Tietojenkäsittelytiede **31** Marraskuu 2010 sivut 22–42
Toimittaja: Ari Korhonen

Arkkitehtuurikomponentin laadun arvioiminen mittaamalla

Sami Hyrynsalmi
Turun yliopisto
Informaatioteknologian laitos
TUCS – Turun tietotekniikan tutkimuskeskus
sthyry@utu.fi

Tiivistelmä

Olioparadigmalle on määritelty useita satoja ohjelmistomittoja sisäisen laadun arvioimiseksi, mutta suurin osa näistä on keskittynyt pienimpien rakenneosasten, metodien ja luokkien, tarkastelemiseen. Mielenkiintoisesti korkeamman abstraktiotason rakenteille, kuten luokkajoukoille, ei kuitenkaan ole esitelty kuin yksittäisiä mittoja. Ohjelmistojen koon kasvaessa mielekkään käsiteltävän yksikön koko on kasvanut. Myös käytettävien ohjelmistomittojen on kasvettava palvelemaan kiihtyvän kehityksen tarpeita. Tässä työssä käsitellään erilaisia luokkien muodostamille joukoille, eli komponenteille, esitettyjä mittoja. Tarkastelemme myös mittojen määrittelyn ongelmia ja niiden kelpuuttamista sekä sitä, mitä useiden luokkien muodostamista joukoista voitaisiin mitata.

1 Johdanto

Ohjelmistomitat ovat perinteinen, mutta kiistelty keino pyrkiä parantamaan ohjelmiston laatua. Tosin mittojen hyödyllisyydestä ja käyttökelpoisuudesta on käyty pitkään keskustelua. Esimerkiksi El Emam ym. [1] kyseenalaistavat suurimman osan mitoille tehdyistä empiirisistä tutkimuksista puutteellisina.

Ihanteellisesti mittoja voitaisiin käyttää tunnistamaan nopeasti laadultaan ongelmallisia ohjelmakohtia ja valitsemaan nämä tarkempaan tarkasteluun sekä testaukseen. Tällaiseen käyttötarkoitukseen on esitetty useita mittoja niin metodi- kuin luokkatasolla. Mielenkiintoisesti komponenteille eli luokkien muodostamille joukoille ei ole kuitenkaan määritelty kuin pieni joukko tällaisia mittareita.

Komponenttimitoille olisi silti käyttötarkoituksensa, sillä jo nykyiset ohjelmat koostuvat tuhansista luokista ja kymmenistä joukoista. Suurimmat itsenäiset ohjelmat rakennetaan tuhansista komponenteista.

Perinteisessä top-down -suunnittelussa komponenttimitat ovat myös käytettävissä aikaisemmassa vaiheessa kuin matalan tason yksityiskohtiin luottavat luokkamitat. Näin mittojen karkeilla versioilla saatua tietoa voitaisiin käyttää jo arkkitehtuurisuunnitelman parantamiseen ennen itse toteutusvaiheen aloittamista.

Tässä työssä esitellään komponenteille määriteltyjä mittoja sekä tarkastellaan mitä piirteitä komponenteista voisi mahdol-

OHJE KIRJAPAINOLLE: B5-arkin vasen- ja yläreuna kohdistetaan A4-arkin vasempaan ja yläreunaan. Näin pitäisi marginaaliksi tulla taitteen puolella noin 33 mm ja muualla noin 22 mm.

lisesti mitata. Tutkimme myös miten tunnettu komponenttimitta soveltuu komponenttien suunnitteluongelmien tunnistamiseen.

Seuraavaksi esitellään ohjelmistojen mittaamista yleisesti. Kolmannessa kohdassa esitellään useisiin mittausohjelmiin jo toteutettu ns. *Martinin metriikka*, joka oli yksi ensimmäisistä komponenttimitoista. Tämän jälkeen työssä keskustellaan mitä ominaisuuksia tai piirteitä ohjelmakomponenteista kannattaisi mitata. Viidennessä kohdassa esitellään lyhyesti empiirinen koe, jossa arvioidaan Martinin metriikan hyödyllisyyttä laatuindikaattorina.

2 Ohjelmistojen mittaaminen

Ohjelmistomitoiksi voidaan mieltää kaikki mittarit, jotka jollain tavalla käsittelevät ohjelmistoa tai sen kehitystä. Fenton ja Pfleeger [2] ovat esittäneet tunnetun jaottelun ohjelmistomitoille, joka kuvaa hyvin mittojen laajaa käyttöaluetta. Heidän kategorisoinnissaan on kolme ryhmää: *prosessi-*, *resurssi-* ja *tuotemitat*.

Prosessimitat arvioivat kehitysprosessia kokonaisuutena tai sen yksittäisiä aktiviteettejä. Vastaavasti resurssimitat arvioivat ohjelmistonkehitysprosessiin liittyviä resursseja. Esimerkiksi tällaisilla resurssimitoilla voidaan tutkia kehitystiimin kokoa tai ryhmän tuottavuutta. Tuotemitat taas arvioivat kehitysprosessissa tuotettuja artefakteja.

Tässä työssä keskitytään vain yksittäiseen tuotemittojen alaryhmään: ohjelmaa sisältä käsin arvioiviin laatumittareihin.

2.1 Lähdekoodiin ja rakenteeseen pohjautuvat tuotemitat

Tunnetuin sisäinen tuotemitta on todennäköisesti lähdekoodirivien lukumäärä (LOC, engl. *Lines of Code*) ja sen useat eri muunnokset. LOC:a on käytetty niin mittojen normalisoinnissa kuin arvioimaan työn tuottavuutta ja ohjelman kehittämiseen vaadittavaa työmäärää. Koodirivien lukumäärä on kuitenkin huomattavan yksinkertainen mitta ja sen käyttämistä muuhun kuin ohjelman koon arvioimiseen on kritisoitu [3].

Toinen perinteinen sisäisen laadun mitta on vuosikymmeniä sitten esitelty McCaben syklomaattinen kompleksisuus [4]. Syklomaattinen kompleksisuus mittaa yksittäisen ohjelman osan, yleensä yhden rutiinin, itsenäisten suorituspolkujen lukumäärää eli kuinka monta erilaista reittiä pitkin ohjelmapalasen suoritus voi edetä.

Syklomaattista kompleksisuutta voidaan käyttää tunnistamaan virhealttiita osia ohjelmakoodista, sillä rakenteellisesti monimutkaisempaan ohjelmakoodiin saattaa helposti kätkeytyä virheitä. Tällaiset virheet saattavat näkyä jopa loppukäyttäjälle asti ohjelman virheellisenä toiminnallisuutena. Syklomaattisen kompleksisuuden käyttäminen mahdollisten virheiden tunnistamiseen ja poistamiseen vaikuttaa myönteisesti sekä ohjelman sisäiseen että ulkoiseen laatuun. Syklomaattisen kompleksisuuden määritelmää on tosin myös kritisoitu useasti [2, 5].

McCaben mitta kuuluu silti klassisimpiin sisäisiin tuotemittoihin. Sitä käytetään yhä edelleen huolimatta useista osoitetuista puutteista. Uudella vuosituhannella on kuitenkin innostuttu Chidamberin ja Kemererin [6, 7] töiden jälkeen määrittelemään ohjelmointiparadigmakohtaisia mittoja erityisesti olio-orientoituneille kielille. Nykyään olioille määriteltyjä mittoja on kymmeniä, ellei satoja, ja uusia mittoja määritellään jatkuvasti.

Chidamberin ja Kemererin oliomittapaketti koostuu kuudesta mitasta [7]:

WMC Painotettujen metodien lukumäärä (engl. *Weighted Methods per Class*) saadaan laskemalla yhteen luokan jokaisen metodin kompleksisuus.

DIT Perintäpuun syvyys (engl. *Depth of Inheritance Tree*) kertoo kuinka syvällä perintähierarkiassa luokka on.

NOC Lasten lukumäärä (engl. *Number of Children*) on luokasta suoraan perivien luokkien määrä.

CBO Luokkien välinen kytkeytyneisyys (engl. *Coupling Between Objects*) on niiden luokkien määrä, joiden kanssa tarkasteltava luokka on kytkeytynyt.

RFC Luokan vastausmäärä (engl. *Response For a Class*) on luokan omien metodien lukumäärä laskettuna yhteen kaikkien luokan kutsumien metodien määrällä.

LCOM Koheesion puute metodeissa (engl. *Lack of Cohesion in Methods*) arvioi luokan yhtenäisyyttä sen metodien kautta.

Chidamberin ja Kemererin mittapakettia on arvioitu empiirisissä kokeissa toimivaksi [8, 9], mutta myös kritisoitu teoreettisen taustan ongelmista [10, 11, 12]. Heidän määrittelemänsä mitat kuitenkin kuvaavat hyvin niitä piirteitä, joita useat tutkijat ovat aikaisemmin ja myöhemmin pyrkineet mittaamaan: *koheesio*, *kytkeytyminen*, *perintä* sekä *rakenteellinen kompleksisuus*.

2.2 Mittojen abstraktiotasot

Olio-orientoituneille kielille esitellyt mitat voidaan pyrkiä jakamaan vielä pienempiin ryhmiin erilaisin tavoin, mutta tässä pidättäydytään tarkastelemaan ainoastaan abstraktiotasopohjaista ryhmittelyä. Briand, Daly ja Wüst [13] esittelivät jaon viiteen ryhmään matalimmasta abstraktiotasosta korkeimpaan: *attribuutti*, *metodi*, *luokka*, *luokkajoukko* ja *järjestelmä*.

Ensimmäiset kolme ryhmää ovat intuitiivisia, mutta neljännen määritelmä on ongelmallinen. Briandin ym. määrittelemä luokkajoukko saattaa tarkoittaa mitä tahansa mielivaltaista joukkoa luokkia järjestelmästä. Tällaiselle joukolle koheesion laskeminen saattaisi esimerkiksi olla mielekästä, mutta se ei kuitenkaan ole luonnollinen määritelmä korkeammalle abstraktiotasolle.

Huomattavasti mielenkiintoisempaa on käsitellä joukkoa luokkia, joilla on jokin *syy* esiintyä yhdessä. Syy voi esimerkiksi olla se, että kehittäjä on päättänyt tietoisesti sijoittaa luokat samaan joukkoon. Tällaisia ovat esimerkiksi Javan paketit, joiden tarkoituksena on koota yhtenäistä palvelua tarjoavat osaset yhteen. Ilmeisesti Briand ym. tarkoittivat juuri tällaisia kokonaisuuksia määritelmässään, mutta he eivät kuitenkaan tarkentaneet luokitteluaan tarpeeksi.

Luokkajoukon rinnalle mielekäs määrittelytaso on *komponentti*, jolla tässä yhteydessä tarkoitetaan luokkien muodostamaa joukkoa. Joukkojen mahdollisesti sisältämiä alijoukkoja ei huomioida. Samaistamme komponentin käsitettä ohjelmistoarkkitehtuurien yhteydessä käytettäviin komponentteihin, sillä arkkitehtuurikomponentit ovat jo käytössä oleva osa ohjelmistokehitystä.

Javassa tarkoituksella yhteen kootut luokkajoukot on helppo tunnistaa, sillä ne ilmaistaan eksplisiittisesti kielen pakettimekanismilla. Komponenttimitat eivät kuitenkaan ole kielikohtaisia. Esimerkiksi eri ohjelmointikielten nimiavaruusmekanismit tai fyysisten lähdekooditiedostojen sijoittaminen samaan hakemistoon voidaan tulkita implisiittiseksi komponenttimekanismiksi.

2.3 Mitan kelpuuttaminen

Ohjelmistomitan kelpuuttamisen (engl. *validation*) tarkoituksena on varmistaa mitan oikeellisuus ja käyttökelpoisuus. Kelpuuttaminen voidaan jakaa kahteen vai-

heeseen: teoreettiseen vahvistamiseen ja empiiriseen hyväksymiseen. [13] Teoreettisessa kelpuuttamisessa osoitetaan, että mitta mallintaa sitä ominaisuutta, jota sen pitäisi mitata. Tavoitteena on osoittaa esimerkiksi koheesiomitan noudattavan yleisiä koheesion mittaamisen periaatteita. Esimerkiksi eräs koheesiomittojen ehto vaatii, että jos kaksi kytkeytymätöntä luokkaa yhdistetään, ei muodostuneen luokan koheesiomitan arvo saa olla suurempi kuin alkuperäisten luokkien [14].

Empiirisessä kelpuuttamisessa taas osoitetaan mitan käyttökelpoisuus yhdistämällä sen arvot ulkoisiin laatuominaisuuksiin. Esimerkiksi monimutkaista ohjelmakoodia on hankala ymmärtää ja vaikea testata täydellisesti, joten tällaiset ohjelmakohdat aiheuttavat käyttäjille näkyviä häiriöitä todennäköisesti enemmän kuin helpommin ymmärrettävät moduulit. Tämän vuoksi kompleksisuusmittojen käyttökelpoisuutta pitäisi arvioida tilastollisesti mitattavaan osaan liittyvien ohjelmointivirheiden määrän kanssa. Käyttökelpoisen kompleksisuusmitan pitäisi korreloida selvästi häiriöiden määrän kanssa.

Teoreettiseen tarkasteluun on esitetty muutamia hyväksymiskehyksiä. Näistä tunnetuimpien joukossa ovat Weyukeriin [15] kompleksisuusehdot ja Briandin, Morascan sekä Basilin [14] kriteerit. Esimerkiksi Chidamberin ja Kemererin [7] mittapakettia on yritetty kelpuuttaa Weyukerin vaatimuksilla. Tosin paketin mitoista ainoastaan WMC:tä voi luonnehtia puhtaaksi kompleksisuusmitaksi.

Kaikkia metriikoita ei pitäisikään kohdella monimutkaisuusmittoina. Esimerkiksi kahden kytkeytyneen komponentin yhdistäminen vähentää järjestelmän kokonaiskytkeytymistä, ja näin *”kytkeytymiskompleksisuus”* vähenee. Tämä on ristiriidassa Weyukerin ehtojen kanssa, joiden mukaan kahden erillisen moduulin yhdistämisestä syntyneen osan monimutkaisuus ei voi pienentyä alkuperäisten yhteenlasketusta arvoista.

Briand ym. [14] esittelivät koheesiolle, kytkeytymiselle, kompleksisuudelle sekä koolle erilliset vaatimukset, jotka soveltuvat yleisiä kompleksisuusehtoja paremmin näiden ominaisuuksien mittojen arvioimiseen. Teoreettiset kelpuuttamiskehykset tarvitsevat kuitenkin vielä jatkokehitystyötä, sillä esimerkiksi Briandin ym. kehys ei sovellu ”suhteellisten” mittojen arvioimiseen [16].

Aksiomaattinen kelvollisuuden todistaminen ei kuitenkaan yksinään riitä. Esimerkiksi Cherniavsky ja Smith [17] johtivat kompleksisuusmitan, joka täyttää Weykerin kompleksisuusehdot, mutta ei intuitiivisesti ole kelvollinen rakenteellisen monimutkaisuuden arvioimiseen. Tämän vuoksi mittoja pitää aina myös koetella empiirisessä testissä, jossa tarkoituksena on osoittaa, että mitta on käyttökelvollinen myös kehitystyössä.

Briand, Daly, ja Wüst [13] vaativatkin, että mitan suhde ohjelmiston ulkoisiin laatutekijöihin on aina osoitettava empiirisesti.

3 Komponenttimitat

Huolimatta ohjelmistomittojen pitkästä historiasta sekä olio-orientoituneiden kielten mittareiden herättämästä kiinnostuksesta, arkkitehtuurikomponenteille on esitetty vain muutamia mittoja. Näistä tärkeimpiä ovat Martinin komponenttimitat [18], Ponision kontekstiriippuvainen koheesiomitta [19], Meltonin ja Temperon [20] CRSS sekä Zhoun ym. [21] koheesiomitta SCC. Lisäksi Ducasse, Lanza ja Ponisio [22] ovat määritelleet ohjelmistokomponenttien perhoskuvaajat (engl. *Butterflies*), jotka pyrkivät muutaman yksinkertaisen mittarin avulla visualisoimaan komponentin luonteen helposti

ymmärrettäväksi kuvaajaksi. Komponenteille ominaisten mittojen ohella on sovellettu joukkoa alemman abstraktiotason mittoja. Seuraavassa on tarkasteltu yleistämisen ongelmia sekä esitellään Martinin komponenttimittoja.

3.1 Yleistäminen alemmilta abstraktiotasoilta

Suoraviivaisin ratkaisu täyttää komponenttimittojen puute on käyttää alemman abstraktiotason mittaa korkeammalla tasolla. Esimerkiksi Ponisio ja Nierstrasz [19] määrittelivät useita muunnoksia luokkatason mitoista komponenteille. Samoin sekä Lindvall, Tesoriero ja Costa [23] että Bengtsson [24] ovat esitelleet Chidamberin ja Kemererin luokkatason mittojen muunnelmia komponenttitasolle. Yacoub, Ammar ja Robinson [25] puolestaan määrittelivät syklomaattisen kompleksisuuden ohjelmakomponenteille.

Briand ym. [13] esittivät, että minkä tahansa luokkamitan voi suoraan peilata ylemmälle tasolle. He käyttivät tästä esimerkkinä tietovuohon perustuvaa kytkeytymismittaa ICH:ta.

ICH lasketaan yhdelle metodille summana sen polymorfisesti muista luokista herättämien metodien kutsukerroista, painotettuna kutsuvan piirteen parametrien määrällä, eli kuinka paljon informaatiota virtaa pois kyseisestä metodista. Kytkeytymismitta myös skaalautuu isommille kokonaisuuksille: Luokan ICH on sen kaikkien metodien informaatiovirtojen summa ja vastaavasti luokkajoukon ICH on sen kaikkien luokkien arvojen summa. [13]

Luokalle laskettava ICH on osin intuitiivinen, sillä ymmärrettävästi luokan informaatiovirta on sen metodien virtojen summa. Luokkajoukoille ICH toimii epämääräisesti, sillä luokkajoukon tietovuokytkeytyminen laskee mukaan joukon sisäiset kytkökset. Näin ICH:ta ei voi käyttää tarkasteltaessa osajärjestelmän kytkeytyneisyyttä muuhun ohjelmaan.

ICH:n käyttäminen luokkajoukon sisäisen kytkeytymisen mittana – eli itse asiassa joukon kiinnevoimaisuuden arvioimiseen – on myös mahdotonta, sillä laskettaessa yhteen yksittäisten luokkien kytkeytymisiä huomioidaan myös luokkajoukon ulkopuoliset kytkökset. Mitan käyttäminen koko järjestelmän kytkeytymisen arvioimiseen ei ole myöskään mielekästä, sillä mittaa ei ole normalisoitu. Tällöin keskenään erikokoisten järjestelmien vertaaminen on mahdotonta, sillä pelkästään metodien määrä vaikuttaa huomattavasti mitan arvoon.

Luokkamittojen on esitetty skaalautuvan luokkajoukoille ja järjestelmille sopiviksi, mutta ainakin ICH:n summaperustainen ratkaisu jättää tarkasteluun huomattavia puutteita. Samanlaisia ongelmia voidaan osoittaa myös muissa suoraan yleistetyissä mitoissa, minkä vuoksi olisi hyvä määritellä myös komponenteille ominiaisia mittoja.

3.2 Martinin komponenttimitat

Robert C. Martin on ollut näkyvimpiä komponenttimittojen kehittäjiä. Hänen alun perin viime vuosituhannen lopulla esittelemänsä Martinin metriikka on yksi ensimmäisistä ja selvästi tunnetuin komponenttimitta; metriikka on toteutettu kymmeniin mittausohjelmiin ja sitä on käytetty useissa tutkimuksissa. Mielenkiintoisesti Martinin metriikan tai hänen myöhemmin esittelemänsä komponenttien koheesiomitan teoreettisia taustoja tai empiiristä toimivuutta ei kuitenkaan ole aiemmin tarkasteltu.

Martinin metriikka

Kehittäjänsä mukaan nimetty Martinin metriikka koostuu kuudesta apumitasta ja

näiden avulla johdetusta päämitasta, jota määrittelijä kutsuu suorasukaisesti komponentin laatumitaksi [18]. Metriikka pyrkii arvioimaan arkkitehtuurikomponentin laatua sen vakauden ja abstraktiuden suhteella.

Metriikka perustuu kahteen Martinin [18] esittelemään ohjelmakomponentin suunnittelun ohjenuoraan. Näistä ensimmäisen, *vakaan riippuvuuden periaatteen* mukaan komponentin pitäisi olla riippuvainen ainoastaan sitä vakaammista komponenteista. Epävakaiden osien muuttuessa niistä riippuvaisia komponentteja joudutaan muuttamaan ja päivittämään yhteensopiviksi uudempien versioiden kanssa. Tämän vuoksi komponentin tulisi olla riippuvainen vain vakaista palveluista, jolloin komponenttia ei tarvitse päivittää palveluntarjoajan muuttuessa.

Vakaus ei tarkoita Martinille ohjelmakomponentin muutosten epätodennäköisyyttä, vaan muutoksen tekemisen vaativuutta. Hän määritteli vakaan komponentin sellaiseksi, jolla on paljon vastuuta ja vain vähän syitä muuttua. Jokainen luokka, joka on riippuvainen komponentin palveluista lisää komponentin vastuita. Vastaavasti jokainen luokka, josta komponentti on riippuvainen, on mahdollinen syy muutokselle.

Martin lähestyy vakauden mittaamista käänteisesti määrittelemällä epävakauden mitan, joka pohjautuu komponenttiin tuleviin ja lähteviin riippuvuuksiin. Epävakausmitta I lasketaan kaavalla:

$$I = \frac{C_e}{C_a + C_e} \quad . \tag{1}$$

Kaavassa C_e on komponentin ulkopuolisten luokkien lukumäärä, jotka ovat riippuvaisia moduulin luokista. C_a on vastaavasti komponentin ulkopuolisten luokkien lukumäärä, joista moduulin sisäiset luokat ovat riippuvaisia.

Mitasta käytetään myös versiota, jossa luokkien määrä on korvattu komponenttien määrällä [26].

Epävakauden arvot vaihtelevat välillä [0, 1]. Arvolla nolla komponentti on vakaa, sillä se ei ole riippuvainen yhdestäkään toisesta paketista. Vastaavasti arvolla yksi, komponenttiin tulevia riippuvuuksia ja vastuita ei ole, jolloin komponentti on epävakaa.

Martinin toinen ohjenuora, *vakaan abstraktiuden periaate*, vaatii vakaista komponenteista abstrakteja. Vakaan riippuvuuden periaatetta noudattaen rakennettujen vakaiden komponenttien muuttamisesta on tehty vaikeata, sillä näistä riippuvia komponentteja on paljon ja niiden muuttaminen työlästä. Silti komponenttien tuottamia palveluita pitäisi pystyä laajentamaan. Martinin ehdottama ratkaisu on tehdä vakaista komponenteista abstrakteja. Tällaisten komponenttien rajapintoja on helppo kasvattaa, sillä komponentin palvelut eivät voi laajennuksesta muuttua.

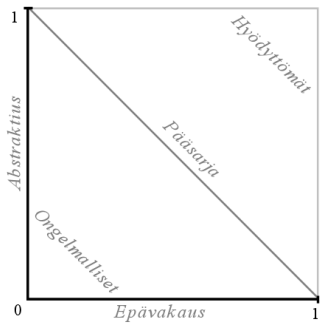
Abstraktisuudelle määriteltiin myös oma mitta:

$$A = \frac{N_a}{N_c} \quad . \tag{2}$$

Kaavassa N_c on komponentin luokkien määrä ja N_a komponentin abstraktien luokkien määrä. Abstraktiuden arvot on skaalattu välille [0, 1], missä arvolla nolla komponentissa ei ole yhtään abstraktia luokkaa.

Kahden ohjenuoran noudattamista voidaan havainnollistaa IA-kuvaajalla, joka on esitetty kuvassa 1. Kuvaajassa vaaka-akselilla on komponentin epävakaus ja pystyakselilla abstraktius. Ihanteellisessa suunnittelussa kaikki komponentit olisivat joko täysin vakaita ja abstrakteja tai konkreettisia ja epävakaita. Ensimmäisessä tapauksessa komponentit sijoittuisivat kuvaajan pisteeseen (0,1) ja toisessa tapauksessa pisteeseen (1,0). Käytännössä tällainen puhdas kaksijako on harvinaista.

Kuvaajaa tarkasteltaessa huomataan,



Kuva 1: Martinin metrikan IA-kuvaaja, jossa hyvää suunnittelua noudattavat komponentit ovat lähellä pääsarjaa.

että lähellä pistettä (1, 1) sijaitsevat komponentit ovat abstrakteja ja niistä riippuvia luokkia on vain muutama. Tällaiset komponentit ovat arkkitehtuurissa selvästi hyödyttömässä asemassa, sillä niiden abstraktisuus lisää kompleksisuutta. Samalla komponenttien yleisyyden tuomia etuja ei hyödynnetä.

Samoin pisteen (0, 0) läheiset komponentit edustavat huonoa suunnittelua. Nämä ohjelmakomponentit ovat täysin konkreetteja ja samalla vastuussa monelle taholle. Näiden muuttaminen on Martinin mukaan työlästä suuren vastuun takia ja konkreettisuus tekee niiden laajentamisesta hankalaa.

Käytännössä kaikkien komponenttien suunnitteleminen lähelle kuvaajan pisteitä (0,1) tai (1,0) on haasteellista. Komponentit eivät kuitenkaan saisi sijaita liian lähellä ongelmallisten tai hyödyttömien komponenttien aluetta kuvaajalla. Tämän vuoksi komponenteille pitäisi pyrkiä löytämään ihanteellinen suhde abstraktiuden ja vakauden välillä. Martin ehdottaa ratkaisuksi suoraa hyvän suunnittelun mukaisten pisteiden (1,0) ja (0,1) välillä – tällöin komponenteilla on sopivassa suhteessa vastuuta ja vapautta. Martin nimittää tätä suoraa *pääsarjaksi* (engl. *Main Sequence*).

Varsinainen Martinin metriikka on komponentin etäisyys hyvää suunnittelua edustavasta pääsarjasta:

$$D = \frac{|A+I-1|}{\sqrt{2}} \text{ ja} \quad (3)$$

$$D' = |A+I-1| \ . \quad (4)$$

D on komponentin etäisyys pääsarjasta ja sen arvot vaihtelevat välillä $[0, \frac{1}{\sqrt{2}}]$. D' on mitan normalisoitu etäisyys, jolloin komponenttien arvot kuuluvat välille [0,1]. Myöhemmin puhuttaessa Martinin metriikasta tarkoitetaan normalisoitua versiota.

Suhteellinen koheesiomitta

Laatumitan lisäksi Martin on esitellyt myös alkuperäiseen mittapakettiin kuulumattoman komponenttien suhteellisen koheesiomitan H [18]:

$$H = \frac{\rho+1}{N_c} \ . \quad (5)$$

Kaavassa ρ on komponentin luokkien välisten riippuvuuksien lukumäärä ja N_c komponentin luokkien määrä. Osoittajan ylimääräisen yhteenlaskun tarkoituksena

Taulukko 1: Martinin [18] määrittelemät komponenttimitat.

Tunniste	Nimi
C_a	Saapuvat kytkökset
C_e	Lähtevät kytkökset
I	Epävakaus
N_a	Abstraktien luokkien lukumäärä
N_c	Luokkien lukumäärä
A	Abstraktisuus
D	Etäisyys pääsarjasta
D'	Normalisoitu etäisyys
H	Suhteellinen koheesio

on estää H:n arvo 0 kun komponentissa on vain yksi luokka.

Koheesiomitassa on kuitenkin muutama huomattava ongelma. Martin ei ohjeistanut luokkien riippuvuuksien laskemisessa. Epäselväksi jää niin kytkeytymisehto – milloin luokat lasketaan kytkeytyneeksi toisiinsa – kuin kytkösten suunnan huomioiminen. Esimerkiksi jos komponentin molemmat luokat käyttävät toisiaan, on epäselvää onko riippuvuuksien lukumäärä tällöin 1 vai 2.

Mitan teoreettisessa kelpuuttamisessa huomasimme ongelmia myös sen määrittelyssä ja esittelimme mitasta korjatun, teoreettisesti kelvollisen version:

$$H' = \begin{cases} 1 & \text{kun} \quad N_c = 1 \\ \frac{2\rho}{N_c(N_c-1)} & \quad\quad N_c > 1 \end{cases} \tag{6}$$

Uudelleen määriteltynä koheesiomitta on lähes identtinen Briandin ym. [13] määrittelemän luokan koheesiomitta Co':n kanssa. Mielenkiintoisesti myös Co' esiteltiin alun perin korjaukseksi Co:n teoreettisiin ongelmiin. Teoreettinen kelpuuttaminen ja H':n perustelut on esitetty kokonaisuudessaan lähteessä [16].

Martinin esittelemät komponenttimitat on koottu taulukkoon 1.

4 Komponentin ominaisuudet

Ohjelmistomittojen on tarkoitus luokitella ohjelmamoduulit paremmuusjärjestykseen yhdistämällä yksi tai useampi ominaisuus numeroarvoihin tai symboleihin yksiselitteisten sääntöjen avulla [2]. Mitattavat ominaisuudet voivat olla konkreettisia suoraan arvioitavia suureita kuten ohjelman koko. Ne voivat olla myös abstrakteja, joiden mittaamisesta ei välttämättä ole yhtenäistä käsitystä.

Ohjelmakomponentille, kuten mille tahansa ohjelman osalle, voidaan määritellä lukuisia erilaisia ominaisuuksia. Kuten luokkatasolla on osoitettu [27], vain osa näistä ominaisuuksista vaikuttaa kiistattomasti ohjelman ulkoiseen laatuun.

4.1 Informaatiotulva

Nykypäivän suuret ohjelmistot koostuvat miljoonista riveistä ja tuhansien ihmisten työmäärästä. Esimerkiksi, aikoinaan maailman suurimmaksi ohjelmaksi arvioidussa, Debian 4.0 -käyttöjärjestelmässä on 10 106 komponenttia, jotka koostuvat yhteensä 288,5 miljoonasta fyysisestä lähdekoodirivistä [28]. Informaatiotulva niin ihmiselle kuin metriikoille on valtaisa, ja järjestelmää kokonaisuudessa arvioivien mittojen näkemiä yksityiskohtia tulisikin karsia [19, 29]. Informaation rajoittamista

voidaan lähestyä esimerkiksi graafimallin avulla.

Ohjelmistoa voidaan mallintaa suunnattuna graafina, jossa soveltuvat elementit toimivat solmuina ja kaaret ovat riippuvuuksia näiden välillä [30]. Järjestelmätasolla solmuja olisivat ohjelmakomponentit [31], komponenttikerroksella solmuina toimisivat luokat ja luokkatasolla metodit sekä muuttujat. Valitsemalla graafin osat näin, yksityiskohtien määrä vähenee ja kokonaiskuva kasvaa siirryttäessä kerrokselta toiselle.

Suurin osa menetetyistä yksityiskohdista on arkkitehtuuritason tarkastelussa ylimääräistä tietoa. Esimerkiksi komponenttitasolta löytyy runsaasti metodeita ja muuttujia, mutta useiden luokkien välisiä suhteita tarkasteltaessa ne kertovat vain, onko metodilla tai muuttujalla jokin suhde toiseen luokkaan. Sama tieto saadaan tarkastelemalla, onko kahden luokan välillä riippuvuussuhteita.

Toinen saavutettava hyöty on vastaavuus UML:n luokka- ja komponenttikaavioiden kanssa. Abstraktiotasojen yhteneväisyyden perusteella voidaan ohjelmistomittojen laskenta automatisoida suunnittelutyökaluihin, ja ensimmäiset karkeat tulokset saadaan arvioitavaksi jo korkean tason suunnittelussa.

Kuvatun kaltaisen elementtien valinnan ongelmana on informaation karsimisesta johtuva tarkkuuden menetys. Esimerkiksi komponenttien välisiä riippuvuuksia voidaan painottamattomilla graafeilla ilmaista vain binäärisesti. Jos suhteiden vahvuuksien mallintaminen on sovellusalueella tarpeen, voidaan graafin kaarille lisätä painot. Tällöin kaarien välisinä painoina olisivat esimerkiksi yksittäisten metodikutsujen tai viittauksien summat. Eri relaatiotyypeille, kuten perintä- ja viittaussuhteille voitaisiin määritellä omat painofunktiot tai käyttää erikoistettua graafia.

4.2 Perinteisiä ominaisuuksia

Kiinnevoima, kytkeytyminen, koko ja kompleksisuus ovat klassisia ominaisuuksia, joita on arvioitu ohjelmista useilla eri abstraktiotasoilla. Näiden lisäksi olioparadigmalle klassinen mitattava ominaisuus on perintä. Esimerkiksi Chidamberin ja Kemererin mittapaketti keskittyi juuri näihin ominaisuuksiin.

Seuraavassa arvioidaan näiden ominaisuuksien soveltuvuutta arkkitehtuurikomponenteille. On kuitenkin huomattava, ettei ohjelmakomponentti ole luokan yleistys, jolle kaikki alemman tason ominaisuudet soveltuisivat. Tämän vuoksi kaikkia mitattavia ominaisuuksia on lähestyttävä kriittisen tarkastelun kautta.

Koheesio

Coad ja Yourdon [32] määrittelivät moduulin kiinnevoiman eli koheesion asteeksi, kuinka hyvin sen osat toteuttava yhden tarkoin määritellyn toiminnallisuuden. Kiinnevoiman voi myös mieltää tarkoittavan kuinka hyvin moduuli toteuttaa semanttisesti mielekkään käsitteen [33], sillä moduulin elementtien ei välttämättä tarvitse olla tiukasti toisiinsa sidottuja kiinnevoimaisissa luokissa. Riittää, että moduulin osien tuottamat palvelut ovat yhtenäisiä.

Tässä työssä ohjelmakomponentit määriteltiin aiemmin joukoksi luokkia, joilla on syy sijaita yhdessä. Tämän vuoksi koheesiota voidaan pitää arkkitehtuurikomponenteille mielekkäänä mitattavana ominaisuutena – siitä huolimatta ettei koheesio ole menestynyt erityisen hyvin luokkatason tutkimuksissa [27, 34].

Nykykäsityksen mukaisesti koheesiota voidaan tarkastella eri tavoilla: moduulin sisältä tai ulkoa käsin [35]. *Sisäiseen näkymään* pohjautuvat koheesiomitat tarkastelevat kohdetta puhtaasti ilman tietoa sen

ympäristöstä. Esimerkiksi komponenttien koheesiomitta *H* on tällainen.

Toinen tapa koheesion arvioimiseen on huomioida moduulin konteksti ja sen asiakkaat. Asiakkaan moduulista käyttämät piirteet muodostavat yksittäisen *asiakasnäkymän* ja jokaisella asiakkaalla on oma näkymänsä luokasta. Jos asiakasnäkymät eroavat toisistaan, moduuli saattaa palvella useammassa kuin yhdessä roolissa. Esimerkiksi Mäkelän ja Leppäsen [35] määrittelemä koheesion puute asiakkaissa (LCIC, engl. *Lack of Coherence in Clients*) on asiakaspohjaiseen näkymään perustuva koheesiomitta.

Asiakasnäkymiin pohjautuvia koheesiomittoja on ehdotettu käyttäväksi perinteisen sisäistä eheyttä tarkastelevien mittojen tukena, sillä yhteen näkökulmaan perustuva mitta ei pysty antamaan täyttä kuvaa komponentista [35]. Luokkatasolla ei myöskään ole laajasti tutkittu asiakasnäkymään perustuvien koheesiomittojen toimivuutta. Tämän vuoksi ohjelmakomponenteille olisi hyvä määritellä sekä sisäinen että asiakaspohjainen koheesiomitta, mutta myös suhtautua huolella niiden empiiriseen kelpuuttamiseen.

Koko

Moduulin koko on yksi klassisimmista mitattavista ominaisuuksista. Melton ja Tempero [20] listasivatkin koon hallitsemisen yhdeksi modularisoinnin rajoitteista, sillä komponentin pitäisi olla sitä rakentavan työryhmän hallittavissa. Samalla toinen suunnitteluperiaate, kytkeytymisen minimoiminen, ajaa luokkia yhdeksi suureksi komponentiksi [36].

Koko on selkeästi yksi keskeisistä mitattavista ominaisuuksista myös komponenttitasolla ja sitä voidaan havainnollistaa esimerkiksi lähdekoodirivien, metodien tai luokkien lukumäärällä. Luokkien piirteisiin perustuvat mitat kertovat enemmän luokista kuin komponentista, eikä esimerkiksi komponenttien koon vertailu metodien määrällä ole mielekästä.

Luokkien määrän mittauksessa voidaan ottaa huomioon kaikki [18, 22] tai vain julkiset luokat [37]. Luokkien roolien huomioiminen on koon laskennassa keskeistä, sillä esimerkiksi Javan käyttöliittymäkomponenteissa hyödynnetään paljon nimettömiä sisäluokkia. Sisäluokkien rajaaminen pois antaa paremman yleiskuvan komponentista, mutta voi vääristää komponentin rakentamiseen tarvittavan työmäärän arvioimista. Toisaalta komponentin koon laskennassa luokkien roolijako voidaan tehdä hyvinkin hienovaraiseksi, ja valita soveltuvat roolit käytettäväksi tarpeen tai tilanteen mukaan.

Kompleksisuus

Kompleksisuus tarkoittaa moduulin, tässä tapauksessa ohjelmakomponentin, ymmärtämisen vaikeutta. Kompleksisuus on abstrakti ominaisuus, jossa osatekijöinä on useita muita ominaisuuksia. Komponenttitasolla tarkastellaan luokkia ja niiden välisiä suhteita, jolloin komponentin monimutkaisuus muodostuu luokkien keskinäisistä suhteista. Mittojen tulisi arvioida alkuperäistä ominaisuutta, ei sen heijastetta [2], jolloin olisi parempi mitata esimerkiksi komponentin koheesiota ja kytkeytymistä kuin sen kompleksisuutta.

Perinteiset kompleksisuusmitat ovat myös usein epäonnistuneet virheiden tai muutosten ennustajina [5], mikä lisää perusteita keskittyä monimutkaisuuden aiheuttajien mittaamiseen seurausten sijasta. Yleinen kompleksisuusmitta kykenee ehkä osoittamaan monimutkaiset komponentit, mutta ei erottelemaan syitä. Esimerkiksi koheesio- ja kytkeytymismitoilla voidaan tunnistaa huonosti modularisoidut arkkitehtuurin osat, antimalleja etsimällä yleisesti tunnetut suunnitteluongel-

mat ja kokomitoilla hallitsemattoman suureksi paisuvat komponentit.

Kytkeytyminen

Kytkeytyminen on koheesion ohessa tärkeä ohjelmistosuunnittelun vuorovaikutuksen ohjenuora. Kun koheesio tarkastelee luokan sisäistä yhtenäisyyttä ja yhteistoimintaa, kytkeytyminen keskittyy luokkien ulkoisiin suhteisiin. Coadin ja Yourdonin klassisen määritelmän mukaan kytkeytyminen on *"olioperustaisen suunnittelun osien välinen yhteen kytkeytyneisyys"* [32, s. 129]. He korostavat, että kytkeytymiseen vaikuttaa kytkösten määrien lisäksi niiden kompleksisuudet.

Kytkeytyminen on tärkeimpiä mitattavista komponentin ominaisuuksista, sillä kytkeytyminen vaikuttaa osaltaan niin uudelleenkäyttävyyteen, ymmärrettävyyteen kuin ylläpidettävyyteenkin [38].

Kuten koheesiota, myös kytkeytymistä voidaan tarkastella eri näkökulmista. Briand, Daly ja Wüst [38] antoivat määrittelykehyksen, jossa he esittelivät useita erilaisia lähtökohtia kytkeytymismittojen esittelyyn. Huomionarvoinen suunnittelulähtökohta on kytkösten suunta. Virheet moduulissa, jota käytetään paljon saattavat olla hyvinkin ongelmallisia. Toisaalta moduuli, joka käyttää paljon palveluita saattaa olla vaikeasti siirrettävissä. Tämän vuoksi Briand ym. suosittelivat tarkastelun jakamista kahteen eri osaan kytkeytymissuunnan mukaan.

Perintä

Luokkajoukoille perintä ei varsinaisesti tarkoita mitään, sillä yksittäisen luokan ulkopuolelle perintä näkyy vain tavallista voimakkaampana kytkeytymisenä. Siitä huolimatta Ducasse ym. [22] jakoivat komponenttitasolla perinnän tarkastelun kolmeen osaan: komponentin sisäiseen hierarkiaan, perimiseen komponentin ulkopuolisesta yliluokasta ja periytymiseen komponentin ulkopuoliselle lapselle.

Empiirisissä tutkimuksissa perintämetriikat ovat kuitenkin menestyneet huonosti alemmalla tasolla, ja niiden uskotaan heijastavan enemmän suunnittelijoiden olioymmärrystä kuin virhealttiutta [27]. Samoin on epäselvää mitä komponentin perintärelaatioiden määrä kertoo komponentin laadusta tai sen ongelmista. Perintämetriikoille ei näyttäisi löytyvän tarkoitusta tai käyttöä komponenttitasolla.

4.3 Komponenttien arvoja

Matalammilla tasoilla erityisesti kytkeytymismitat ovat suoriutuneet kohtuullisesti virheiden osoittajina [27, 34], mutta toiset ominaisuudet saattaisivat toimia niitä paremmin komponenttitasolla.

Seuraavassa on tarkasteltu erilaisia ominaisuuksia, joita on luokkajoukoilta yritetty mitata, sekä ominaisuuksia, jotka ovat esiintyneet arkkitehtuurikomponentin määritelmissä.

Abstraktisuus

Abstraktisuus tarkoittaa tässä yhteydessä komponentin abstraktien osien osuutta kokonaismäärästä. Pelkästään komponentin abstraktiusasteen perusteella on vaikea tehdä päätelmiä sen laadukkuudesta tai virhealttiudesta. Ainoastaan muutamat komponentit asettuvat mitan ääripäihin, joko puhtaan abstrakteiksi tai konkreettisiksi. Mitan väliarvoille tulkinta on monimutkaisempi: edustaako 30 % abstraktiusaste hyvää laatua vai virhealttiutta? Arvon perusteella voidaan ainoastaan päätellä konkreettisia luokkia olevan enemmän kuin abstrakteja. Tosin tuloksen tulkintaan vaikuttavat myös kokonaismäärän laskennassa käytetyt luokkaroolit. Esimer-

kiksi sisäluokkien huomioiminen pienentäisi joidenkin komponenttien abstraktiutta.

Abstraktiuden käyttämistä suorana mittana on vaikea perustella, sillä todennäköisesti kyseisellä ominaisuudella ei ole vaikutusta komponentin ulkoiseen laatuun. Ominaisuudella saattaisi olla käyttöä johdetuissa mitoissa, mutta ei puhtaana laadun arvioijana.

Kapselointi ja informaation kätkeminen

Informaation piilottaminen ja kapselointi ovat modularisoinnin perusperiaatteita. Tarkoituksena on paljastaa ainoastaan palveluiden vaatima tieto, mutta salata toteutuksen yksityiskohdat [39]. Komponenteille tämä tarkoittaa julkista rajapintaa, mutta yksityisiä palveluita toteuttavia luokkia. Tiedon kätkemisellä pyritään estämään sekä komponentin väärinkäyttö että riippuvuus konkreettisesta toteutuksesta. Ominaisuudet ovat tärkeässä roolissa myös esimerkiksi muutosten leviämisen estämisessä ja ns. *särkyvän yliluokan ongelmassa* (ks. [40]). Tämän vuoksi huonosti kapseloidut komponentit olisi hyvä pystyä tunnistamaan ajoissa.

Modulaarisuus

Modulaarisuus kuvaa kuinka hyvin ohjelma on jaettu itsenäisiin ja yhtenäisiin komponentteihin [36]. Ominaisuus on yksi arkkitehtuurisuunnittelun tärkeimmistä tavoitteista [41]. Siksi modulaarisuudelle on kehitetty myös omia mittoja. Tosin muodostettujen komponenttien itsenäisyyttä voidaan arvioida kytkeytymisellä, yhtenäisyyttä koheesiolla ja hallita kokoa kokomitoilla. Ominaisuuden arvioiminen erillisellä mitalla on tuskin tarpeellista, sillä jokaista sen seurausta pystytään mittaamaan yksittäisillä mitoilla.

Muutoksen propagoituminen

Muutoksen propagoituminen on todennäköisyys, jolla muutos tai virhe komponentissa vaikuttaa toiseen komponenttiin. Propagoituminen on järjestelmän modularisoinnin epätoivottu ominaisuus, jota pyritään estämään esimerkiksi matalalla kytkeytymisellä sekä kapseloinnilla [36]. Virheen leviämisen todennäköisyys on riippuvainen komponentin muista ominaisuuksista, ja on tehokkaampaa keskittyä näihin ominaisuuksiin kuin uuden johdetun mitan kehittämiseen.

Riskialttius

Yacoub ym. [25] johtivat riskialttiudelle mitan kompleksisuuden ja kytkeytymisen pohjalta. Ominaisuuden erillinen arviointi on kuitenkin ylimääräistä työtä, sillä komponentit voidaan priorisoida testattavaksi esimerkiksi kytkeytymisen ja koheesion avulla, ilman erillisen johdetun mitan määrittelemistä. Esimerkiksi riskialttiutta voidaan arvioida käyttämällä kytkeytymis-, koheesio- tai jopa kokomittoja.

Vakaus

Vakaus tarkoittaa tilan muuttamiseen vaadittavaa työmäärää. Vakaiden komponenttien tunnistaminen on keskeistä arkkitehtuurisuunnittelussa, sillä vakaan riippuvuuden periaatteen mukaan komponenttien pitäisi olla kytkeytyneinä ainoastaan vakaisiin komponentteihin.

Suunnitteluperiaatteiden mukaan luodut vakaat komponentit ovat myös uudelleenkäytettäviä, sillä niillä on vain vähän ulkoisia riippuvuuksia. Lisäksi vakaiden komponenttien pitäisi asiakasmäärän perusteella olla arkkitehtuurissa keskeisiä, jolloin ne tulisi priorisoida testauksessa korkealle.

Toisaalta yksin vakauden perusteella voi olla hankala asettaa komponentteja

laatujärjestykseen, minkä vuoksi ominaisuuden käyttämistä suorana laatuindikaattorina on tutkittava empiirisesti vielä lisää.

Uudelleenkäytettävyys

Uudelleenkäytettävyys on yksi komponenttijaon perusteluista, sillä komponentin pitäisi olla itsenäinen palveluyksikkö [40]. Ominaisuutta voitaisiin käyttää esimerkiksi testauksen ja kehityksen priorisoimisessa sellaisiin komponentteihin, joiden tiedetään soveltuvan uudelleenkäytettäväksi. Kierrätettyihin komponentteihin myöhemmin tehtävien muutosten kertautuva työmäärä saattaa olla huomattava.

Uudelleenkäytettävyys on kuitenkin hyvin abstrakti ominaisuus, jonka arvioiminen staattisella analyysillä on vaikeaa, ellei mahdotonta. Uudelleenkäyttökerrat on helppo mitata, jos komponentin historiatiedot ovat käytettävissä. Niiden ominaisuuksien, jotka tekevät komponentista uudelleenkäytettävän, havainnollistaminen on huomattavasti monimutkaisempaa.

4.4 Mitattavien ominaisuuksien joukko

Useat arvioiduista komponentin ominaisuuksista ovat äärimmäisen abstrakteja käsitteitä, ja niitä voitaisiin mitata yksinkertaisemmin ja tehokkaammin muiden ominaisuuksien mittareilla. Esimerkiksi komponentin kompleksisuus muodostuu sen jäsenluokkien välisistä suhteista, komponentin koosta ja sen ulkoisista suhteista. Näitä voidaan arvioida erikseen helpommin komponentin koheesio-, koko- ja kytkeytymismitoilla kuin yhdellä monimutkaisella mitalla.

Käytettäväksi ehdotettavien ohjelmistomittojen määrää tulisi myös rajata, sillä todennäköisesti suuresta joukosta osa korreloi keskenään ja paljastaa samanlaisia virheitä. Vaikka mittaohjelmat laskevat kymmenien mittojen arvoja hetkessä, pitkien tuloslistausten tulkitseminen on työlästä ja hankalaa. Käyttäjän pitäisi myös ymmärtää jokaisen mitan tarkoitus, sen mahdolliset ongelmat ja menetelmät huonon arvon korjaamiseksi. Huomattavasti käyttäjäystävällisempää on kehittää pieni, tehokas ja kattava joukko komponenttimittoja, joita on helppo ymmärtää ja soveltaa.

Taulukossa 2 on esitetty tarkastellut komponenttien ominaisuudet. Komponenttien keskeisimmiksi piirteiksi on tunnistettu *kapselointi*, *koheesio*, *koko*, *kytkeytyminen* ja *vakaus*. Tarkastelu rajattiin vain suoriin mittoihin, joten johdetuissa mitoissa käytettyjä ominaisuuksia ei ole käsitelty.

Kirjallisuudessa keskustelluista komponenttimitoista erityisesti Martinin esittelemät mitat kattavat useimmat edellä mainituista ominaisuuksista. Näitä mittoja voitaisiin käyttää perustana komponenttimittapaketin määrittelemiseksi.

Huomattavaa on myös, ettei kapselоinnille ole määritelty mittaria. Kapselointia ja informaation piilottamista voidaan lähestyä arvioimalla yksityisten osien suhdetta julkisiin [42], mutta tällöin ohjelmaa selkeyttävä kompleksisten funktioiden työmäärän jakaminen useille pienemmille piilotetuille alifunktioille muuttaisi mitan arvoa [43]. Toinen vaihtoehto olisi tarkastella abstrakteille rajapinnoille saapuvien riippuvuuksien suhdetta kaikkiin riippuvuuksiin.

Rajapintojen käyttöasteen huono arvo ei kuitenkaan sovellu suoraan ulkoisten laatutekijöiden mittariksi, sillä kyseessä on enemmänkin hyvä ohjelmointitapa kuin selkeästi laatua heikentävä virhe. Sisäisiin laatutekijöihin ominaisuus kuitenkin vaikuttaa. Näiden perusteella voidaan olettaa huonon arvon saaneiden komponenttien olevan virhealttiimpia kuin muut, sillä vaikeasti ylläpidettäviin ja vaikeasti ymmärrettäviin komponentteihin oletetta-

Taulukko 2: Komponenteille tunnistetut ominaisuudet sekä niiden arvioimiseen mahdollisesti käytettävät mitat. Taulukossa on annettu myös ominaisuuksille, joiden suora mittaaminen on hankalaa, vaihtoehtoisia ominaisuuksia.

Ominaisuus	**Mahdollisia mittoja**
Abstraktius	A
Kapselointi	–
Koheesio	
Sisäinen	H, H'
Asiakaspohjainen	CPC [19], SCC [21]
Koko	N_c
Kompleksisuus	Koheesio-, koko- ja kytkeytymismitat
Kytkeytyminen	
Tulevat	C_a
Lähtevät	C_e
Modulaarisuus	Koheesio-, kytkeytymis- ja kokomitat
Muutoksen propagoituminen	Kytkeytymis- ja kapselointimitat
Perintä	Kytkeytymismitat.
Riskialttius	Kytkeytymis-, koheesio- ja kokomitat
Vakaus	I
Uudelleenkäytettävyys	Koheesio, kytkeytymis- ja vakausmitat

vasti kertyy enemmän virheitä kuin muihin. Oletus pitäisi kuitenkin varmistaa empiirisillä mittauksilla ennen mitan hyväksymistä.

Kapselointi ja informaation piilottaminen ovat laajoja käsiteitä, joista rajapintojen käyttöaste edustaa vain yhtä kapeaa osa-aluetta. Ominaisuuksia voitaisiin myös tarkastella muista näkökulmista, esimerkiksi miten perijät käyttävät esivanhempiensa attribuutteja tai miten komponentin julkinen rajapinta on määritelty.

5 Martinin metriikan empiirinen testaaminen

Tarkastelluista ominaisuuksista huomattiin Martinin metriikan kattavan suurimman osan mahdollisesti mielenkiintoisista mitattavista piirteistä. Mittoja ei kuitenkaan ole kelpuutettu teoreettisesti tai empiirisesti aiemmin, joten sen käyttökelpoisuus mittapaketin perustana on epävarmaa. Tässä työssä keskitytään arvioimaan vain metriikan soveltuvuutta laatuindikaattoriksi.

Martinin metriikan empiirinen kelpuuttaminen on huomattavan haasteellista, sillä vaadittava ulkoisen laadun arviointi ei tässä tapauksessa ole yksiselitteistä. Komponenttien laadun ja riippuvuuksien parantaminen liittyy ohjelman ylläpidettävyyteen. Ylläpidettävyyttä voisi pyrkiä arvioimaan komponenttien muutoksen määrällä kuten Champagne, Malton ja Dong [44], mutta metriikka pyrkii ennustamaan muutoksen työmäärää eikä sen toistuvuutta. Myös virheiden määrän käyttäminen kelpuuttamiseen on ongelmallista, sillä todennäköisesti suurempi osa virheistä syntyy ohjelmointi- ja luokkasuunnittelun virheistä kuin erheellisestä komponenttijaosta.

Selkein vaihtoehto metriikan kelpuuttamiseen olisi mitata komponentin ylläpitämisen hintaa, mutta käytännössä tällaisia mittaustuloksia on harvoissa ohjelmistoprojekteissa. Tässä työssä yritetään toisen-

laista vaihtoehtoa tutkimalla voidaanko Martinin metriikan avulla tunnistaa suunnitteluvirheitä tunnetuista avoimen lähdekoodin ohjelmistoista.

5.1 Testit kirjallisuudessa

Martinin mittaa ovat käyttäneet useat tutkijat (mm. [45, 46, 47, 48, 49]), mutta ainoastaan Champaign ym. [44] ovat yrittäneet kelpuuttaa osaa mittapaketista. He vertasivat Martinin epävakauden mitan I arvoja Linuxin ytimen muutoshistoriaan. Muutosherkkyyttä he arvioivat suhteuttamalla muuttuneiden julkaisujen määrän kaikkiin julkaisuihin, joissa komponentti oli mukana.

Tutkimuksessa komponentti luokiteltiin muuttuneeksi, jos sen koko oli muuttunut edellisestä julkaisuversiosta. Valitulle tekniikalle sekä suuri refaktorointi että yksittäisen kirjoitusvirheen oikaiseminen ovat samanarvoisia muutoksia. Champaign ym. eivät suorittaneet tilastollista analyysiä riippuvuuksista, mutta esitettyjen kuvaajien perusteella I:n ja muutosherkkyyden välillä ei ollut yhteyttä.

Martin määritteli vakauden muutoksen tekemisen vaikeudeksi. Hän painottaa, ettei komponentin vakaus liity muutosten määriin tai toistuvuuteen. Selvästi Champaign ym. eivät mitanneet muutoksen vaativuutta tai työmäärää, jos yksittäinen muutos oli mikä tahansa komponentin koon muutos.

5.2 Järjestelyt

Mittasimme viittä ohjelmaa Martin metriikalla: *Vuze*, *Tomcat*, *ArgoUML*, *Xerces* ja *jEdit*, jotka ovat pitkään kehitettyjä, vakaina ja hyvinä pidettyjä Javalla toteutettuja avoimen lähdekoodin ohjelmia. Ohjelmista tarkasteltavien osien joukko rajattiin vain projektien itse tuottamiin komponentteihin, sillä nämä edustavat varmasti samanlaisia suunnitteluperiaatteita ja käytäntöjä.

Kirjastokomponentin vakauden arvioiminen on hankalaa, sillä mitattava järjestelmä saattaa antaa siitä todellisuudesta poikkeavan kuvan. Esimerkiksi jos järjestelmä käyttää vain pientä joukkoa mahdollisista palveluista, on tulevien riippuvuuksien määrä huomattavasti pienempi kuin toisessa ympäristössä mitattuna. Kirjastokomponentteja pitäisi arvioida erikseen riittävällä määrällä asiakkaita.

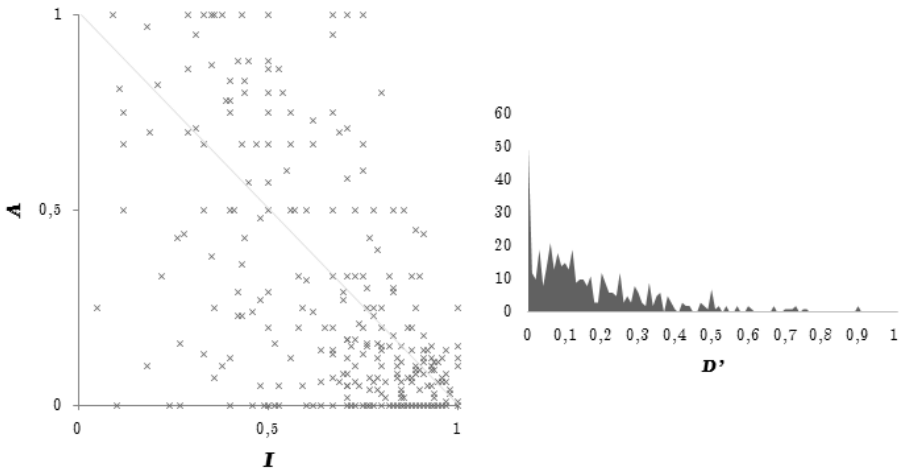
Kuvassa 2 on esitetty kaikkien mitattujen komponenttien IA-kuvaaja sekä komponenttien D':n arvojen histogrammi. Jälkimmäisestä huomataan, että suurin osa komponenteista on varsin lähellä pääsarjaa: 75 % paketeista on korkeintaan 0.25 etäisyyden päässä. Taulukossa 3 on esitetty tilastollisia tunnuslukuja mittauksista.

5.3 Huomioita komponenteista

Mittauksen yhteydessä kiinnitettiin erityisesti huomioita paketteihin, jotka olivat kaukana pääsarjasta. Tavoitteena oli tutkia sisältyykö näihin komponentteihin selkeästi tunnistettavia suunnittelupuutteita tai helposti tehtäviä parannuksia.

ArgoUML:n paketin `org.argouml.i18n` D':n arvo on 0.9. Komponentissa on vain `Translator`-luokka, joka vastaa järjestelmän lokalisoinnista. Tästä luokasta ovat riippuvia 495 luokkaa 43 paketista. Tällaisen komponentin palvelut olisi voinut piilottaa rajapinnan taakse, siitä huolimatta, että rajapinnalla olisi vain yksi toteuttaja. Nyt yhden luokan muuttaminen vaikuttaisi puolessa järjestelmän komponenteista.

Vastaavasti myös Tomcatin lokalisoinnin toteuttavan paketin `org.apache.tomcat.util.res` palvelut olisi hyvä piilottaa rajapinnan taakse. Paketissa on vain `StringManager`-luokka ja tästä luokasta ovat suoraan riippuvaisia kahdeksan eri pakettia. Komponentin D' arvo on 0.73. Paketissa huomattavaa on se, että kaik-



Kuva 2: Kaikkien 430 arvioidun komponentin *IA*-kuvaaja ja D':n histogrammi, josta nähdään suurimman osan komponenteista olevan varsin lähellä pääsarjaa.

ki sen käyttämät palvelut ovat osa Javan luokkakirjastoa. Jos ne luokiteltaisiin vakaiksi palveluiksi ja jätettäisiin tarkastelun ulkopuolelle, paketin etäisyys pääsarjasta olisi 1.

Vuzen komponentilla `com.aelitis.azureus.login` on ainoastaan yksi konkreetti luokka ja vain yksi asiakas. D':n arvo paketille on korkea, sillä se käyttää yhteensä yhdeksän paketin palveluita. Kaikki käytetyt palvelut ovat osa Javan luokkakirjastoa. Luokkakirjaston palveluita voidaan pitää staattisina, sillä ne eivät ole sellaisia muutoksen syitä, mitä Martin kuvasi. Jos luokkakirjasto jätettäisiin huomiotta, paketin etäisyys pääsarjasta olisi 0.

ArgoUML:n paketin `org.argouml.application.helpers` etäisyys pääsarjasta on 0.76. Se muodostuu kolmesta konkreettisesta luokasta, jotka tarjoavat järjestelmälle sekalaisia palveluja. Tarkasteltavat luokat eivät muodosta yhtenäistä kokonaisuutta toiminnallisuudeltaan. Luokka `ApplicationVersion` tarjoaa ohjelman versionumeron ja osoitteen kotisivuille. Luokat `ResourceLoader` ja `ResourceLoaderWrapper` ylläpitävät listaa ohjelman käytettävissä olevista ulkoisista resursseista. Komponentin palveluista ovat riippuvaisia 119 luokkaa, ja selvästi komponentin voisi hajottaa kahteen osaan tai poistaa se sijoittamalla luokat toisiin komponentteihin.

Komponentti `org.apache.xerces.impl.xs.util` koostuu 14 konkreetista luokasta, jotka toteuttavat erilaisia tietovarastoja. Järjestelmässä 36 luokkaa on riippuvaisia komponentin palveluista. Selvästi myös nämä palvelut olisi voinut piilottaa rajapinnan taakse.

Paketin `org.gtj.sp.jedit.msg` normalisoitu etäisyys on 0.72, ja se koostuu komponenttien keskinäisessä keskustelussa käytettävistä viesteistä. Paketti olisi perusteltua määritellä kokonaan rajapinnoista koostuvaksi, sillä nykyinen toteutus muodostaa syklin `org.gtj.sp.jedit.gui`:n kanssa. Riippuvuus konkreetteihin luokkiin voitaisiin poistaa *Abstraktin tehtaan* [50] avulla. Vaikka pa-

Taulukko 3: Normalisoidun etäisyyden tilastollisia tunnuslukuja tarkasteltavista ohjelmista.

	Keskiarvo	**Mediaani**	**min**	**Q1**	**Q3**	**max**	**N**
Vuze 4.2.0.0	0.15	0.10	0	0.05	0.21	0.90	208
Tomcat 6.0.18	0.18	0.14	0	0.01	0.27	0.73	90
ArgoUML 0.28	0.20	0.16	0	0.07	0.28	0.90	80
Xerces2 J 2.9.1	0.22	0.15	0	0.05	0.32	0.75	36
jEdit 4.2	0.18	0.11	0	0.08	0.20	0.73	16
Kaikki	0.17	0.12	0	0.05	0.25	0.90	430

ketti on riippuvainen vain kahdesta komponentista, on se todennäköisesti herkkä muutoksille. Käyttöliittymän ja toiminnallisuuden kehittyessä tarve uudenlaisten viestien määrittelemiseksi kasvaa.

Samoin pakettia `org.gtj.sp.util`, jonka etäisyys pääsarjasta on 0.51, olisi perusteltua muuttaa abstraktimmaksi. Paketti koostuu luokista, joita lähes jokainen `org.gtj.sp.jedit`:in alipaketti käyttää. Silti vain kaksi luokkaa on abstrakteja. Muutokset yksittäisiin paketin luokkiin vaatisivat pahimmillaan koko järjestelmän refaktoroimista. Paketin konkreetit luokat myös käyttävät toisten osapuolten toimittamia kirjastoja, joiden vaihtuminen voisi pahimmillaan johtaa näistä paketista riippuvien luokkien refaktoroimiseen.

Havaintojen pohjalta voidaan väittää Martinin mitan toimivan ainakin jossain määrin arkkitehtuurikomponenttien laatuongelmien indikaattorina, sillä suuressa osassa tarkastelluista korkean D':n paketeissa oli huomautettavaa.

Empiirisestä kokeesta on huomattava myös, että järjestelmiä tuntematon tarkastelija pystyi kohtuullisen pienellä vaivalla tunnistamaan metriikan avulla refaktorointia tarvitsevia kohteita yli neljänsadan näytteen joukosta.

Samalla tosin huomattiin, että joukossa oli mukana sekä muutamia vääriä positiivisia – jolloin mittarin arvo on liian suuri – että vääriä negatiivisia tuloksia. Tässä esitetyissä tapauksissa väärän tuloksen aiheutti luokkakirjastojen tulkitseminen muutoksen aiheuttajiksi. Mittaa pitäisikin soveltaa niin, ettei vakaita palveluita huomioitaisi I:tä laskettaessa. Tosin vakaiden palvelun määrittely jää avoimeksi kysymykseksi: Javan luokkakirjastoja voidaan selvästi pitää muuttumattomina palveluina, mutta kaikkia kolmannen osapuolen tarjoamia kirjastoja ei. Vakaiden palveluiden huomioimisesta kytkeytymisen yhteydessä on keskusteltu jo Briandin ym. artikkelissa [38].

Tarkastelu on osin puutteellinen, sillä merkittäviä vääriä positiivisia tuloksia ei löydetty. Tällöin selkeästi refaktoroimista kaipaava komponentti on jäänyt tunnistamatta metriikan annettua sille liian hyvän arvosanan. Tällaisen laadulliseen analyysiin tarvitaan kuitenkin huomattavasti tarkemmin kontrolloitu näytejoukko kuin mitä tässä tutkimuksessa oli käytössä. Olisi myös mielenkiintoista tutkia suuren, mutta huonolaatuisen järjestelmän arkkitehtuuria Martinin metriikalla. Emme kuitenkaan tällaista järjestelmää tähän tutkimukseen löytäneet.

6 Yhteenveto

Tässä tutkimuksessa on tarkasteltu sekä arkkitehtuurikomponenttien ohjelmistomittoja että näiden komponenttien mahdollisesti mielekkäitä mitattavia ominaisuuksia. Esiteltyjä komponenttimittoja on vain muutamia verrattuna satoihin luokkamittoihin. Näidenkin määrittelyissä huo-

mattiin puutteellisuuksia.

Työssä pyrittiin myös tarkastelemaan millaisia ominaisuuksia arkkitehtuurikomponenteista voisi mitata. Arvioiduista ominaisuuksista myös pyrittiin valitsemaan minimalistinen joukko, jota voitaisiin käyttää komponenttimittapaketin kehityksen lähtökohtana.

Työssä tarkasteltiin myös voidaanko Martinin metriikka käyttää komponenttien sisäisen laadun puutteen indikaattorina. Tapaustutkimuksessa se onnistui, mutta metriikan täydellinen empiirinen kelpuuttaminen vaatii huomattavasti laajempaa tutkimusta.

Tässä työssä käytetty lähestymistapa komponentin ominaisuuksien tunnistamiseksi keskittyi luettelemaan hyviä ohjelmointitapoja. Toisenlainen tarkastelutapa on esimerkiksi suunnittelun ongelmien [18] tai Meyerin [39] viiden modulaarisuuden kriteerin mittaaminen. Huonoja suunnitteluvaihtoehtoja on kuitenkin huomattavasti enemmän kuin hyviä, ja esimerkiksi jäykkyyden (engl. *rigidity*) eli muutosten tekemisen vaikeuden [18], arvioiminen yksittäisellä mittarilla on hyvin hankalaa.

Toivottavien ominaisuuksien mittojen, kuten koheesion, huonojen arvojen ei ole osoitettu riippumattomasti vaikuttavan ulkoisten laatukriteereiden heikentymiseen. Ongelmaksi on arvioitu niin riippuvuutta kokoon [1], kuin käsitteiden huonoa ymmärrystä [34]. Tämän vuoksi suunnittelun ongelmien mittaaminen voisi olla vaihtoehto kehitettäessä ohjelmistomittoja, sillä huonon ohjelmakoodin refaktoroinnilla yritetään parantaa ainakin järjestelmän ylläpidettävyyttä ja ymmärrettävyyttä.

Kiitokset

Haluan kiittää Tietojenkäsittelytieteen seuraa saamastani lukuvuoden 2008–2009 Pro gradu -palkinnosta sekä diplomityöni ohjaajia, dosentti Ville Leppästä ja tohtorikoulutettava Sami Mäkelää – kuten myös seuran määräämiä tarkastajia – pitkähkön työn läpikahlaamisesta sekä tutkielmaan liittyvistä neuvoista.

Viitteet

1. K. El Emam, S. Benlarbi, N. Goel ja S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, 2001.
2. N. E. Fenton ja S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company, USA, 2nd edition, 1998.
3. C. Jones. *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley Longman Publishing Co., Inc., USA, 2000.
4. T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.
5. M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.
6. S. R. Chidamber ja C. F. Kemerer. Towards a metrics suite for object oriented design. Kirjassa *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, s. 197–211, USA, 1991. ACM.
7. S. R. Chidamber ja C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
8. V. R. Basili, L. C. Briand ja W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.
9. W. Li ja S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.
10. N. I. Churcher ja M. J. Shepperd. Comments on 'A metrics suite for object orien-

ted design'. *IEEE Transactions on Software Engineering*, 21(3):263–265, 1995.

11. T. Mayer ja T. Hall. A critical analysis of current OO design metrics. *Software Quality Control*, 8(2):97–110, 1999.
12. M. Hitz ja B. Montazeri. Chidamber and Kemerer's metrics suite: A measurement theory perspective. *IEEE Transactions on Software Engineering*, 22(4):267–271, 1996.
13. L. C. Briand, J. W. Daly ja J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.
14. L. C. Briand, S. Morasca ja V. R. Basili. Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, 22(1):68–86, 1996.
15. E. J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.
16. S. Hyrynsalmi ja V. Leppänen. A validation of Martin's metric. Kirjassa *Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering*, s. 87–101, Tampere, Suomi, 2009. Tampereen teknillinen yliopisto.
17. J. C. Cherniavsky ja C. H. Smith. On Weyuker's axioms for software complexity measures. *IEEE Transactions on Software Engineering*, 17(6):636–638, 1991.
18. R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, USA, 2002.
19. M. L. Ponisio. *Exploiting Client Usage to Manage Program Modularity*. Väitöskirja, University of Bern, Sveitsi, 2006.
20. H. Melton ja E. Tempero. The CRSS metric for package design quality. In *ACSC '07: Proceedings of the thirtieth Australasian conference on Computer science*, s. 201–210, Darlinghurst, Australia, 2007. Australian Computer Society, Inc.
21. T. Zhou, B. Xu, L. Shi, Y. Zhou ja L. Chen. Measuring package cohesion based on context. Kirjassa *WSCS '08: Proceedings of the IEEE International Workshop on Semantic Computing and Systems*, s. 127–132, USA, 2008. IEEE Computer Society.
22. S. Ducasse, M. Lanza ja L. Ponisio. Butterflies: A visual approach to characterize packages. Kirjassa *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium*, s. 7, USA, 2005. IEEE Computer Society.
23. M. Lindvall, R. Tesoriero ja P. Costa. Avoiding architectural degeneration: An evaluation process for software architecture. Kirjassa *METRICS '02: Proceedings of the 8th International Symposium on Software Metrics*, s. 77, USA, 2002. IEEE Computer Society.
24. P.-O. Bengtsson. Towards maintainability metrics on software architecture: An adaptation of object-oriented metrics. Kirjassa *NOSA'98: Proceedings of the First Nordic Workshop on Software Architecture*, s. 87–91, Ruotsi, 1998. University of Karlskrona/Ronneby.
25. S. M. Yacoub, H. H. Ammar ja T. Robinson. A methodology for architectural-level risk assessment using dynamic metrics. Kirjassa *ISSRE '00: Proceedings of the 11th International Symposium on Software Reliability Engineering*, s. 210–221, USA, 2000. IEEE Computer Society.
26. jDepend. `http://clarkware.com/software/JDepend.html`. Viitattu 17.10.2010.
27. L. C. Briand, J. Wüst ja H. Lounis. Replicated case studies for investigating quality factors in object-oriented designs. *Empirical Software Engineering*, 6(1):11–58, 2001.
28. J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor ja D. M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285. 2009.
29. W. Abdelmoez, D. M. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan, H. H. Ammar, Bo Yu ja A. Mili. Error propa-

gation in software architectures. Kirjassa *METRICS '04: Proceedings of the Software Metrics, 10th International Symposium*, s. 384–393, USA, 2004. IEEE Computer Society.

30. T. Mens ja M. Lanza. A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, 72(2):57–68, 2002.
31. V. Lakshmi Narasimhan ja B. Hendradjaya. Some theoretical considerations for a suite of metrics for the integration of software components. *Information Sciences*, 177(3):844–864, 2007.
32. P. Coad ja E. Yourdon. *Object-Oriented Design*. Prentice Hall, USA, 1991.
33. B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, USA, 1996.
34. L. C. Briand, J. Wüst, J. W. Daly ja V. Porter. A comprehensive empirical validation of design measures for object-oriented systems. Kirjassa *METRICS '98: Proceedings of the 5th International Symposium on Software Metrics*, s. 246–257, USA, 1998. IEEE Computer Society.
35. S. Mäkelä ja V. Leppänen. Client-based cohesion metrics for Java programs. *Science of Computer Programming*, 74(5-6):355–378, 2009.
36. T. B. Van Belle. *Modularity and the Evolution of Software Evolvability*. Väitöskirja, University of New Mexico, USA, 2004.
37. G. Baxter, M. Frean, J. Noble, M. Rickerbyand, H. Smith, M. Visser, H. Melton ja E. Tempero. Understanding the shape of Java software. *ACM SIGPLAN Notices*, 41(10):397–412, 2006.
38. L. C. Briand, J. W. Daly ja J. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.
39. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, USA, 2nd edition, 1997.
40. K. Koskimies ja T. Mikkonen. *Ohjelmistoarkkitehtuurit*. Talentum Media Oy, Jyväskylä, Suomi, 2005.
41. J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education, USA, 2000.
42. F. Brito e Abreu ja R. Carapuça. Object-oriented software engineering: Measuring and controlling the development process. Kirjassa *Proceedings of the 4th International Conference on Software Quality*, McLean, VA, USA, 1994.
43. T. Mayer ja T. Hall. Measuring OO systems: A critical analysis of the MOOD metrics. Kirjassa *TOOLS '99: Proceedings of Technology of Object-Oriented Languages and Systems*, s. 108–117, USA, 1999. IEEE Computer Society.
44. J. C. Champaign, A. Malton ja X. Dong. Stability and volatility in the Linux kernel. Kirjassa *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, s. 95–102, USA, 2003. IEEE Computer Society.
45. N. Ahmad ja P. A. Laplante. Employing expert opinion and software metrics for reasoning about software. Kirjassa *DASC '07: Proceedings of the Third IEEE International Symposium on Dependable, Autonomic and Secure Computing*, s. 119–124, USA, 2007. IEEE Computer Society.
46. R. M. Redin, M. F. S. Oliviera, L. B. Brisolara, J. C.B. Mattos, L. C. Lamb, F. R. Wagner ja L. Carro. On the use of software quality metrics to improve physical properties of embedded systems. Kirjassa *Distributed Embedded Systems: Design, Middleware and Resources*, s. 101–110. Springer Boston, 2008.
47. M. F. S. Oliveira, R. M. Redin, L. Carro, L. da Cunha Lamb ja F. R. Wagner. Software quality metrics and their impact on embedded software. Kirjassa *MOMPES '08: Proceedings of the 2008 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, s. 68–77, USA, 2008. IEEE Computer Society.
48. A. Capiluppi ja C. Boldyreff. Coupling patterns in the effective reuse of open source software. Kirjassa *FLOSS '07:*

Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development, USA, 2007. IEEE Computer Society.

49. M. Wermelinger, Y. Yu ja A. Lozano. Design principles in architectural evolution: A case study. Kirjassa *ICSM '08: IEEE International Conference on Software Maintenance*, s. 396–405, USA, 2008. IEEE Computer Society.
50. E. Gamma, R. Helm, R. Johnson ja J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing, USA, 1995.