Sim79b]. Esimerkiksi ohjelmassa olevaan virheeseen ohjelmoija voi reagoida hyvinkin tunnepohjaisesti. Toisaalta Simon toteaa, että mitä paremmin henkilö hallitsee tehtävänsä, sitä vähemmän tunteita sen tekemiseen liittyy. Näin ollen ammattiohjelmoijalle ohjelmointi ei ole erityisen tunnepitoista työtä, mikä vastaakin hyvin intuitiotamme - esimerkiksi virheiden etsimisessä systemaattisuus kasvaa kokemuksen myötä.

Ammattimainenaan ohjelmointi ei ole kuitenkaan vapaa tunteista. Suurin osa ohjelmistoista liittyy johonkin tiettyyn sovellusalueeseen, jota ohjelmoijat eivät ennalta käsin tunne. Niinpä heillä ei voi olla muodostunut ammattimaista suhtautumistapaa sovellusalueeseen. Ohjelmoijat voivat olla esimerkiksi epäluuloisia, jopa vihamielisiä, ohjelmointitehtävää kohtaan. Ohjelmointityö tapahtuu myös aina jossain ympäristössä, kuten yrityksessä tai virastossa, jossa vallitsevat tietyt säännöt ja lait. Ohjelmistoja tuottavat organisaatiot, jotka koostuvat erilaisista ihmisistä. Organisaatioilla on historiallinen tausta, arvonsa ja normista [Hof97]. Ohjelmointityöhön vaikuttavia аспекteja on siis hyvin paljon, joten ammattimainen rutinoituminen ei voi sammuttaa tunteita kaikilta työnteon osa-alueilta. Ja voi myös kysyä, johtaako rationaalinen ja tunteet tukahduttava ohjelmointityö tunnekouhuihin muussa työelämässä.

Ehkä merkittävin ohjelmointiin liittyvä tunteiden osa-alue on estetiikka⁷, tunnepohjainen kokemus siitä mitä henkilö pitää kauniina ja hyvänä [Boo91, s. 19-20]. Alunperin teknologia ymmärrettiin yhtä hyvin taiteena kuin tieteenä [Air03, s. 11-19]. Simonin mukaan tekniikka, kuten ohjelmointitekniikka, on kuitenkin nykyaikana muuttunut liian tieteelliseksi. Perinteiset insinööritaidon hyveet, kuten estetiikka, ovat jääneet liian vähälle huomiolle [Sim82, s. 129-132]. Luonnontieteellisen taustansa vuoksi tietojenkäsittelytieteessä on totuttu tutkimaan miten asiat ovat. Oh-

⁷Vaikka tässä yhteydessä keskitytään ohjelman rakenteen estetiikkaan, myös ohjelman syntaksin estetiikalla on merkitystä. Esim. C++:ssa rumilla syntaktisilla rakenteilla on pyritty viestimään siitä, että rakenteen käyttö ei yleensä ole suositeltavaa [Str94, s. 119].

jelmistosuunnittelun kannalta tärkeämpää olisi tutkia sitä miten asioiden tulisi olla. Silloin kuitenkin törmätään voimakkaan subjektiivisiin asioihin, kuten estetiikkaan, joiden tutkiminen perinteisin luonnontieteellisin menetelmin on hyvin hankalaa.

Vaikka rutinoituminen vähentääkin tunteiden merkitystä työnteossa, tunne-elämä on silti merkittävä tekijä myös ammattimaisessa ohjelmointityössä. Ohjelmoijan työelämä on inhimillistä elämää kaikkine ulottuvuuksineen, jotka välttämättä vaikuttavat myös itse työntekoon. Erityisesti esteettiset tunnetilat tulee ottaa huomioon poh-tiessa ihmisen ajattelun ja ohjelmoinnin suhdetta.

6 Monimutkaisuuden hallinta

Monimutkaisuuden hallinta on merkittävä haaste ohjelmistoprojekteissa [Ber93, s. 164-165]. Myös ohjelmoijan ajattelun sujuvuudelle monimutkainen ohjelmointitehtävä asettaa suuria haasteita. Luovuuspsykologiassa sujuvaa ajattelutyötä kuvataan nk. virtaustilaksi (eng. flow). Luovuustutkija Csikszentmihalyi on laajan haastattelututkimuksen perusteella päättänyt liittämään virtaustilaan yhdeksän määrettä: tavoite on selkeä koko ajan, omista toimistaan saa jatkuvaa palautetta, tehtävä on sopivan haasteellinen, työ on täysin keskittynyttä, häiritseviä tekijöitä ei ole, epäonnistumisen pelkoa ei ole, itsetietoisuus katoaa, ajankäsitys hämärtyy ja työstä tulee autotelista, eli työn tekeminen on päämäärä itsessään. Virtaustila on universaali tunne: kulttuuri-, etnisestä- tai koulutustaustasta huolimatta ihmiset kuvailevat virtaustilaa hyvin samanlaisin määrein [Csi97].

Ohjelmointitehtävään liittyvä monimutkaisuus voi tuhota virtaustilan. Monimutkainen ja jäsentymätön tehtävä sisältää paljon erillisiä tavoitteita, joiden keskinäistä suhdetta on aluksi vaikea nähdä. Jotta työ olisi sujuvaa, tulisi tavoitteiden olla selkeitä koko ajan: monimutkaisuuden hallitseminen vaatii siis paljon ihmisen tiedonkäsittelyltä.

6.1 Monimutkaisuus ja ihmisen muisti

Ihmisen ajattelussa muisti on korkeamman tiedonkäsittelyn perusta [Saa90, s. 19-21]. Muisti voidaan jakaa karkeasti säilömuistiin ja työmuistiin [Roe80]. Säilömuistin kapasiteetti on hyvin suuri, joten lukuisienkaan yksityiskohtien muistaminen ei ole mahdotonta. Säilömuistin hyvä toiminta edellyttää kuitenkin syvää käsittelyä ja oikeanlaisia palautusvihjeitä.

Termi “syvä käsittely” viittaa siihen, että ihminen muistaa paljon paremmin asioita, joita on prosessoanut mielessään perinpohjaisesti [Cra72]. Toisto ei vielä yksinään auta muistamaan. Esimerkiksi harva muistaa miltä hänen ulko-ovensä näyttää, vaikka onkin nähnyt sen ehkä tuhansia kertoja. Sen sijaan yksi elämyksellinen

tapahtuma voidaan muistaa hyvinkin tarkkaan, vaikka se on ollut ainutkertainen. Itse ohjelmointitehtävään keskittyminen on siis säilömuistin kannalta erittäin tärkeää - esimerkiksi häiritsevät tekniset yksityiskohdat heikentävät muistin toimintaa, vaikeuttaen näin myös kokonaisuuden hahmottamista.

Palautusvihjeet ovat eräänlaisia muistin "hakupolkuja". Monet jo unohdetuiksi luullut asiat palautuvat mieleen tutun tilanteen tai esim. aistielämyksen, kuten hajun, kautta. Samoin laajan ohjelmointitehtävän muistaminen vaatii, että sen hetkisestä henkisestä tilasta on löydettävissä hakupolku muistettavaan asiaan [Hin90]. Asioiden mieleenpalauttaminen on hankalampaa, jos alkuperäisessä tilanteessa oli hyvin erilaisessa mielentilassa kuin muistamishetkellä. Esimerkiksi lapsuuden tapahtumien unohtumista on selitetty siten, että aikuisen erilainen maailmankatsomus ei tarjoa sopivia hakupolkuja lapsuuden muistoihin. Ohjelmointityössä taas muun muassa liika teknisyys voi haitata sovellusalueen asioiden muistamista: jos joutuu jatkuvasti ajattelemaan teknisesti, ei sopivia hakupolkuja ohjelman määrittelyyn ja esimerkiksi asiakkaan kanssa käytyihin keskusteluihin välttämättä löydy. Tällöin määrittely jää helposti huomioimatta, mikä johtaa siihen että syntynyt ohjelma ei ole määrittelyn mukainen - mikä onkin yleinen ohjelmistoprojektien ongelma.

Ihmisen työmuisti on hyvin pieni. Siihen mahtuu noin 4 - 7 mieltämysyksikköä (engl. chunk) [Mil56], jotka voivat kuitenkin olla hyvinkin suuria. Mieltämysyksikkö voi olla yksi numero, tai toisaalta yksi puhelinnumero, joka koostuu useista numeroista [Sim79a]. Monimutkaisen ohjelman kohdan hahmottamiseen seitsemän mieltämysyksikköä on vähän: harvan algoritmin voi toteuttaa seitsemällä käskyllä. Ohjelmoijat kuitenkin toteuttavat sujuvasti vaativiakin algoritmeja. Mieltämysyksikköjen täytyykin olla silloin laajempia. Esimerkiksi taulukon iterointi voi olla yksi mieltämysyksikkö, vaikka se koostuukin lähdekoodissa useista käskyistä ja vielä useammista kirjaimista.

Avain työmuistin tehokkaaseen käyttöön on siis suurempien mieltämysyksiköiden rakentaminen. Näin ollen kapselointi on myös ihmisen tiedonkäsittelyn näkökulmasta hyvin merkittävä monimutkaisuuden hallinnan työkalu. Jotta säilömuisti toimisi hy-

vin, tulisi ohjelmoijan huomion liikkua mahdollisimman paljon itse ohjelmointiongelmassa ja asiakkaan asettamissa vaatimuksissa.

Hyvä muistin toiminta on perusta hyvälle tiedonkäsittelylle. Ongelmat muistamisessa häiritsevät työtä ja tuhoavat virtaustilan. Tietotyöläiset työskentelevät “informaatioähkyn” ytimessä [Kos98], joten hyvä henkilökohtaisen muistin käyttö on keskeinen taito ohjelmoijan työssä.

6.2 Ohjelma on ohjelmointitehtävän yleistys

Ohjelman sisältämän kapseloidun abstraktion, kuten oliomallin, voidaan ajatella olevan yleistys todellisesta maailmasta, siis eräänlainen stereotypia. Stereotypiat ovat paljon tutkittu ilmiö sosiaalipsykologiassa [Lie98]. Eri tutkijoiden käsitykset aiheesta ovat ristiriitaisia, mutta kohtuullisen yleisesti hyväksytään, että stereotypiat ovat tärkeä tapa hallita suuria havaintojen määriä. Samalla tavoin abstraktiot ovat erittäin merkittävä monimutkaisuuden hallinnan väline ohjelmistosuunnittelussa [Boo91, s. 39-45]. Todellisuuden yleistäminen ja käsiteltävän tiedon tehokas valikoiminen ovat ihmisajattelulle keskeisiä ominaisuuksia [Saa90, s. 15-19].

Olio-ohjelmoija hahmottaa ratkaistavaa ongelmaa yleistämällä sitä luokiksi ja olioiksi, samaan tapaan kuin ihminen hahmottaa muita ihmisiä yleistämällä heitä stereotypioiksi. Stereotypiat eivät ole pysyviä, vaan ne tarkentuvat ja muuttuvat jatkuvasti. Stereotypiat voivat kuitenkin ohjata havaintoja: ihmiset näkevät toisissaan sellaisia ominaisuuksia, kuin olettavat etukäteen heissä olevan. Opitut skeemat ohjaavat voimakkaasti ajatteluprosessia [Saa90, s. 32-35]. Sama vaara piilee myös ohjelmointikielten avulla rakennettavissa yleistyksissä - ja yleensä kaikessa luovassa suunnittelutyössä [Hak02, sivu 210]. Esimerkiksi kun monimutkaisen ohjelmointitehtävän hahmottaminen on aloitettu rakentamalla alustava oliomalli⁸, voidaan kaikki tehtävään liittyvät vaatimukset alkaa nähdä sen kautta, mikä vääristää ohjelman jatkokehitystä.

⁸Kuten suositussa Unified Process -menetelmässä [Lar01, s. 127-153].

Kuten Neisser on todennut [Nei63], ihmisen ajattelu on jatkuvasti rakentuvaa. Monimutkaisuuden hallinnassa abstraktiot ovat tärkeitä, myös sen vuoksi, että ne mahdollistavat tiettyjen aspektien, kuten teknisen toteutuksen, piilottamisen ja sitä kautta vähentävät ohjelman hahmotettavia ulottuvuuksia. Yleistämällä saadaan ote ohjelmointitehtävästä, mutta aluksi rakennettu hahmottamismalli on yleensä riittämätön, joten sen ei saa antaa ohjata ohjelman kehitystä. Ohjelman rakenteen tulee kehittyä jatkuvasti, jotta se olisi mahdollisimman hyvä yleistys ratkaistavasta ongelmasta. Näin ollen jatkuva refaktorointi on tärkeää laajempien ohjelmistojen kehityksessä [Fow99].

Tulee myös huomata, että keskeneräinen ohjelma voi ohjata jatkokehitystä myös positiivisessa mielessä. Monimutkainen biologinen järjestelmä kehittyy nopeammin, jos sillä on vakaita välimuotoja [Sim82, s. 209]. Samaa evolutiivista ajatusta voidaan soveltaa myös suuriin tietojärjestelmiin: uutta järjestelmää pyritään rakentamaan "vakaiiksi" havaittujen abstraktioiden päälle [Boo91, s. 19-20]. Ohjelmoinnin yhteydessä vakaita välimuotoja vastaavat prototyypit, joilla tehdyt kokeilut antavat realistisemmän kuvan ratkaistavasta ongelmasta.

Ohjelman rakenteessa siis heijastuu ihmisen käsitys ratkaistavasta ongelmasta, mutta toisaalta ohjelman rakenne omalta osaltaan ohjaa ihmisen ajattelua - sekä negatiivisessa että positiivisessa mielessä. Ohjelman ja ohjelmoijan välisen vuorovaikutuksen voidaankin ajatella olevan eräänlaista dialogia.

6.3 Ohjelmointityö dialogina

Sujuvalle ajattelutyölle ominainen virtaustila on mahdollinen vain kun liialliset yksityiskohdat eivät häiritse ajatustyötä [Csi97]. Tämän vuoksi ajattelu tapahtuu sujuvimmin tietyn näkökulman kautta, koska monien tarkastelukulmien samanaikainen huomioonotto lisää yksityiskohtien määrää valtavasti. Mutta toisaalta todellisten ongelmien ratkaiseminen vaatii usein monia tarkastelukulmia. Korkeammille henkisille toiminnoille onkin keskeistä kyky havaita ja sovittaa yhteen erilaisia todellisuutta koskevia näkökulmia [KJH98].

Ihmisen tulee ottaa huomioon useita näkökulmia - ja toisaalta pyrkiä minimoimaan niiden tuomaa monimutkaisuutta. Tämä taito on mielen toiminnalle keskeinen, ja se myös erottaa kokeneemmat aloittelijoista [STSL97]. Sosiaalipsykologi George Mead näkee ihmisen ajattelun dialogina, mielen sisäisenä vuoropuheluna, useamman eri näkökulmasta asiaa tarkastelevan persoonan kesken [Mea34]. Dialogimaisuus onkin tyypillistä ihmisajattelulle [KJH98].

Dialogin käyminen vaatii näkökulman vaihdoksia. Yksiparadigmainen ohjelmointikieli ei tarjoa mitään tukea tarkastelukulman vaihdokselle. Niinpä törmätessään ongelmaan, jota ohjelmoija ei kykene ratkaisemaan kielen paradigman avulla, hänen tulee tukeutua pelkästään omaan järkeensä tai ulkopuoliseen apuvälineeseen, kuten muistilehtiöön tai CASE-ohjelmistoon, voidakseen rakentaa erilaisen näkökulman ongelmaan. Usein monimutkainen ongelma ratkeaa, kun löytää uuden näkökulman, josta tarkastellen ongelma onkin hyvin yksinkertainen. Oivallukseen liittyy voimakas tunne-elämys [Csi97]. Hyvä esimerkki tästä tunteesta on Arkhimedeksen kuuluisa huudahdus: "Heureka!" jota hän toisteli keksittyään tavan tilavuuden määrittämiseen. Oivallusta voidaankin pitää yhtenä keskeisimmistä elementeistä luovassa prosessissa [Hak02, sivu 218].

7 Visio hyvästä ajattelun apuvälineestä

Esitän visioni siitä millainen ohjelmointikielen tulisi olla, jotta se olisi hyvä ajattelun apuväline. Ajatukseni perustuvat tutkielmassa tekemiini havaintoihin ja toimivat yhteenvetona mielestäni tärkeimmistä tutkielmassa esille tulleista ohjelmointikielten suunnitteluperiaatteista.

Virtaustilan yhdeksästä määreestä mielestäni ohjelmointityön kannalta tärkeimpiä ovat tavoitteiden selkeys, palaute ja häiriöttömyys. Tavoitteiden selkeys vaatii sekä hyviä työkaluja ohjelmointitehtävän määrittelyyn, eli kaikkien vaatimusten selvittämiseen, että löydettyjen vaatimusten hallintaan.

Oliot ovat ihmiselle luonnollinen tapa hahmottaa todellista maailmaa, joten oliopohjaisuus on hyvä lähtökohta. Erityisesti olioiden tarjoama kapselointi auttaa monimutkaisuuden hallinnassa merkittävästi. Näin ollen voisi olla perusteltua tehostaa kapselointia yleisiin oliokieliin verrattuna. Oliot voisivat esimerkiksi kapseloida oman suoritussäikeensä, siis olla nk. aktiivisia olioita. Tiukempi kapselointi tehostaisi abstrahointia, joka on oleellista monimutkaisuuden hallitsemisessa. Jotta kieli tukisi ohjelmoijan kehittyvää ajattelua, sen ei tulisi lukita häntä sen hetkisiin abstraktioihin. Kielen tulisi siis tukea myös refaktorointia mahdollisimman hyvin.

Sujuva ajattelutyö vaatii jatkuvaa palautetta. Normaalin ohjelmointityön käännyssykli voi olla ongelmallinen. Ohjelmaan tulee tehdä kerralla kohtuullisen suuria muutoksia, että sen voi kääntää uudelleen, joten ohjelmoija ei voi aina varmistaa tekemiään pieniä muutoksia. Nykyaikaiset kääntäjät ovat hyvin monimutkaisia, mikä kertoo hyvin ohjelmoijan ja koneen välisen dialogin löyhyydestä: kone joutuu tekemään hyvin paljon työtä ymmärtääkseen ohjelmoijan viestin, lähdekoodimuodossa olevan ohjelman; ja toisaalta ohjelmoija voi joutua testaamaan ja profiloimaan kääntäjän tuottamaa ohjelmaa, jotta hän ymmärtää sen riittävän hyvin.

Tulkattavissa kielissä syntaksin oikeellisuuden voi tarkistaa nopeammin. Pelkkä syntaksivirheiden nopea korjaaminen ei kuitenkaan tee ohjelmointityöstä kovin dialogista. Ohjelman toimintavirheiden löytäminen vaatii sen käynnistämistä, ja silloin-

kaan ohjelman ajonaikaisesta käyttäytymisestä ei yleensä saa kovin tarkkaa tietoa. Virheenjäljitystyökaluilla ohjelman toimintaa voidaan seurata paremmin, askel kerrallaan, jolloin dialogi on tiiviimpää. Ajasta on tullut siis jossain määrin käsiteltävä abstraktio ohjelmoijalle, mikä tiivistää dialogia ja helpottaa ohjelmointityötä.

Dialogin kannalta merkittävää on mahdollisuus kokeilla ja saada välitöntä palautetta. Ohjelmoijan tulisi voida kokeilla ohjelman eri osia erilaisilla syötteillä. Ohjelmointikieli tukisi silloin parhaiten hiljalleen kehittyvää inhimillistä ajattelua. Jatkuvan palautteen idea on keskeinen myös Extreme Programming -menetelmässä ja siihen liittyvässä yksikkötestauksessa. Koko ohjelmointiprosessin ajan jatkuvan yksikkötestauksen onkin todettu lisäävän ohjelmointityön tehokkuutta [Bec00].

Ohjelmoijan ajatusten dialogi voi päätyä umpikujaan - kuten kirjailijalle voi tulla kirjoituseste (engl. writer's block) dialogissaan tarinan ja itsensä kanssa. Este tilanteessa näkökulman vaihtaminen on todettu hyväksi ratkaisuksi. Jotta eri näkökulmien kautta käytävä ohjelman kehittämisen dialogi on mahdollinen, vaatii se monenlaisia tarkastelukulmia, siis tukea useille ohjelmointiparadigmoille.

Perinteiset moniparadigmaiset ympäristöt eivät täytä vaatimusta, koska niissä ohjelma on paloiteltu eri paradigmojen kesken. Eri tehtäviin voi käyttää parhaiten sopivaa paradigmaa, esimerkiksi perinteisessä kolmitasoarkkitehtuuriin perustuvassa web-sovelluksessa sovelluslogiikan voi ohjelmoida olioparadigman avulla, kun taas tietorakenteen luoda relaatioparadigman avulla. Paradigmojen välillä vaihtaminen on kuitenkin hankalaa: ongelma tunnetaan nk. impedanssivastaamattomuutena (engl. impedance mismatch) [GAO95].

Perinteisessä moniparadigmaisessa ympäristössä paradigman vaihdokset tapahtuvat tarkasteltavan sovelluksen osan vaihtuessa, eivät ohjelmoijan tahdon mukaan. Paradigman vaihdokset ovat hyödyllisiä, mutta suurin hyöty niistä on kun ohjelmoija voi itse päättää vaihdoksista. Toisin sanoen samaa koodia tulisi pystyä tarkastelemaan eri paradigmoissa. Kielen ei tulisi olla "heterogeenisesti" moniparadigmainen, vaan "homogeenisesti" moniparadigmainen.

Eri paradigmat soveltuvat eri tehtäviin. Kaikille paradigmoille voidaan löytää tilanteita, joihin ne soveltuvat huonosti. Koska yleiskäyttöisten ohjelmointikielten tulee kuitenkin olla Turing-täydellisiä, eli niiden tulee mahdollistaa kaikkien nykyaikaisella tietokoneella ratkaistavissa olevien ongelmien ratkaiseminen, tulee myös paradigman kannalta hankalat tehtävät hoitaa tavalla tai toisella [Tur36].

Homogeenisesti moniparadigmainen ohjelmointikieli voisi sen sijaan sisältää laskennallisesti epätäydellisiä paradigmoja. Koko ohjelmaa ei siis voisi hahmottaa yhden paradigman avulla. Ajatus on kohtuullisen radikaali, koska yleiskäyttöisten kielten Turing-täydellisyyttä on totuttu pitämään lähes itsestäänselvyytenä. Tulee kuitenkin huomata, että yhden paradigman epätäydellisyys ei tee koko kielestä Turing-epätäydellistä. Ihminen hahmottaa ulkomaailmaa useiden näkökulmien avulla, joista yksikään ei kerro koko totuutta. Monen paradigman kautta toimiminen voisi olla siis hyvin luontevaa, vaikka ne olisivatkin epätäydellisiä.

Epätäydellisyydestä seuraisi merkittäviä etuja. Yksittäisen paradigman syntaksi voisi olla hyvinkin yksinkertainen, koska sen avulla tulisi voida esittää vain siihen paradigmaan hyvin soveltuvat ohjelman osat. Tällöin dialogi tietokoneen ja ohjelmoijan välissä tulisi yksinkertaisemmaksi.

Teknisesti ohjelmointikieli vaatisi palvelinohjelmistoa taakseen, se siis muistuttaisi nykyisiä sovelluspalvelimia, mutta veisi palvelimen roolin paljon pidemmälle. Kieli olisi järkevintä toteuttaa Javan tai muun yleisesti käytössä olevan ohjelmointiympäristön päälle. Valmis ympäristö kirjastoineen tarjoaa paljon hyväksi havaittuja abstraktioita, joten tyhjän päälle rakentaminen ei ihmisajattelunkaan näkökulmasta ole järkevää.

Ihmisen historiassa ulkopuolisten apuvälineiden käyttö oman ajatteluprosessin osana on ollut merkittävä piirre. Taito käyttää kieltä, kirjoitustaitoa ja muita apuvälineitä rajallisen henkisen potentiaalin jatkeena on mahdollistanut omalta osaltaan ihmiskunnan kehityksen [LL78]. Ohjelmointikielet ovat nykyajan tehokkain tietojenkäsittelyn apuväline, joten mielestäni myös niiden pitäisi tulla tiiviimmin ohjelmoijan ajatteluprosessin osaksi.

8 Yhteenveto

Tietokoneen arkkitehtuuri tai laskettavuuden formalismit eivät tarjoa ratkaisuja tärkeisiin ohjelmointikielten suunnittelua koskeviin ongelmiin. Ohjelmointikielten rakennetta käsittelevä klassinen kirjallisuus antaa hyviä neuvoja kielten suunnitteluun, mutta neuvot perustuvat pääasiassa kirjoittajien omakohtaiseen pohdiskeluun. Tässä tutkielmassa pyrittiin etsimään yhteistä pohjaa ohjelmointikielten suunnittelulle ihmisen ajattelua tutkivien tieteiden tuloksista.

Yhteenvetona tutkielmassa löydetyistä suunnitteluperiaatteista esitin visioni ohjelmointikielestä, joka toimisi mahdollisimman hyvin ajattelun apuvälineenä. Vision tärkein ajatus oli ohjelmointikielen tuominen tiiviimmin ohjelmoijan ajatteluprosessin osaksi.

Nopea tietotekninen kehitys on vähentänyt ohjelmointiin liittyviä teknisiä rajoituksia. Valinnanvapauden liittyy kuitenkin valinnan vaikeus. Mihin suuntaan ohjelmointikielten kehityksen tulisi kulkea? Uskoakseni kognitiotieteellä ja psykologialla on vielä paljon annettavaa ohjelmointikielten kehityksen suunnannäyttäjinä.

Lähteet

- Air03 Airaksinen, T., *Tekniikan suuret kertomukset, Filosofinen raportti*. Ota-
va, 2003.
- Bec00 Beck, K., *Extreme programming explained: embrace change*. Addison-
Wesley Longman Publishing Co., Inc., 2000.
- Ber93 Berard, E. V., *Essays on Object-Oriented Software Engineering, Volume
I*. Prentice-Hall, 1993.
- BGvN61 Burks, A. W., Goldstine, H. H. ja von Neumann, J., Preliminary discus-
sion of the logical design of an electronic computing instrument, second
edition. Teoksessa *Collected Works, Vol. 5*, Pergamon Press Ltd., 1961,
sivut 34–80.
- Boo91 Booch, G., *Object Oriented Design with Applications*. The Benja-
min/Cummings Publishing Company, Inc., 1991.
- Cra72 Craik, F., Levels of processing; a framework for memory research. *Jour-
nal of Verbal Learning and Verbal Behavior*, 11, sivut 671–684.
- Csi97 Csikszentmihalyi, M., *Creativity*. HarperCollins Publishers, 1997.
- Dij65 Dijkstra, E. W., Programming considered as a human activity. *Proc.
IFIP Congress 1965*, sivut 213–217.
- Dij68 Dijkstra, E. W., Letters to the editor: Go-To statement considered
harmful. *Communications of the ACM*, 11,3(1968).
- Fow99 Fowler, M., *Refactoring: Improving the Design of Existing Code*.
Addison-Wesley, 1999.
- FPB87 Frederick P. Brooks, J., No silver bullet: essence and accidents of softwa-
re engineering. *Computer*, 20,4(1987), sivut 10–19.

- GAO95 Garlan, D., Allen, R. ja Ockerbloom, J., Architectural mismatch, or, why it's hard to build systems out of existing parts. *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, April 1995, sivut 179–185.
- GHJV95 Gamma, E., Helm, R., Johnson, R. ja Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, 1995.
- Hak02 Hakala, J. T., *Luova prosessi tieteessä*. Gaudeamus, 2002.
- Hin90 Hintzman, D. L., Human learning and memory: connections and dissociations. *Annual Review of Psychology*, 41, sivut 109–139.
- Hoa89 Hoare, C. A. R., Hints on programming language design. Teoksessa *Essays in Computing Science*, Prentice Hall, 1989, sivut 193–217.
- Hof97 Hofstede, G., *Cultures and Organizations: Software of the Mind*. McGraw-Hill, 1997.
- KHH⁺01 Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. ja Griswold, W. G., An overview of AspectJ. *Lecture Notes in Computer Science*, 2072, sivut 327–355. URL citeseer.nj.nec.com/kiczales01overview.html. Tarkistettu 24.2.2003.
- KJH98 Koski-Jännes, A. ja Hänninen, V., Dialogiset prosessit ja riippuvuudesta vapautuminen. Teoksessa *Sosiaalinen vuorovaikutus*, Lahikainen, A.-R. ja Pirttilä-Backman, A.-M., toimittajat, Otava, 1998, sivut 173–191.
- Kos98 Koski, J. T., *Infoähky ja muita kirjoituksia oppimisesta, organisaatiosta ja tietoyhteiskunnasta*. Gummerus, 1998.
- Lar01 Larman, C., *Applying UML and patterns, An introduction to Object-oriented analysis and design and the Unified Process, Second Edition*. Prentice-Hall, 2001.

- LH02 Lieberherr, K. J. ja Holland, I., Preventive maintenance of object-oriented software. 2002.
- LHR88 Lieberherr, K. J., Holland, I. ja Riel, A. J., Object-oriented programming: An objective sense of style. *OOPSLA*, numero 11, September 1988.
- Lie98 Liebkind, K., Etnisten ryhmien identiteettineuvottelut. Teoksessa *Sosiaalinen vuorovaikutus*, Lahikainen, A.-R. ja Pirttilä-Backman, A.-M., toimittajat, Otava, 1998, sivut 100–120.
- LL78 Leakey, R. ja Leakey, L., *Ihmisen synty*. Kirjayhtymä, 1978.
- Mea34 Mead, G., *Mind, Self and Society*. University of Chicago Press, 1934.
- Mey86 Meyer, B., Genericity versus inheritance. *Conference proceedings on Object-oriented programming systems, languages and applications*. ACM Press, 1986, sivut 391–405.
- Mil56 Miller, G. A., The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63, sivut 81–97.
- Nei63 Neisser, U., *The imitation of man by machine*. Numero 139. 1963.
- Par72 Parnas, D., On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15,12(1972).
- Pre00 Pressman, R. S., *Software Engineering, A Practitioner's Approach*. McGraw-Hill, 2000.
- Ril87 Riley, H. N., The von Neumann architecture of computer systems. URL <http://www.csupomona.edu/~hnriley/www/VonN.html>. Tarkistettu 24.4.2003.
- Roe80 Roediger, H., Memory metaphors in cognitive psychology. *Memory and Cognition*, ,8, sivut 231–246.

- Saa90 Saariluoma, P., *Taitavan ajattelun psykologia*. Otava, 1990.
- Sim79a Simon, H. A., How big is a chunk? Teoksessa *Models of Thought*, Yale University Press, 1979, sivut 50–62.
- Sim79b Simon, H. A., Motivational and emotional controls of cognition. Teoksessa *Models of Thought*, Yale University Press, 1979, sivut 29–39.
- Sim82 Simon, H. A., *The Sciences of the Artificial, Second Edition*. The MIT Press, 1982.
- Str94 Stroustrup, B., *The Design and Evolution of C++*. Addison-Wesley, 1994.
- STSL97 Simon, H. A., Tabachneck-Schijf, H. J. ja Leonardo, A. M., Camera: A computational model of multiple representations. *Cognitive Science*, 21,3(1997), sivut 305–350.
- Tur36 Turing, A., On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, numero 42, 1936, sivut 230–265, URL <http://www.abelard.org/turpap2/tp2-ie.asp>. Tarkistettu 24.4.2003.
- WH89 Wirth, N. ja Hoare, C. A. R., A contribution to the development of Algol. Teoksessa *Essays in Computing Science*, Prentice-Hall, 1989, sivut 31–45.
- Wir74 Wirth, N., On the design of programming languages. *IFIP Congress 74*.