Ohjelmistometriikat arkkitehtuuritasolla

Diplomityö
Turun yliopisto
Informaatioteknologian laitos
Ohjelmistotekniikka
Kesäkuu 2009
Sami Hyrynsalmi

Tarkastajat:
Ville Leppänen
Sami Mäkelä

TURUN YLIOPISTO
Informaatioteknologian laitos

SAMI HYRYNSALMI: Ohjelmistometriikat arkkitehtuuritasolla

Diplomityö, 185 s.
Ohjelmistotekniikka
Kesäkuu 2009

Ohjelmistometriikat ja -arkkitehtuurit ovat hyvin erilaisia työvälineitä ohjelmistojen laadun parantamiseksi. Ohjelmistojen mittaamisella etsitään ongelmallisia kohtia niin tuotteesta, projektista kuin prosessista. Mittojen avulla voidaan keskittää lisäresursseja ongelmallisiin kohtiin ja poistaa mahdolliset virheet ennen kun niistä tulee ongelmallisia häiriöitä ohjelman suoritukseen.

Ohjelmistoarkkitehtuurit ovat korkean tason suunnittelun apuväline. Arkkitehtuurin avulla valtavat järjestelmät voidaan pilkkoa pienempiin osiin, joiden toteuttaminen on huomattavasti alkuperäistä ongelmaa yksinkertaisempaa. Arkkitehtuuri koostuu tällaisista komponenteista, jotka ovat uudelleenkäytettäviä, itsenäisiä ja yhtenäisiä palveluita tarjoavia ohjelmistoyksiköitä.

Tässä opinnäytteessä tarkastellaan arkkitehtuuritasolle esiteltyjä ohjelmistomittoja. Arkkitehtuuritasolla perehdytään sekä yksittäiselle komponentille että kokonaiselle järjestelmälle määriteltyihin mittoihin. Työssä esitellään klassisia ohjelmistometriikoita sekä erityisesti olioparadigmalle kehitettyjä mittoja. Opinnäytteessä käsitellään myös keskeisimpiä ohjelmistoarkkitehtuurien menetelmiä ja arkkitehtuurityylejä.

Työssä tutkitaan komponenttien ja järjestelmien mitattavia ominaisuuksia. Arkkitehtuuritason ohjelmistomitoista keskitytään Robert C. Martinin (2002) *Agile Software Development: Principles, Patterns, and Practices* –kirjassa esiteltyyn metriikkaan. Martinin mitan teoreettista taustaa sekä mitan kelpuuttamista käsitellään opinnäytteen lopuksi.

Lukuun ottamatta muutamia puutteita koheesiokriteereissä, Martinin metriikka on teoreettisesti kelvollinen komponenttimitta. Opinnäytteessä esitellään myös uusi komponenttien koheesiomitta. Empiirisessä testissä havaittiin viiden avoimen lähdekoodin ohjelman noudattavan mitan periaatteita. Teoreettisen taustan ja empiiristen havaintojen pohjalta, Martinin metriikka voidaan pitää käyttökelpoisena komponenttimetriikkana.

Asiasanat: *ohjelmistometriikat*, *ohjelmistoarkkitehtuurit*, *ohjelmistokomponentit*, *komponenttimetriikat*, *Martinin metriikka*

UNIVERSITY OF TURKU
Department of Information Technology

SAMI HYRYNSALMI: Software Metrics for Architectural Levels

Master of Science in Technology Thesis, 185 p.
Software Engineering
June 2009

Software metrics and architectures are two very different kinds of tools for improving the quality of software. The measurement of software helps to identify the weak parts of a product, a resource or a process. Thus, it is possible to focus additional work on improving the quality of these pieces of software. The objective is to prevent possible faults leading into a failure in the software.

Software architectures are high-level tools for software design. The architectures enable one to divide a complex program into smaller components, which makes the implementation easier than working with the original program as a whole. The architecture consists of components i.e. software units which are reusable, independent and offer uniform services.

This master's thesis examines software metrics on the architectural level. The architectural level can be divided into two main categories, the component and system levels. The metrics for both sublevels are presented in the study. This thesis reviews the classic software metrics presented in the the 1960–80s and newer, object-oriented metrics. Some of the most central tools and architectural styles in software design are also presented.

This thesis discusses a list of measurable characteristics for the architectural levels. The Martin's metric, presented by Robert C. Martin in his book *Agile Software Development: Principles, Patterns, and Practices* (2002), is the most extensive component level measurement. The thesis also presents a theoretical validation and an empirical test of Martin's metric.

With a few exceptions in the theoretical test, Martin's metric worked as a theoretically valid software metric. The thesis presents an improved version of Martin's component cohesion metric. The results of the empirical tests show that five open source projects follow the principles of Martin's metric. The theoretical and empirical tests combined lead to the conclusion that Martin's metric is a useful, valid software metric.

Keywords: *software metrics*, *software architecture*, *package*, *software package metrics*, *Martin's metric*

Not everything that can be counted counts,

and not everything that counts can be counted.

— Albert Einstein

Sisältö

Kuvat

Taulukot

Symboliluettelo

A	Abstraktiusaste
ACD	Komponenttiriippuvuuden keskiarvo
ADP	Syklittömien riippuvuuksien periaate
AHF	Attribuutin piilotuskerroin
AIF	Attribuutin periytymiskerroin
C_a	Saapuvat kytkökset
C_e	Lähtevät kytkökset
CBM	Komponenttien kytkeytyneisyys
CBMC	Luokan jäsenten kytkeytyneisyys / Komponentti-luokka -kytkeytyminen
CBO	Luokkien välinen kytkeytyminen
CC	Komponentin luokka-asiakkaiden määrä
CCD	Komponentin kumulatiivinen riippuvuus
CDG	Luokkien riippuvuusgraafi
CLF	Ryvästyskerroin
CLM	Kommenttirivejä metodia kohti
COF	Kytkeytymiskerroin
CP	Asiakaskomponenttien määrä
CPC	Komponentin konteksti-riippuvainen koheesio
CR	Koheesiokerroin
CRSS	Luokan saavutettavuus
D	Etäisyys pääsarjasta
DAC	Tietotyyppikytkeytyminen
DCO	Koheesioaste
DIT	Perintäpuun syvyys
FP	Toimintopiste
FPA	Toimintopisteanalyysi
GQM	Tavoite-kysymys-metriikka -malli
H	Komponentin koheesio
HNL	Hierarkiataso
ICP	Tietovuokytkeytyminen
I	Komponentin epästabiilius

IUR	Rajapintojen käyttöaste
IDR	Sisäisten riippuvuuksien suhde
ILCO	Koheesion puute
LCC	Luokan löyhä kiinnevoima
LCIC	Koheesion puute asiakkaissa
LOC	Lähdekoodirivien lukumäärä
lLOC	Loogisten rivien lukumäärä
LCOP	Koheesion puute komponentissa
LCOM	Koheesion puute metodeissa
MHF	Metodin piilotuskerroin
MIF	Metodin periytymiskerroin
MPC	Viestinvälityskytkentä
MUI	Moniperinnän käyttö
MVC	Malli-näkymä-ohjain -arkkitehtuuri
N_a	Abstraktien luokkien määrä
N_c	Luokkien määrä
NC	Syklien lukumäärä
NCCD	Normalisoitu kumulatiivinen riippuvuus
NCM	Luokkametodien määrä
NCT	Poistettujen luokkien määrä
NCV	Luokkamuuttujien määrä
NIM	Instanssimetodien määrä
NIV	Instanssimuuttujien määrä
NMA	Lisättyjen metodien määrä
NMI	Perittyjen metodien määrä
NMO	Korvattujen metodien määrä
NOA	Attribuuttien määrä / Esi-isien määrä
NOAM	Aksessorimetodien määrä
NOC	Lasten lukumäärä
NOD	Perijöiden lukumäärä
NOM	Metodien lukumäärä
NOP	Vanhempien lukumäärä
PIM	Julkisten metodien määrä
POF	Polymorfismikerroin
PPM	Parametreja metodia kohti
RF	Uudelleenkäyttökerroin
RFC	Luokan vastausmäärä
RIC	Attribuutin suhteellinen kytkeytyminen
RMC	Metodin suhteellinen kytkeytyminen
SAP	Vakaan abstraktiuden periaate
SCC	Yhteisen kontekstin koheesio
SDP	Stabiilin riippuvuuden periaate
SIX	Erikoistumisindeksi
TCC	Luokan tiukka kiinnevoima
TCF	Tekninen vaativuus
UFC	Alustava toimintopistemäärä
WMC	Painotettujen metodien lukumäärä

Luku 1

Johdanto

Ohjelmistotuotannon historiassa on vuosikymmenen aikana nähty useita epäonnistumisia. Ongelmia on löytynyt niin aikatauluissa ja budjeteissa pysymisestä kuin tuotteiden laadusta. Menetelmiä ohjelmistotekniikan vaikeuksien voittamiseksi on kehitetty useita, joista ohjelmistometriikat ja -arkkitehtuurit ovat kaksi hyvin erilaista, mutta tehokasta lähestymistapaa. Metriikat tai arkkitehtuurit eivät kuitenkaan ole ohjelmistoteollisuuden hopealuoteja, mutta oikein käytettyinä yhdessä muiden työvälineiden kanssa ne auttavat rakentamaan laadukkaita tuotteita.

Ohjelmistometriikat arvioivat ohjelmistoa tai sen osia eri tavoin. Esimerkiksi tuotemittojen on tarkoitus löytää ohjelmiston osia, jotka saattavat sisältää virheitä. [63] Näiden mahdollisten häiriöiden poistamiseksi voidaan laadullisesti heikkoja osia testata ja arvioida muita tarkemmin. Mittoja voidaan käyttää myös prosessin tai resurssien arvioimiseen.

Harju ja Koskela paheksuvat termin ohjelmistometriikka käyttämistä "*kvantitatiivisesti orientoituneessa kielenkäytössä*", sillä matematiikassa metriikka on vakiintunut käsite metristen avaruuksien yhteydessä. [77, s. 85] Heidän vastasuosituksensa on ohjelmistomitoista puhuminen. Alan termistössä ohjelmistometriikka on kuitenkin vakiinnuttanut asemansa ja tässä opinnäytteessä termejä käytetään sekaisin, synonyymeinä toisilleen.

Ohjelmistoarkkitehtuurit ovat korkean tason suunnittelun menetelmä, joka auttaa hallitsemaan järjestelmän monimutkaisuutta jakamalla kokonaisuus pienempiin osiin. Arkki-

tehtuuri sisältää osiin jakamisessa syntyneiden arkkitehtuurikomponenttien väliset suhteet ja periaatteet niiden kehittämiseksi. [102] Arkkitehtuurin, kuten ohjelmistometriikoidenkin tarkoitus on parantaa ohjelmiston laatua. Arkkitehtuurin avulla hallitaan järjestelmän monimutkaisuutta, sillä se auttaa rakentamaan ja ylläpitämään kokonaisuutta huomattavasti pienemmissä, helpommin ymmärrettävissä ja hallittavissa osissa.

Tässä opinnäytteessä tarkastellaan ohjelmistometriikoita arkkitehtuuritasolla. Arkkitehtuurista voidaan tunnistaa vielä kaksi kerrosta: komponentti- ja järjestelmätasot. Työssä perehdytään kummallekin alatasolle esitettyihin mittoihin sekä pohditaan ominaisuuksia, joita arkkitehtuuritasolta voidaan mitata. Sekä komponentti- että järjestelmätasolle esitellään useita piirteitä, joiden avulla arkkitehtuuritason suunnittelua voidaan arvioida. Piirteitä myös arvioidaan kriittisesti, jotta niiden avulla voitaisiin muodostaa tehokas, mutta helposti hallittava ohjelmistometriikkapaketti arkkitehtuuritasolle.

Arkkitehtuuritasolle esitellyistä mitoista keskitytään erityisesti Robert C. Martinin [120] metriikkaan, sillä se on laajin komponenteille määritelty mittapaketti. Opinnäytteessä tutkitaan Martinin mitan teoreettista ja empiiristä kelvollisuutta komponenttimetriikaksi. Viiden avoimen lähdekoodin ohjelmaa arvioidaan metriikalla. Mitalla pyritään myös tunnistamaan komponenteista ohjelman laatua heikentävää suunnittelua.

Opinnäytteessä myös esitellään uusi komponenttien koheesiomitta H', jonka laskenta perustuu komponentin sisäisten kytkösten mallintamiseen graafilla. Mittaa koetellaan Briandin, Morascan ja Basilin [35] koheesiokriteereillä, sekä erilaisilla arviointikehyksillä. Teoreettiselta taustaltaan mitta todetaan kelvolliseksi.

Klassisia ja oliometriikoita esitellään luvussa 2. Lisäksi luvussa tarkastellaan luokkien mitattavia ominaisuuksia. Erityisesti tutkitaan koheesio- ja kytkeytymismittoja sekä niille esiteltyjä luokittelukehyksiä. Ohjelmistoarkkitehtuureihin syvennytään luvussa 3. Tärkeimmät arkkitehtuurityylit sekä suunnittelu- ja antisuunnittelumallit esitellään. Luvussa tarkastellaan myös arkkitehtuurin osia ja ominaisuuksia sekä niiden dokumentoimista.

Arkkitehtuuritason metriikoita ja määritelmiä esitellään luvussa 4. Martinin mitan lisäksi tarkastellaan Ducassen, Lanzan ja Ponision [57] sekä Ponision [144] määrittelemiä komponenttimittoja. Luvussa perehdytään myös komponenttien ja järjestelmien mitattaviin ominaisuuksiin. Lopuksi esitellään lista piirteistä, joita arkkitehtuuritasolta voisi mitata.

Luku 5 keskittyy Martinin metriikan kelpuuttamiseen. Teoreettisen tarkastelun tarkoituksena on osoittaa mitta kelvolliseksi komponenttien ominaisuuksien mittariksi. Empiirisessä kokeessa mitalla pyritään tunnistamaan huonolaatuisia komponentteja. Luvussa esitellään myös H' parannusehdotuksena Martinin koheesiomitalle. Yhteenveto työn sisällöstä esitellään luvussa 6.

Luku 2

Ohjelmistometriikat

Kyky ohjelmien mittaamiseen sanotaan olevan se askel, joka erottaa ohjelmistotekniikan perinteisistä insinöörialoista. [46] Mittareiden avulla voidaan suunnitella, arvioida ja ohjata kehitysprojektien etenemistä. Ohjelmistometriikat (engl. *Software Metrics*) ovat työvälineitä ohjelmiston tai sen osien ominaisuuksien arvioimiseksi mittaluvuin. [110]

Mittoja voidaan luokitella lukuisilla tavoilla. Esimerkiksi staattisen ja dynaamisen metriikan ero on, voidaanko mittaus tehdä ilman ohjelmakoodin suorittamista (staattinen) vai ainoastaan ajettavasta ohjelmasta (dynaaminen). [77] Huomattavasti monipuolisemman luokittelun antavat Fenton ja Pfleeger [63], jotka tarjoavat metriikoille kolme pääkategoriaa: prosessi-, tuote- ja resurssiluokat.

Fentonin ja Pfleegerin [63] pääluokat jakautuvat vielä sisäisten ja ulkoisten ominaisuuksien mittoihin. Ensimmäiseen alakategoriaan kuuluvat metriikat, joita voidaan mitata tuotteesta, prosessista tai resursseista itsestään ilman niiden käyttäytymistä. Ulkoisten ominaisuuksien metriikat ovat sellaisia, joissa huomioidaan myös ympäristö. Esimerkiksi ohjelmiston käytettävyys tai työryhmän tuottavuus ovat ulkoisia ominaisuuksia.

Seuraavissa alaluvuissa keskitytään staattisesti kerättäviin sisäisten ominaisuuksien tuotemetriikoihin. Klassisten, osittain jo historiallisten, metriikoiden esittelyn lisäksi käsitellään joukkoa olioparadigmalle suunniteltuja mittoja. Luvun lopuksi tarkastellaan lyhyesti metriikoiden käyttämisen syitä ja seurauksia.

2.1 Klassiset ohjelmistometriikat

Tässä opinnäytteessä klassisilla metriikoilla käsitetään 60- ja 70-luvuilla kehityt ohjelmistomitat, joita yhä käytetään jossain määrin teollisuudessa. Lähdekoodirivien lukumäärä, toimintopisteanalyysi, McCaben syklomaattinen kompleksisuus ja Halsteadin metriikat ovat yksinkertaisia, monesti kritisoituja ja tunnettuja esimerkkejä ohjelmien sisäisten ominaisuuksien mittareista.

Seuraavissa alaluvuissa nämä metriikat on jaoteltu karkeasti joko ohjelman koon tai kompleksisuuden mittareiksi. Kategorisointi perustuu mitan alkuperäiseen käyttöön, mutta myöhemmin esimerkiksi lähdekoodirivien lukumäärää on käytetty mm. laadun, kompleksisuuden ja tuottavuuden mittaamiseen. [62]

Halsteadin metriikkaperheestä löytyy mittareita implementaation kestosta ohjelman kokoon ja virhetiheyteen. Luokittelu ainoastaan kompleksisuusmitaksi on keinotekoinen, mutta esimerkiksi lähteet [150, 42, 63, 94] muistavat mainita Halsteadin yhtenä varhaisimmista monimutkaisuusmitoista.

2.1.1 Kokometriikat

Koko on ollut ensimmäisiä ohjelmistoista mitattuja ominaisuuksia, jonka indikaattorina on käytetty niin reikäkorttien kuin lähdekoodirivien lukumäärää. Ohjelmakoko on mielenkiintoinen ominaisuus, sillä se on tärkeässä osassa monessa johdannaismetriikassa, mm. työmäärän, työtehon ja virhetiheyden arvioinneissa. Kokometriikoita voi kuitenkin käyttää myös väärin. Esimerkiksi työntekijöiden tuottavuutta voidaan yrittää karkeasti mitata kokometriikoilla. Seuraavaksi on esitelty kaksi klassista kokometriikkaa: lähdekoodirivien lukumäärä ja toimintopisteet.

Lähdekoodirivien lukumäärä

Lähdekoodirivien lukumäärä (engl. *Lines of Code*, LOC) on yksi vanhimmista käytetyistä ohjelmistometriikoista. Perinteisesti sillä on mitattu vain ohjelman kokoa [80], mutta metriikkaa käytetään myös arvioimaan työn tuottavuutta, ohjelman kehittämiseen vaadittavaa työmäärää ja virhetiheyttä. [94] LOC on suosittu metriikka, sillä sen laskeminen on helposti automatisoitavissa ja tulos on ymmärrettävä.

LOC:n laskeminen ei ole kuitenkaan yksiselitteistä, sillä parantaakseen luettavuutta ohjelmoijat lisäävät tyhjiä rivejä ja selventäviä kommentteja ohjelmakoodin joukkoon. Arvioitaessa työmäärää tällaiset rivit ovat eriarvoisessa asemassa verrattuna monimutkaista algoritmia toteuttavaan lähdekoodiriviin. Kirjallisuudesta löytyykin useita ehdotuksia LOC:n laskemiseksi, joista seuraavaksi lyhyesti esiteltynä Conten ym. ja Jonesin laskentatavat.

Yksittäiseksi koodiriviksi Conte, Dunsmore ja Shen [52, s. 34] määrittelevät "*minkä tahansa rivin, joka ei ole kommentti- tai tyhjärivi*". Rivit, joissa on useampi kuin yksi ohjelmalause, lasketaan vain kerran. [52] Määritelmän ongelma on, että erilaiset ohjelmointityylit ja muotoilutottumukset saattavat tuottaa huomattavan eron rivilukumäärään.

Capers Jones [91] tarkentaa määritelmää jakamalla lähdekoodirivien lukumäärän fyysiseen (engl. *Physical Line of Code*) ja loogisen (engl. *Logical Line of Code*, lLOC) osaan. Fyysisessä LOC:ssä on mukana kaikki lähdekoodirivit — niin kommentit kuin tyhjätkin. Loogisessa yksi ohjelmalause (engl. *statement*) eli käsky, vastaa yhtä riviä. [91]

Listauksessa 2.1 `parity_short($in)` ja `parity_huge($in)` toteuttavat saman toiminnallisuuden: kumpikin funktio tulostaa, onko parametrina annettu argumentti parillinen vai ei. Ensimmäinen funktio on toteutettu Conten laskentatavan mukaan yhdellä lähdekoodirivillä; jälkimmäinen vie väljästi kirjoitettuna 11 riviä. Jonesin loogisia rivejä kummassakin toteutuksessa on kolme: kaksi tulostuslausetta sekä ehtolause.

Lähdekoodirivien lukumäärän laskeminen on monimutkaistunut uudempien, uudelleenkäytettävyyttä tukevien ohjelmointikielien myötä. Esimerkiksi visuaalisilla kehitys-

```
1 function parity_short($in){if($in % 2 == 0) print("true");else
    ↪ print("false");} #Tulostaa true/false
    ↪ parillisuudelle/parittomuudelle.
2
3 /*
4  * Funktio tulostaa "true", jos annettu
5  * muuttuja on parillinen. Muuten tulostetaan
6  * "false".
7  */
8 function parity_huge($in)
9 {
10  if($in % 2 == 0)
11  {
12    print("true");
13  }
14  else
15  {
16    print("false");
17  }
18 }
```

Listaus 2.1: Kaksi funktiota, jotka toteuttavat saman toiminnallisuuden.

työkaluilla voidaan luoda kymmeniä rivejä lähdekoodia yhdellä hiiren napautuksella. Entä pitäisikö laskennassa huomioida luokan perimällä saamat piirteet olioperustaisissa kielissä? Alalla ei ole olemassa standardisoitua laskentatapaa LOC:lle. [91]

On huomattava, että LOC:in oli alun perin tarkoituksena toimia vain konekielisten ohjelmien koon mittana. [80] Mittaa käytetään nykyään laajasti mm. tuottavuuden (ohjelmariviä kuukautta kohti) ja virhetiheyden (virhettä tuhatta riviä kohti) arvioimiseen ja ennustamiseen. Eri kielillä tai ohjelmointityyleillä rakennettujen sovelmien keskinäinen vertaaminen LOC:hen pohjautuvilla metriikoilla on kuitenkin järjetöntä. [91] Esimerkiksi yksinkertaisen *Hello World* -ohjelman toteuttaminen onnistuu joillain kielillä muutamalla rivillä, mutta toisilla voi vaatia useita kymmeniä lähdekoodirivejä. Tämän vuoksi virhetiheyden, koon tai työn tuottavuuden mittaaminen eri kielillä toteutettujen järjestelmien kesken kertoo enemmän käytettävästä ohjelmointityylistä -ja kielestä kuin sovelmien kes-

kinäisestä paremmuudesta.

Basili ja Hutchens [22] sekä Shepperd [150] suosittelevat uusien metriikoiden arvioimista vasten LOC:n käyttäytymiseen vastaavan ominaisuuden mittarina. Esimerkiksi lähdekoodirivien lukumäärä on näyttänyt suoriutuvan hyvin myös kompleksisuuden mittana. [150] Vaikka koodirivien määrä on hyödyllinen mitta, se ei ole metriikkamaailman hopealuoti. Kaikkea mahdollista ei tarvitse, eikä kannata yrittää mitata LOC:llä.

Toimintopisteanalyysi

Allan Albrecht kehitti vuonna 1977 IBM:llä vaihtoehtoisen tavan LOC:lle mitata ohjelmiston kokoa ja työn tuottavuutta. [161] Albrechtin tarkoituksena oli luoda metriikka, joka olisi riippumaton ohjelmointikielestä: yhtäläisen toiminnallisuuden toteuttavat ohjelmat ovat yhtä laajoja riippumatta siitä, millä kielellä ne on kirjoitettu. Albrecht kollegoineen kutsui kehittämäänsä menetelmää toimintopisteanalyysiksi (engl. *Function Point Analysis*, FPA). [91]

Toimintopisteanalyysiä on käytetty onnistuneesti arvioimaan ohjelmistoprojektin kestoa, tuottavuutta ja vaatimusten muutosten hintaa. [161] IBM:n jälkeen FPA:n kehityksestä on vastannut *International Function Point Users Group* (IFPUG), joka toistuvasti uudistaa metriikkaa ja kouluttaa sen käyttäjiä toimintopistearvioiden yhtenäistämiseksi. [84]

IFPUG [84] määrittelee analyysin yhdeksi päätavoitteeksi normalisoidun mitan tarjoamisen eri projekteilla ja organisaatioille. Järjestö toisaalta väittää FPA:n käyvän kompleksisuusmitasta ja sen käyttämisen parantavan sekä tuottavuutta että ohjelman laatua. FPA nojautuu kuitenkin vahvasti arvioijan katsomuksiin, minkä vuoksi voidaan kyseenalaistaa eri organisaatioiden ja laskijoiden toimintopistemäärien vertailu.

Albrechtin alkuperäisessä metriikassa ohjelmiston kokoa mitataan arvioimalla viittä mittaria: syötteiden ja tulosteiden määrää, käyttäjän interaktiota, ulkoisia liittymiä ja tiedostoja. Näin saaduille tehtäville arvioidaan monimutkaisuus asteikoilta “*helppo*”, “*keski-*

vaikea", "*vaikea*". Kaavassa 2.1 alustava toimintopistemäärä (engl. *Unadjusted Function Point Count*, UFC), on saatu laskemalla yhteen viiden mittarin ja kolmen vaikeustason muodostamien viidentoista kategorian edustajat painoarvoineen. [63]

$$\text{UFC} = \sum_{i=1}^{15} tapahtumat_i \times paino_i \tag{2.1}$$

$$\text{FP} = \text{UFC} \times \text{TCF} \tag{2.2}$$

Lopullinen toimintopistemäärä (FP) saadaan kertomalla UFC teknisellä haastavuudella (engl. *Technical Complexity Factor*, TFC) kuten kaavassa 2.2 on esitetty. Tekninen haastavuus saadaan arvioimalla jokainen Albrechtin neljästätoista teknisestä tekijästä painoarvoilla 0–5 ja laskemalla nämä yhteen (kaava 2.3). Teknisiä tekijöitä ovat esimerkiksi suorituskyky, asennuksen helppous ja automaattinen päivittyminen. Paino 0 tarkoittaa tekijän olevan ohjelman kannalta merkityksetön; välttämättömyyttä kuvaa paino 5. [63]

$$\text{TCF} = 0.65 + 0.01 \sum_{i=1}^{14} F_i \tag{2.3}$$

Tekninen vaativuus voi vaihdella välillä 0.65–1.35. Arvioimalla tekniset tekijät kohtuullisiksi (2) tai keskitasoiksi (3) TFC:n arvo jää lähelle yhtä, jolloin alustava toimintopistemäärä ei huomattavasti eroa lopullisesta.

FPA:ta on arvosteltu erityisesti sen yksinkertaisuudesta. Charles Symons [154] kritisoi UFC:n kolmea monimutkaisuusluokkaa ja neljäätoista teknistä tekijää riittämättömiksi kuvaamaan kaikkia mahdollisia järjestelmiä. Samoin hän kyseenalaistaa Albrechtin tekemät havainnot IBM:n projekteista yleispäteviksi muissa kehitysympäristöissä ja -yhteisöissä.

Suurimmat metriikan käytön ongelmat ovat kuitenkin sen vaatima ammattitaito ja työmäärä. Tarkka funktiopisteanalyysi on monimutkainen operaatio, joka vaatii sertifioidun ammattilaisen käyttämistä. [91] Teknisen vaativuuden subjektiivisella painoarvojen

valinnalla on jopa ± 35 % vaikutus lopputulokseen [63], minkä vuoksi IFPUG yrittää yhtenäistää arvioimiskäytäntöjä koulutuksella ja ohjeistuksilla.

Analyysin laskemisen vaativuus on myös huomattava. Asiantuntija käsittelee Kemererin [97] havaintojen mukaan keskimäärin yhdessä tunnissa sadan pisteen edestä toimintopistedataa. Tuhansien pisteiden projekteille pisteiden arvioiminen saattaisi viedä viikkoja, mikä kuluttaa jo huomattavasti käytettävissä olevia resursseja.

FPA:n suosio ja ongelmat ovat poikineet useita eri variaatioita alkuperäisestä funktiopisteanalyysistä. Näistä tunnetuimpia ovat Charles Symonsin *Mark II Function Points* ja Capers Jonesin ominaisuuspisteet (engl. *Feature Points*). Symonsin metriikka on saavuttanut jalansijaa Isossa-Britanniassa, mutta näiden vaihtoehtojen käyttäjiä on vain pieni joukko maailmanlaajuisesti verrattuna FPA:n kannattajiin. [91]

Jonesin ominaisuuspisteet kehitettiin 80-luvulla reaaliaika- ja järjestelmäohjelmistoille, mutta IFPUG:in päivittäessä ohjeitaan tarve toiselle laskentatavalle hiipui. [91] Myös Symons [154] koki Albrechtin FPA:n osin puutteelliseksi ja yksinkertaistetuksi. Hänen Mark II FPA:nsa yrittää paikata näitä puutteita erityisesti sisäisen kompleksisuuden mallintamisessa. Symonsin versio toimintopisteistä esitellään lähteessä [155].

2.1.2 Kompleksisuusmetriikat

Erilaisia monimutkaisuuksia voidaan tunnistaa useita. Tällaisia ovat esimerkiksi ongelman ja algoritmin aika- ja tilakompleksisuudet. [63] Tässä alaluvussa keskitytään kuitenkin ohjelman rakenteelliseen monimutkaisuuteen, minkä voidaan ajatella tarkoittavan, kuinka vaikeaa on ymmärtää ohjelman osan tai algoritmin toimintaa. Rakenteellisten kompleksisuusmetriikoiden takana on laaja uskomus [63, 80, 94], että yksinkertaiset rakenteet johtavat pienempään virhemäärään. Tämä osaltaan parantaa ohjelman laatua ja säästää rahaa ylläpitovaiheessa vähentyneellä korjaustarpeella. Seuraavaksi tarkastellaan klassisista kompleksisuusmetriikoista Halsteadin mittoja ja McCaben syklomaattista kompleksisuutta.

Halsteadin metriikat

Yksi historiallisesti tärkeistä metriikoista on Maurice Halsteadin kirjassaan *Elements of Software Science* vuonna 1977 julkaisema mittariperhe. [76] Halsteadin metriikat (engl. *Halstead's Software Science*) on jättänyt jälkensä ohjelmistomittojen kehittämiseen, mutta nykyinen suhtautuminen metriikoihin on hyvin kriittinen. Esimerkiksi Stephen Kan [94] ja Conte ym. [52] kirjoittavat, etteivät empiiriset tutkimukset tue metriikoiden antamia tai ennustamia arvoja. Henderson-Sellers [80] toteaa Halsteadin johtaneen kaavansa ideaalisista algoritmeista ja kyseenalaistaa näin niiden laskennallisen sopivuuden ohjelmointikielille. Fenton ja Pfleeger [63] osoittavat mittausteoreettisia puutteita Halsteadin kaavoissa. Kriittisimmin suhtautuvat Card ja Glass [42], joiden mielestä ohjelmistoala voi turvallisesti unohtaa Halsteadin metriikat nykymuotoisina.

Halstead listaa neljä perussuuretta, joiden mittaaminen ohjelmakoodista voidaan automatisoida [76]:

$n_1 =$ Erilaisten operaattorien määrä

$n_2 =$ Erilaisten operandien määrä

$N_1 =$ Operaattoreiden esiintymien kokonaismäärä

$N_2 =$ Operandien esiintymien kokonaismäärä

Perussuureiden, operandin ja operaattorin, määritelmä eroaa hieman totutuista: Operandi tarkoittaa Halsteadille mitä tahansa tyyppinimeä, muuttujaa tai vakiota. Ohjelmointikielen varatut sanat, binäärioperaatiot sekä kontrollirakenteet lasketaan operaattoreiksi. [157]

Operaattoreiden ja operandien pohjalta johdetaan joukko Halsteadin arvoja [76]:

$$\text{Sanasto (engl. } \textit{Vocabulary}) \qquad n = n_1 + n_2 \tag{2.4}$$

$$\text{Pituus (engl. } \textit{Length}) \qquad N = N_1 + N_2$$

$$\approx n_1 \log_2(n_1) + n_2 \log_2(n_2) \tag{2.5}$$

$$\text{Laajuus (engl. } \textit{Volume}) \qquad V = N \log_2(n) \tag{2.6}$$

$$\text{Ohjelman laajuus (engl. } \textit{Level}) \qquad L = \frac{V^*}{V} = (\frac{2}{n_1})(\frac{n_2}{N_2}) \tag{2.7}$$

$$D = \frac{1}{L} = \frac{V}{V^*}$$

$$= (\frac{n_1}{2})(\frac{N_2}{n_2}) \tag{2.8}$$

$$\text{Työmäärä (engl. } \textit{Effort}) \qquad E = \frac{V}{L} \tag{2.9}$$

$$\text{Virheiden määrä (engl. } \textit{Faults}) \qquad B = \frac{V}{S^*} \tag{2.10}$$

V^* on minimaalisen ohjelman laajuus eli sellaisen, joka kootaan valmiista osista. S^* on psykologinen vakio, joka kuvaa virheiden esiintymistaajuutta. Näin kaava 2.10 tarkoittaa, että Halsteadin mielestä ohjelmassa on aina vakioitu määrä virheitä riippumatta ohjelmointityökaluista, tekijöistä tai prosessista. [94]

McCaben syklomaattinen kompleksisuus

Thomas McCabe julkaisi vuonna 1976 syklomaattisen kompleksisuuden (engl. *Cyclomatic Complexity*) metriikan, jonka tarkoituksena on mitata ohjelman osan itsenäisten suorituspolkujen lukumäärää. [123] Mitta kertoo, kuinka monta erilaista reittiä pitkin ohjelmapalasen suoritus voi edetä sen alustamisesta päättymiseen.

Syklomaattisen kompleksisuuden taustalla on idea esittää ohjelman suorituksen eteneminen graafina, jonka solmut esittävät ohjelmalauseita ja kaaret siirtymiä näiden välillä. Mahdollisten suorituspolkujen lukumäärä saadaan kaavasta 2.11, missä G on ohjelman suorituksen kulku graafina, e_G graafin kaarien lukumäärä, n_G solmujen määrä ja p_G kytkemättömien osien määrä [123]:

$$v(G) = e_G - n_G + 2p_G \tag{2.11}$$

McCaben metriikan tarkoituksena on auttaa ohjelmien testauksessa ja ylläpitämisessä. [123] Kompleksisten ohjelmien testaaminen on työläämpää, sillä testien kirjoittaminen kaikille mahdollisille suorituspoluille vie aikaa. Samoin näiden ymmärtäminen ja ylläpi-

Taulukko 2.1: Syklomaattisen kompleksisuuden ja riskin välinen suhde.

Syklomaattinen kompleksisuus	Riskiarvio
1-10	Ei suurta riskiä.
11-20	Kohtalainen riski
20-50	Suuri riski
50+	Erittäin suuri riski

Edmond VanDoren [162]

täminen vaikeutuu. McCabe suositteleekin monimutkaisten moduulien — kompleksisuuden arvo suurempi kuin 10 — jakamista alirutiineihin tai kirjoittamista uudelleen. [123]

Koska asiantuntijat uskovat laajasti ohjelmistotuotteen sisäisen kompleksisuuden ja laadun väliseen yhteyteen, McCaben metriikan kaltaisille mittareille on kysyntää. Joel Troster [160] löysikin mittauksissaan korrelaation syklomaattisen kompleksisuuden ja virhemäärän suhteen, mutta havaitsi myös McCaben kompleksisuuden kasvavan ohjelmakoon kanssa.

Martin Shepperd [150] kyseenalaistaa koko metriikan hyödyllisyyden, väittäen sen vain heijastavan LOC:n vastaavia tuloksia. Shepperd pitää syklomaattisen kompleksisuuden lähtökohtaa liian yksinkertaistettuna, sillä se ei huomioi haarautumispäätöksiin liittyvää kontekstia. McCaben kompleksisuus myös kasvaa ohjelmakoodin modularisoinnin myötä [150], vaikka ohjelman pilkkomisen pienempiin pitäisi yksinkertaistaa sen ymmärtämistä ja testaamista.

Fenton ja Pfleeger [63] pitävät metriikkaa riittämättömänä yleiseksi kompleksisuuden mittariksi, mutta myöntävät sen näyttävän vaikeasti testattavat osat. Kan suositteleekin [94] McCaben metriikan käyttämistä virhealttiin ohjelmakoodin tunnistamisessa ja testauksen priorisoimisessa. Esimerkiksi valintaperusteina voidaan käyttää taulukon 2.1 riskikategorioita.

2.2 Oliometriikat

Olioparadigman korvatessa vanhemmat ohjelmointisuuntaukset, eivät klassiset metriikat olleet riittäviä paradigman uudenlaisten ominaisuuksien mittaamisen. Proseduraalisten kielten ohjelmistometriikat eivät ymmärrä oliofilosofian käsitteitä, kuten luokkaa, perintää tai viestien välitystä. [109] Tämä on inspiroinut tutkijoita kehittämään uusia oliospesifejä mittoja.

Oliometriikoita on kehitetty kiihtyvällä tahdilla, esimerkiksi Purao ja Vaishnavi [146] löysivät yli 350:tä olioparadigmalle määriteltyä metriikkaa. Seuraavassa on esitelty kaksi kritisoitua, kiiteltyä ja käytettyä metriikkapakettia sekä useita sekalaisia olioiden ja luokkien ominaisuuksia mittoja. Luvuissa 2.3 ja 2.4 keskitytään vielä enemmän luokan yhtenäisyyden ja olioiden yhteistoiminnan metriikoihin.

2.2.1 Chidamberin ja Kemererin metriikkapaketti

Shyam Chidamber ja Chris Kemerer esittelivät [46] vuonna 1991 kuuden ohjelmistomitan pakettinsa, jota tarkennettiin [47] vielä vuonna 1994. Nämä olivat ensimmäiset empiirisesti arvioidut [109, 47] ja formaalisesti määritetyt [46] metriikat olio-orientoituneille kielille.

Painotettujen metodien lukumäärä

Chidamberin ja Kemererin [46] ensimmäinen metriikka on painotettujen metodien lukumäärä (engl. *Weighted Methods per Class*, WMC). WMC saadaan laskemalla yhteen luokan jokaisen metodin kompleksisuus (kaava 2.12). Yksittäinen kompleksisuus c_i voidaan laskea esimerkiksi McCaben syklomaattisella kompleksisuudella [109], Halsteadin vaikeudella tai tulkita vakioksi jokaiselle metodille.

WMC on muodollisesti annettuna [46]:

$$\text{WMC} = \sum_{i=1}^{n} c_i \tag{2.12}$$

Kalakota, Rathnam ja Whinston [93] pitivät tarkan kompleksisuusmetriikan puuttumista mittarille vahingollisena, mutta Chidamber ja Kemerer [47] mainostavat ominaisuuden lisäävän WMC:n joustavuutta. Ainoa vaatimus käytettävälle monimutkaisuusmitalle on sen arvojen yhteenlaskettavuus.

Chidamber ja Kemerer [46] väittävät luokan metodien määrän ja kompleksisuuden ennustavan luokan kehittämisen ja ylläpitämisen työmäärää. He myös uskovat korkean WMC:n heikentävän luokan uudelleenkäytettävyyttä ja rasittavan luokan lapsia. Perijät saavat taakakseen kaikki luokan ominaisuudet, mikä monimutkaisella isäluokalla vaikeuttaa ymmärtämistä ja testaamista.

Perintäpuun syvyys

Paketin toinen metriikka on perintäpuun syvyys (engl. *Depth of Inheritance Tree*, DIT), joka kertoo kuinka syvällä perintähierarkiassa luokka on. Moniperinnässä DIT on pisimmän perintäpolun pituus luokasta juureen. [46] Juuriluokan DIT on 0.

Chidamber ja Kemerer [46] pitävät pitkää perintäpolkua haitallisena sen lisätessä lasten perimien piirteiden määrää ja näin monimutkaistaessa niitä. Samoin heistä pitkä perintäpolku luokkahierarkiassa monimutkaistaa ohjelmistosuunnittelua. Toisaalta syvällä hierarkiassa olevat luokat tukevat uudelleenkäyttöä kierrättämällä aiemmin toteutettuja piirteitä. [47]

Lasten lukumäärä

Kolmas metriikka on luokasta suoraan perivien, lapsiluokkien, lukumäärä (engl. *Number Of Children*, NOC). Chidamberin ja Kemererin [46] määritelmässä lapsenlapsia ei huomioida. Kirjoittajat [46] pitävät yleisesti parempana leveää kuin korkeaa perintäpuuta. He myös korostavat lasten määrän kertovan luokan asemasta luokkahierarkiassa. Korkean NOC:n luokat pitäisi priorisoida testauksessa huipulle, sillä niiden kokonaisvaikutus arkkitehtuurissa on muita suurempi.

Luokkien välinen kytkeytyminen

Luokkien välinen kytkeytyneisyys (engl. *Coupling Between Objects*, CBO) on paketin neljäs metriikka, joka mittaa olioiden keskinäisiä riippuvuuksia. Chidamber ja Kemerer [47] määrittelevät luokan kytkeytyneeksi toiseen, jos se käyttää suoraan toisen metodeja tai muuttujia. Kytkeytymisen määritelmää on tarkasteltu tarkemmin alaluvussa 2.4.

CBO on niiden luokkien määrä, joiden kanssa tarkasteltava luokka on kytkeytynyt. [47] Korkea CBO on vahingollinen luokan uudelleenkäytettävyydelle, sillä sen kierrättäminen yksistään vaikeutuu. Tiukasti toisiinsa kytkeytyneiden luokkien yksikkötestaus hankaloituu. [46]

Luokan vastausmäärä

Viides metriikka on luokan vastausmäärä (engl. *Response For a Class*, RFC). RFC on luokan omien metodien lukumäärä laskettuna yhteen kaikkien luokan kutsumien metodien määrällä. [46] Kutsupolkua ei kuitenkaan seurata ensimmäistä askelta pidemmälle.

Mitä suurempi RFC:n arvo on, sitä kompleksisemmaksi Chidamber ja Kemerer [46] luokan mieltävät. Korkea RFC vaikeuttaa luokan testaamista ja virheiden etsimistä, sillä testaajalta vaadittava ymmärrys koko järjestelmästä, ei vain testattavasta luokasta, kasvaa.

Koheesion puute metodeissa

Chidamerin ja Kemerin [46] viimeisen, kuudennen, metriikan lähestymistapa on käänteinen. Sen sijaan, että mitattaisiin luokan koheesiota, lasketaan sen puutetta metodeissa (engl. *Lack of Cohesion in Methods*, LCOM). Koheesion määritelmä on käsitelty tarkemmin luvussa 2.3.

Täydennetyssä metriikkapaketissa LCOM määriteltiin uusiksi ja esitettiin joukkojen avulla: P on sellaisten metodiparien joukko, jotka eivät käsittele samoja instanssimuuttujia. [47] Metodiparien joukkoa, joilla on yhteisiä instanssimuuttujia, merkitään Q:lla. LCOM saadaan vähentämällä P:n alkioiden määrä Q:n alkioiden määrästä (kaava 2.13).

$$\text{LCOM} = \begin{cases} |P| - |Q| & \text{kun} \quad |P| > |Q| \\ 0 & \quad\quad\quad\quad |P| \leq |Q| \end{cases} \tag{2.13}$$

Luokan korkea koheesio — pieni LCOM:in arvo — on toivottavaa, sillä silloin luokka on muodostettu olio-ohjelmoinnin periaatteiden mukaisesti vain yhdestä käsitteestä. Matala koheesio saattaa tarkoittaa luokan palvelevan useammassa roolissa, milloin se pitäisi jakaa pienempiin osiin. Chidamber ja Kemerer uskovat koheesiottoman luokan lisäävän virheiden mahdollisuutta kehitysprosessissa. [46]

Chidamber ja Kemerer havainnollistavat luokan koheesiota sillä, kuinka paljon metodit käyttävät luokan instanssimuuttujia. Heidän lähestymistapansa on tietokeskeinen, minkä kirjoittajat [47] arvelivat innostavan erilaisiin tulkintoihin — ja olivat harvinaisen oikeassa. Luvussa 2.3 on käsitelty osaa lukuisista ensimmäisen koheesiomitan seuraajaehdokkaista.

Kritiikkiä ja empiirisiä tuloksia

Churcher ja Shepperd [49] kritisoivat metriikkapaketin jättäneen määritelmät liian avoimeksi. Heidän vastaesimerkissään WMC:n arvo vaihteli samalla luokalla 12:sta 37:än riippuen mukaan otettavien metodien määritelmästä. Chidamberin ja Kemererin [48] vastineessa kehotetaan laskemaan kaikki luokkaan kirjoitetut metodit ja jättämään huomioimatta perittyjä piirteitä.

Hitz ja Montazeri [83] sekä Mayer ja Hall [121] osoittavat puutteita paketin mittausteoreettisissa perusteissa. Mayer ja Hall hyökkäävät vasten metriikkapaketin kykyä mitata olioideologian ominaispiirteitä. Mitat eivät esimerkiksi huomioi polymorfismia tai abstraktiota. He myös osoittavat puutteellisuuksia määritelmissä: esimerkiksi lasten lukumäärä edustaa varsin rajallista kuvaa kaikista luokan jälkeläisistä ja DIT luokan esivanhemmista.

Silti Lin ja Henryn [109] mittausten mukaan metriikkapaketti — ilman CBO:ta — toimii parempana ennustajana ylläpidettävyydelle kuin kokometriikat yksin. Basili kolle-

goineen [20] suositteli viittä metriikkaa virhealttius- ja laatumittariksi. He eivät löytäneet tutkimuksessaan LCOM:ille vastaavuutta virheiden kanssa.

2.2.2 Abreun MOOD-metriikat

Fernando Brito e Abreu kollegoineen [6] julkaisi vuonna 1994 kahdeksan mitan metriikkapaketin, joka keskittyi mittaamaan olio-ohjelmoinnin keskeisiä käsitteitä: perintää, kapselointia, informaation piilottamista, polymorfismia ja uudelleenkäyttöä. Myöhemmissä julkaisuissa [9, 10, 8] ryvästyskerroin (engl. *Clustering Factor*, CLF) ja uudelleenkäyttökerroin (engl. *Reuse Factor*, RF) jäivät pois paketista.

MOOD-paketin (engl. *Metrics for Object-Oriented Design*) metriikat ovat normalisoitu välille 0–1, missä nolla kuvaa ominaisuuden puuttumista ja yksi sen täydellistä toteutumista. Näin metriikan arvot voidaan laskea sekä yhdelle luokalle että luokkien muodostamalle joukolle, ja arvot ovat edelleen vertailukelpoisia. [6] Abreun metriikoiden tarkasteluissa summakaavan yläraja TC tarkoittaa laskennassa mukana olevien luokkien määrää. Kun TC:n arvo on 1, kertoimet lasketaan vain yhdelle luokalle.

Metriikkapaketista on julkaistu uusi, yhdeksällä uudella mitalla päivitetty versio nimeltä MOOD2. Osia alkuperäisen paketin metriikoista on uudelleenmääritelty tai hajotettu pienempiin osiin. [5] Tässä opinnäytteessä ei uudempaan pakettiin tutustuta, mutta määritelmät löytyvät Abreun julkaisuista [4, 7].

Piilotuskertoimet

Metodin (engl. *Method Hiding Factor*, MHF) ja attribuutin (engl. *Attribute Hiding Factor*, AHF) piilotuskertoimet mittaavat Abreun mukaan luokan kapselointia ja informaation kätkemistä. [6] MHF on piilotettujen metodien suhde kaikkiin metodeihin (kaava 2.14), ja AHF on vastaava attribuuteille (kaava 2.15).

Piilotuskertoimet määriteltiin uudelleen lähteessä [10] niin, ettei arvon laskeminen yksittäiselle luokalle enää onnistu. Opinnäytteessä pitäydytään tämän vuoksi alkuperäisissä

MHF:n ja AHF:n muodollisissa määrittelyissä [6]:

$$\text{MHF} = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)} \tag{2.14}$$

$$\text{AHF} = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)} \tag{2.15}$$

$M_h(C_i)$ tarkoittaa luokan C_i piilotettujen ja $M_d(C_i)$ kaikkien metodien lukumäärää. A_h ja A_d kuvautuvat samoin attribuuteille. C++ -esimerkissään Abreu laskee ylikuormitetut metodit erillisiksi, eikä huomioi perittyjä piirteitä. [9]

Abreu [6] halusi luoda kieliriippumattoman metriikkapaketin, mutta tarjota myös konkreettisia esimerkkejä. Piilotetun metodin ja attribuutin määritelmät ovat aina kielikohtaisia, mutta esimerkiksi C++:ssa piilotettuja ovat `private` tai `protected` suojatut metodit ja attribuutit. [9]

Metriikat ilman käyttöohjetta ovat hyödyttömiä, minkä vuoksi MOOD-paketin luojat haluavat myös neuvoa mittojen käyttämisessä. [9] He käyttävät ohjeistuksessaan elektroniikasta tutumpia yli- ja kaistanpäästösuodattimia analogiana: Ylipäästöheuristiikka varoittaa suunnittelijaa, kun metriikan arvo on liian pieni. Kaistanpäästöheuristiikka varoittaa silloin, kun arvo on annetun ylä- ja alarajan ulkopuolella.

AHF:lla ideaalinen arvo on 1, jolloin kaikki attribuutit on suojattu. Liian alhaiseksi tippuneen arvon pitäisi varoittaa suunnittelijaa, minkä vuoksi ylipäästöheuristiikka sopii ohjeistamaan AHF:n käyttöä. Vastaavasti kaistanpäästöheuristiikka sopii metodin piilotuskertoimelle, sillä pieni MHF:n arvo kuvastaa puutteellista abstraktiota. Korkea arvo taas heijastaa luokan julkisen toiminnallisuuden vähäisyyttä. [9] Esimerkiksi mittauksissaan Abreu ja kollegat [10] saivat MHF:lle 90 %-luottamusvälille arvot 0.154–0.387.

Periytymiskertoimet

Abreun toinen metriikkapari keskittyy olio-ohjelmoinnin yhteen tärkeimmistä ominaisuuksista. Metodin (engl. *Method Inheritance Factor*, MIF) ja attribuutin (engl. *Attribute*

Inheritance Factor, AIF) periytymiskertoimet kuvaavat kuinka monta prosenttia luokan metodeista ja attribuuteista on perittyjä.

Abreu kollegoineen [9] suosittelee käyttämään molempia periytymiskertoimia kaistanpäästöheuristiikalla. Olemattomat MIF:in ja AIF:in arvot kielivät paradigman tarjoaman uudelleenkäytettävyyden väärinymmärtämisestä. Toisaalta metriikan korkeat tulokset tarkoittavat luokan ymmärrettävyyden ja testattavuuden pienentyneen perimätaakan alla.

MIF:in ja AIF:in määritelmät ovat MOOD-paketista ainoat, jotka ovat säilyneet muuttumattomina. Muodolliset määrittelyt näille ovat [9]:

$$\text{MIF} = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)} \tag{2.16}$$

$$\text{AIF} = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)} \tag{2.17}$$

M_a on kaikki luokan käytettävissä olevat metodit ja M_i perityt metodit. A_i ja A_a ovat vastaavia attribuuteilla. Syrjäytettyjä, uudelleen määriteltyjä, piirteitä ei huomioida laskettaessa M_i:tä ja A_i:tä. [9]

Kytkeytymiskerroin

Kytkeytymiskerroin (engl. *Coupling Factor*, COF) kuvaa prosenttilukuna kuinka lähellä tarkasteltava järjestelmä on täysin kytkeytynyttä järjestelmää, missä jokainen luokka käyttää kaikkia muita luokkia. COF:in arvo maksimaalisessa järjestelmässä on 1. Tällöin järjestelmän osien uudelleenkäyttäminen yksinään on lähes mahdotonta.

Kytkeytymiskertoimen määritelmä on vaihdellut vuodesta toiseen. Vuoden 1995 paperissa Abreu [9] muutti määritelmää, mutta seuraavaan vuoden julkaisuissa [8, 10] käytettiin taas alkuperäistä laskentakaavaa. Tässä opinnäytteessä käytetään vuoden 1995 ma-

temaattista määritelmää, sillä siinä poistetaan perinnän vaikutus [9]:

$$\text{COF} = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} \textit{is\_client}(C_i, C_j) \right]}{TC^2 - TC - 2 \times |\bigcup_{i=1}^{TC} DC(C_i)|} \tag{2.18}$$

Kaavassa 2.18 $DC(C_i)$ on kaikkien luokasta C_i perivien luokkien, jälkeläisten, lukumäärä. Jälkeläisiä laskettaessa huomioidaan kaikki luokat, joihin ominaisuuksia on voinut periytyä. Tarkasteltavan järjestelmän maksimaalinen kytkeytyminen on $TC^2 - TC$. $2 \times |\bigcup_{i=1}^{TC} DC(C_i)|$ on perimän lisäämä enimmäiskytkeytyneisyys. Kaavan nimittäjä on tarkasteltavan kokonaisuuden maksimikytkeytyminen, kun huomiotta jätetään periytymisen vaikutukset. [9]

$$\textit{is\_client}(C_i, C_j) = \begin{cases} 1 & \text{joss} \quad C_i \Rightarrow C_j \wedge C_i \neq C_j \wedge \neg(C_i \rightarrow C_j) \\ 0 & \text{muulloin} \end{cases}$$

Luokka C_i määritellään luokan C_j asiakkaaksi, jos se käyttää suoraan luokan palveluita ($C_i \Rightarrow C_j$). Luokat eivät kuitenkaan saa olla perimysrelaatiossa ($\neg(C_i \rightarrow C_j)$), ja tarkastelusta jätetään huomioimatta luokan viittaukset itseensä ($C_i \neq C_j$).

Kytkeytymistä pidetään yleisesti huonona ominaisuutena luokkien välillä [128], mutta ohjelmia ei voida rakentaa ilman olioiden välistä yhteistoimintaa. Tämän vuoksi matalat arvot ovat hyväksyttäviä, mutta korkeita arvoja on varoittava. Kaistanpäästöheuristiikka on paras varoitusjärjestelmä myös COF:ille. [9]

Polymorfismikerroin

Olioparadigman polymorfismi kuvaa tilanteita, joissa kutsuja ei tiedä kuka palvelun pyyntöön vastaa. Kutsun sitova luokka saattaa olla mikä tahansa samaan perintäpuuhun kuuluvista luokista, jotka ovat ylikirjoittaneet kyseisen metodin. Polymorfismikerrointa (engl. *Polymorphism Factor*, POF) uudistettiin vuoden 1995 julkaisussa, missä se annettiin for-

maalisti muodossa [9]:

$$\text{POF} = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} \left[M_n(C_i) \times DC(C_i) \right]} \tag{2.19}$$

M_o on uudelleenmääriteltyjen, ja M_n uusien metodien lukumäärä. Kaavan osoittaja kuvaa luokan mahdollisten polymorfisten tilanteiden — jossa kutsuun voi vastata joku toinen luokka — lukumäärää. Jos luokan C_i kaikki perijät uudelleenmäärittelevät kaikki metodit, on kyse polymorfisten tilanteiden enimmäismäärästä. Kaavan 2.19 nimittäjä esittää tällaista maksimitilannetta. [9]

Polymorfismi auttaa vähentämään järjestelmän kokonaiskompleksisuutta, mutta samalla vaikeuttaa testausta ja virheiden etsimistä. Abreu suosittelee myös POF:lle kaistanpäästöheuristiikkaa, jossa liiallinen ja liian vähäinen käyttö varoittaisi kehittäjiä. [9]

Kritiikkiä ja empiirisiä tuloksia

Harrison, Counsell ja Nithi [78] vertasivat MOOD-pakettia vasten Kitchenhamin ja kollegoiden [100] esittämiä kriteerejä ja vahvistivat Abreun metriikat teoreettisesti kelvollisiksi, kunhan määritelmiä tarkennettiin. Abreun ja kollegoiden [9] mittauksissa paketti oli melko riippumaton ohjelman koosta, ainoastaan AHF ja MIF korreloivat kokomittojen kanssa. Kirjoittajat tosin uskoivat riippuvuuden selittyvän otannan suppeudesta, sillä vastaavasti määritellyt MHF ja AIF eivät osoittaneet riippuvuutta kokometriikoihin. [9]

Abreu ja Melo [10] koettelivat paketin arvoja samalla testimateriaalilla, millä Basili ym. [20] mittasivat Chidamberin ja Kemererin metriikkapakettia. Mittauksessa he löysivät pieniä riippuvuuksia MHF:n, MIF:n, AIF:n ja POF:n ja käytettyjen laatumittojen välille. Attribuutin piilotuskertoimelle ei löydetty mitään yhteyttä laatumittareihin. Parhaiten suoriutui COF, jolla oli korkea riippuvuussuhde kaikkiin kolmeen käytettyyn laatumittaan. Vastaavia tuloksia saivat Misra ja Bhavsar [130], joiden mittausten perusteella piilotus- ja polymorfismikertoimien korkeat arvot korreloivat pienemmän virhetiheyden kanssa.

Empiirisistä tuloksista huolimatta Mayer ja Hall [122] kritisoivat kovin sanoin Abreun

ja kollegoiden olioparadigman ymmärrystä: AIF mittaa attribuuttien periytymistä, mutta mitan perusta on epämääräinen. Perinnässä attribuutteja ei voi syrjäyttää: jos luokka C perii luokan D instanssimuuttujan $D.x$ ja määrittelee $C.x$:n "uudelleen", molemmat muuttujat säilyvät silti käytössä. Vastaavasti ylikirjoitettua metodia voidaan kutsua luokan perijöissä. Mayer ja Hall argumentoivat näiden havaintojen vievän pohjan polymorfismi- ja periytymiskertointen määritelmiltä.

MHF:ää ja AHF:ää luonnehditaan MOOD-paketin kapseloinnin mittareiksi [9], minkä Mayer ja Hall [122] kiistävät. Heidän mukaansa mittojen määrittelijät ovat samaistaneet informaation piilottamisen ja kapseloinnin, jotka ovat kaksi erillistä käsitettä. He myös pitävät metodien piilotuskerrointa mielenkiinnottomana ja turhana mittana, sillä kompleksisen funktion työmäärä voidaan jakaa pienempien, piilotettujen alifunktioiden tehtäväksi. Luokan kapselointi tai informaation kätkeminen eivät tästä muutu.

2.2.3 Muita oliometriikoita

Oliometriikoita on esitelty kymmeniä, ellei satoja, paradigman läpilyönnin jälkeen. Esimerkiksi Lorenz ja Kidd [112] määrittelevät kirjassaan *Object-Oriented Software Metrics* kahdeksan projekti- ja 27 sisäistä tuotemittaa, sekä lukuisia näistä johdettuja mittoja. Lanza ja Marinescu [107] käyttävät teoksessaan *Object-Oriented Metrics in Practice* kahtakymmentäneljää metriikkaa, joista osa on tosin tässä opinnäytteessä jo kuvattuja.

Metriikoita on ehdotettu monenlaisten piirteiden mallintamiseen. Esimerkiksi Lorenz ja Kidd [112] esittelevät mitat poisheitettyjen luokkien lukumäärälle (engl. *Number of Classes Thrown Away*, NCT), kommenttien määrälle metodeissa (engl. *Comment Lines per Method*, CLM) sekä parametrien määrän keskiarvolle (engl. *Parameters per Method*, PPM). Jälkimmäiselle, PPM:lle, he antavat vielä ylärajan: 0.7 parametria metodia kohti ylittävistä luokista kehotetaan tarkastamaan, miksi parametreja vaaditaan niin paljon.

On kyseenalaista, kuinka hyödyllisiä piirteitä edellä mainitut metriikat mittaavat. Esimerkiksi painoindeksin laskevalle metodille voidaan parametrina antaa joko liukuluvut

`weight` ja `height` tai luokan `Person` ilmentymä. Alkeistietotyypin varaan toteutettu versio on huomattavasti siirrettävämpi, sillä se ei ole riippuvainen ylimääräisestä luokasta. Samoin tällaiseen metodiin ei vaikuta luokan `Person` julkiseen rajapintaan tehtävät muutokset. Lorenzin ja Kiddin mielestä ylimääräiset parametrit lisäävät asiakkaan työmäärää [112], minkä vuoksi PPM kannustaa luokkapohjaisiin parametreihin.

Huomattava osa oliometriikoista keskittyy kuitenkin arvioimaan tunnettuja ja keskeisiä piirteitä, kuten kokoa, perintää, koheesiota ja kytkeytymistä. Kahta viimeisintä käsitellään alaluvuissa 2.3 ja 2.4. Seuraavassa luetellaan muutamia valittuja paradigman koko- ja perintämittoja.

Kokometriikat

Olioparadigman kokometriikat perustuvat yleensä luokkien piirteiden — metodien ja attribuuttien — lukumäärään. Esimerkiksi Lorenz ja Kidd [112] esittelevät kolme metodien ja kaksi attribuuttien kokomittaa. Piirteisiin perustuvien mittojen suosiota selittää niiden yksinkertainen automatisoitavuus. Esimerkiksi metriikkaohjelma FrontEndArt Monitorin [66] 19:stä Javan kokomitasta yhdeksän perustuu luokan piirteiden määrään. Suurin osa lopuista on LOC:n eri muunnelmia.

Metodien lukumäärä (engl. *Number of Methods*, NOM) [109] on yksinkertainen kokomitta, mutta Lorenz ja Kidd [112] määrittelevät siitä erillisiksi mittareiksi vielä julkisten instanssimetodien määrän (engl. *Number of Public Instance Methods*, PIM), kaikkien instanssimetodien määrän (engl. *Number of Instance Methods*, NIM) ja staattisten metodien määrän (engl. *Number of Class Methods*, NCM). Lanza ja Marinescu [107] esittelevät vielä mitan luokan aksessoreille (engl. *Number of Accessor Methods*, NOAM), joka laskee yhteen luokan `get-` ja `set`-metodit.

Attribuuteilla on myös osansa kokomitoissa, esimerkiksi Lorenzin ja Kiddin [112] kirjassa lasketaan instanssi- ja staattisten muuttujien määriä. Li ja Henry [109] määrittelivät yhdeksi kokomitoistaan piirteiden lukumäärää summan, eli kaikkien metodien ja attri-

buuttien yhteenlasketun määrän. Metriikkaohjelmat, esimerkiksi FrontEndArt Monitor [66] ja Project Analyzer [13], laskevat edellä mainittujen lisäksi myös mm. luokan kommenttirivien lukumäärän, funktioiden parametrien kokonaismäärän, luokan konstruktoreiden ja toteutettujen rajapintojen lukumäärän.

Todennäköisesti kaikki edellä nimetyt kokomitat vain heijastavat LOC:n arvoja, sillä mitä enemmän piirteitä löytyy, sitä enemmän ohjelmakoodia vaaditaan. Oletusta tukevia tuloksia löysivät Ronchetti kollegoineen [148] ja El Emam ym. [60]. He havaitsivat huomattavan korrelaation metodien ja lähdekoodirivien määrän välillä.

Se, että edellä mainitut mitat todennäköisesti vain toistavat LOC:n arvoja, ei tee niistä täysin käyttökelvottomia. Esimerkiksi julkisilla piirteillä voidaan myös arvioida luokan käytettävyyttä. Lanza ja Marinescu [107] käyttävät aksessorien lukumäärää osana huonosti kapseloitujen luokkien tunnistamiseen. Ronchetti ym. [148] ehdottivat laskentatapaa, millä suunnitteluvaiheen metodien määrästä voidaan arvioida ohjelman lopullista kokoa ja implementointiin vaadittavaa työmäärää.

Perintämetriikat

Useimmat periytymismitoista tarkastelevat luokan esivanhempien, jälkeläisten tai piirteiden muutosten määriä. Lorenz ja Kidd [112] esittelevät kaksi luokkien ja neljä metodien periytymisen metriikkaa. Luokkametriikoita ovat hierarkiataso (engl. *Hierarchy Nesting Level*, HNL) ja moniperintä (engl. *Multiple Inheritance*, MUI), joista ensimmäinen vastaa Chidamberin ja Kemererin DIT:ä. Jälkimmäinen määritellään nollaksi luokille, jotka eivät käytä moniperintää ja yhdeksi muille luokille. Lorenz ja Kidd pitävät usealta vanhemmalta perimistä huonona suunnitteluna ja haluavat määritellyllä metriikalla havaita moniperinnän käytön ohjelmistossa.

Chidamberin ja Kemererin NOC:stä innostuneena Lake ja Cook [104] määrittelivät luokille vielä vanhempien määrän (engl. *Number of Parents*, NOP) ja perijöiden määrän (engl. *Number of Descendents*, NOD), joista jälkimmäinen poiketen NOC:sta huomioi

kaikki perijät. Ensimmäinen soveltuu ainoastaan moniperintään, sillä se ilmaisee luokan vanhempien määrän. Tegarden, Sheetz ja Monarchi [156] lisäsivät vielä esi-isien määrän (engl. *Number of Ancestor*, NOA).

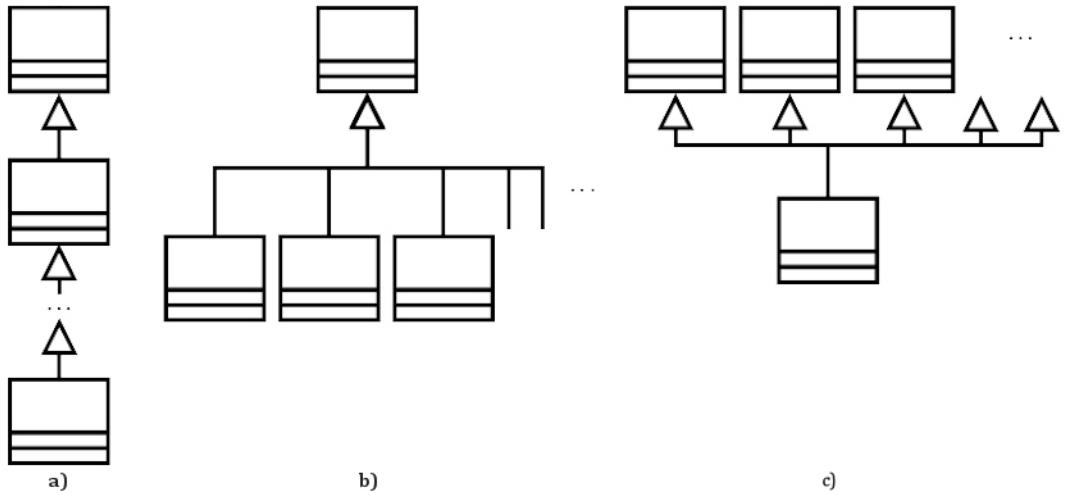
Lorenzin ja Kiddin [112] metoditason perintämetriikoista ylikirjoitettujen (engl. *Number of Methods Overridden*, NMO), perittyjen (engl. *Number of Methods Inherited*, NMI) ja lisättyjen metodien (engl. *Number of Methods Added*, NMA) määrät ovat varsin mitäänsanomattomia. Ainutlaatuisempaa lähestymistapaa edustaa heidän erikoistumisindeksinsä (engl. *Specialization Index*, SIX), joka kuvaa kuinka paljon luokka on erikoistuneempi kuin sen yliluokat, eli kuinka paljon luokka korvaa edeltäjiensä palveluita. Erikoistumisindeksi lasketaan kaavasta [112]:

$$\text{SIX} = \frac{\text{NMO} \times \text{HNL}}{\text{NOM}} \tag{2.20}$$

Ylikirjoittaviksi metodeiksi lasketaan ainoastaan ne, jotka eivät kutsu alkuperäistä toteutustaan. Lorenz ja Kidd [112] käyttävät yli 15 % erikoistumisastetta kynnysarvona, jonka ylittävien luokkien sijainti hierarkiassa pitäisi tarkistaa. Kynnysarvo vastaa perintäpuun ensimmäisellä tasolla yli kolmen metodin uudelleen määrittelemistä. [112]

SIX yrittää havaita perijöitä, jotka käyttävät hyväksi joitain yliluokan piirteitä, mutta eivät yritä erikoistaa suoraa alityyppiä. Lorenz ja Kidd [112] mieltävät tämän huonoksi suunnitteluksi. Esimerkki perinnän väärinkäytöstä on Javan luokkakirjaston vektorista peritty pino, jonka toteutus mahdollistaa alkioiden lisäämisen ja poistamisen tietorakenteen keskeltä. Tämä rikkoo räikeästi pinon toimintaperiaatteita. Perittäviä piirteitä käytetään hyväksi myös sovelluskehyksissä, joissa vaatimuksena on keskeisten metodien ylikirjoittaminen. Lorenz ja Kidd [112] kehottavat jättämään sovelluskehysten keskeiset metodit SIX:n laskennan ulkopuolelle.

Vastaavia erikoistumismittoja luokkakokonaisuuksille määritteli Henderson-Sellers, joka esitteli uudelleenkäyttö- (engl. *Reuse Ratio*, kaava 2.21) ja erikoistumissuhteet (engl.



Kuva 2.1: Kolme perinnän ääritapausta esitettynä UML:n luokkaakaaviona. [80]

Specialization Ratio, kaava 2.22) [80]:

$$U = \frac{\text{Yliluokkien määrä}}{\text{Luokkien määrä}} \tag{2.21}$$

$$S = \frac{\text{Aliluokkien määrä}}{\text{Yliluokkien määrä}} \tag{2.22}$$

Luokka on yliluokka, jos sillä on lapsi. Vastaavasti aliluokka perii eksplisiittisesti vähintään yhdeltä vanhemmalta. Uudelleenkäyttösuhteen arvo lähellä nollaa kertoo matalasta perintäpuusta (kuva 2.1b), jossa on huomattava määrä lehtiluokkia ja vähän perinnän hyväksikäyttöä. Lähellä yhtä oleva arvo ilmaisee lineaarista hierarkiaa (kuva 2.1a), jossa lehtiluokkia on vähän ja perittyjä piirteitä paljon.

Korkea erikoistumissuhde tulee järjestelmissä, joista löytyy paljon uudelleenkäyttöä perinnän kautta. [80] Lineaarisen hierarkian (kuva 2.1a) S lähenee yhtä ja matalan perintäpuun (kuva 2.1b) ääretöntä. Erikoistumissuhde voi olla myös yhtä pienempi moniperintää tukevissa järjestelmissä, esimerkiksi kuvan 2.1c järjestelmän S lähenee nollaa.

Käyttöohjeena kehotetaan välttämään U:n ja S:n lähellä 1 olevia arvoja, mitkä edustuvat huonona suunnitteluna pidettyä lineaarista perintää. [80] Näiden mittojen yleishyödyllisyyden voi kyseenalaistaa, sillä ainoastaan ääriarvot ovat mielenkiintoisia. Toisaalta ääriarvoja saadaan ainoastaan perinnän erikoistapauksia muistuttavista suunnitel-

mista, jolloin ratkaisu on tietoinen tai luokkarakennekaaviosta selvästi nähtävissä.

Briand, Daly, Porter ja Wüst [36] tarkastelivat kahdeksan opiskelijaprojektin virheraportit ja vertasivat tuloksia yhteentoista periytymismitan arvoon. He olettivat syvällä hierarkiassa olevien luokkien olevan virhealttiimpia kuin lähempänä juurta olevien. Tulokset tukivat näitä oletuksia, sillä luokkien virhealttius kasvoi mitä enemmän perittäviä piirteitä oli. Samoin virhealttius on pienempi niillä luokilla, joiden jälkeläisten määrä on suuri.

Briand ja kollegat [36] huomasivat myös Lorenzin ja Kiddin NMA:n, NMO:n ja SIX:in suurten arvojen korreloivan luokkien virhealttiuden kanssa. Lisättyjen metodien ja virheiden määrän riippuvuussuhde ei ole toisaalta yllätys, sillä mitä enemmän lähdekoodia luokan määrittelyyn lisätään, sitä todennäköisemmin virheiden määrä kasvaa. Myöhemmissä mittauksissaan Briand ym. [37] löysivät NMA:lle vahvan ja NMO:lle heikon korrelaation ohjelmakoon kanssa.

Briand, Wüst ja Lounis [37] toistivat opiskelijoiden projekteista tehdyt mittaukset ammattilaisten rakentamalla järjestelmällä. Yllättäen perintämetriikoista ainoastaan NMA ja NMO toimivat virhealttiuden mittareina. DIT toimi käänteisenä mittarina: järjestelmässä lähempänä juurta sijaitsevat luokat olivat virhealttiimpia kuin lehtiluokat. Briand ym. [37] arvelivat perintämittojen sopivan paremmin mittaamaan ohjelmoijien ymmärrystä perinnästä kuin järjestelmän laadukkuutta.

2.3 Koheesiometriikat ja luokittelukehys

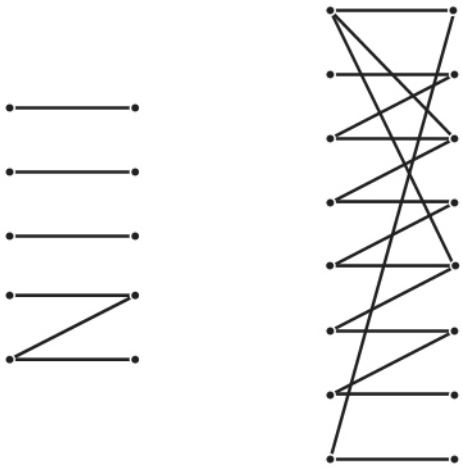
Coad ja Yourdon [51] määrittelivät luokan kiinnevoiman eli koheesion (engl. *Cohesion*) asteeksi, kuinka hyvin luokan osat toteuttava yhden, tarkoin määritellyn toiminnallisuuden. Kiinnevoiman voi myös mieltää tarkoittavan kuinka hyvin luokka toteuttaa semanttisesti mielekkään käsitteen [80], sillä luokan elementtien ei välttämättä tarvitse olla tiukasti toisiinsa sidottuja kiinnevoimaisissa luokissa. Riittää, että luokan osien tuottamat

palvelut ovat yhtenäisiä.

Yleisesti hyväksytyn käsityksen mukaan matalan koheesion luokat ovat virhealttiimpia kuin korkean koheesion. Samoin kiinnevoimaiset luokat ovat helpompia ylläpitää, testata, kehittää ja käyttää uudelleen kuin alhaisen koheesion luokat. [32, 172] Luokan kiinnevoimaisuus ei kuitenkaan ole selkeästi mitattava suure, ja metriikat perustuvatkin suunnitteluohjenuoran karkeaan arvioimiseen. Useimmat koheesiomitoista laskevat luokan yhtenäisyyttä arvioimalla metodien käyttämien attribuuttien määriä. Mäkelä ja Leppänen [134] kritisoivat lähestymistapaa epärealistiseksi, sillä isot luokat voivat olla kiinnevoimaisia, vaikka jokainen metodi ei käyttäisi jokaista attribuuttia.

Chidamberin ja Kemererin LCOM:in jälkeen on esitetty lukuisia ehdotuksia kiinnevoiman mittariksi. Briand kollegoineen [32] tarkasteli useimpia koheesiometriikoita kehittäessään niille luokittelukehystä. He numeroivat ehdotetut metodien koheesion puutteen metriikat aloittaen Chidamberin ja Kemererin [46] alkuperäisestä määritelmästä, joka tunnetaan nyt LCOM1:nä. Uudempi, tässä opinnäytteessä esitetty versio, sai nimen LCOM2. Lin ja Henryn [108] ehdotus on LCOM3 ja Hitzin ja Montazerin [81] versio on LCOM4. Henderson-Sellersin [80] puhtaasti attribuuttien käyttöön perustuva näkemys nimettiin LCOM5:ksi.

Ehdotetut LCOM-metriikat ovat periaatteiltaan hyvin samankaltaisia ja myös kaatuvat samoissa ongelmissa. Niille vaikeuksia tuottavat esimerkiksi aksessorimetodien käsittely. Tässä opinnäytteessä esitellään LCOM:in seuraajista vain metriikkaohjelmien, kuten Project Analyzer [13], suosittelemaa LCOM4:ä. Toisenlaisen lähestymistavan luokan yhtenäisyyden mittaamiseen tuovat Biemanin ja Kangin [29] TCC ja LCC, sekä Mäkelän ja Leppäsen [136] LCIC. Luvun lopussa esitellään Briandin, Dalyn ja Wüstin [32] luokittelukehys koheesiometriikoille, sekä empiiristen tutkimusten tuloksia.



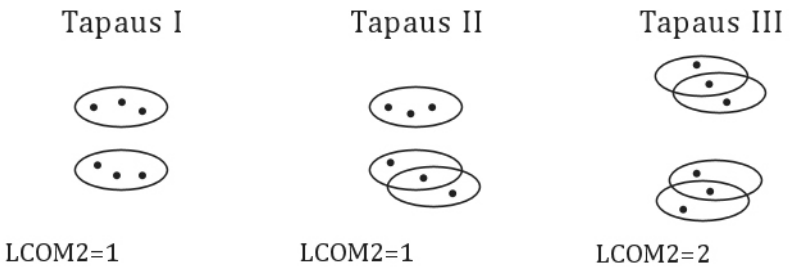
Kuva 2.2: Kaksi luokkaa graafina, joilla molemmilla LCOM2 on 8, vaikka oikeanpuoleinen on selvästi kiinnevoimaisempi. Graafin pisteet ovat metodeja sekä attribuutteja ja kaaret käyttörelaatioita näiden välillä. [80]

2.3.1 LCOM4 ja graafin kytkeytyneisyyden apumitta

Ensimmäisiä koheesion puutteen mittareita kritisoitiin laajasti niiden epäjohdonmukaisuuksista. Esimerkiksi Hendersson-Sellers [80] antaa vastaesimerkin (kuva 2.2), jossa selvästi koheesioton luokka ja yhtenäinen luokka saavat molemmat LCOM2:lta arvon 8. Hitz ja Montazeri [81] huomasivat tapauksia, jossa vähemmän kiinnevoimainen luokka saa LCOM2:lta kiinnevoimaisempaa luokkaa paremman tuloksen. Kuvassa 2.3 on kolme tapausta, joissa intuitiivisesti toisen tapauksen koheesio olisi suurempi kuin ensimmäisen, vaikka LCOM2 on sama kummallekin. Selvästi myös tapaus III on kahta muuta koheesiivisempi, mutta LCOM2:lla sen kiinnevoiman arvo on huonoin.

Chidamberin ja Kemererin määritelmässä ei huomioitu konstruktoreiden ja aksessorimetodien vaikutusta. Konstruktorit alustavat kaikki luokan attribuutit, näin lisäten keinotekoisesti samoja attribuutteja käsittelevien metodiparien määrää. [29] `Get-` ja `set-`metodit toimivat päinvastaisella tavalla: ne käsittelevät vain yksittäisiä muuttujia, vähentäen luokan LCOM2:den arvoa. [32]

Luokka voi myös sisäisessä toteutuksessaan käyttää aksessorimetodeja suorien attri-



Kuva 2.3: Kolmen luokan kiinnevoimaisuus Venn-diagrammilla esitettynä. Kuvassa ellipsi ja pisteet kuvaavat metodeja ja attribuuteja. [81]

buuttiviittausten sijasta, mikä vähentää yhteisiä attribuutteja käyttävien metodiparien lukumäärää. [81] Näiden puutteiden valossa Hitz ja Montazeri [81] ehdottivat alkuperäisten LCOM:ien määritelmän korvaajaksi omaa, graafiteoriaan perustuvaa, laskentatapaansa.

LCOM4:n esittämiseksi pitää määritellä suuntaamaton graafi $G_c = (V_c, E_c)$, missä pistejoukkona on luokan c metodien joukko V_c. Graafin kaarina on metodiparien $<m, n>$ joukko, jotka käyttävät samaa muuttujaa tai toista metodiparin metodeista. Muodollisesti ilmaistuna kaarijoukko on [81]:

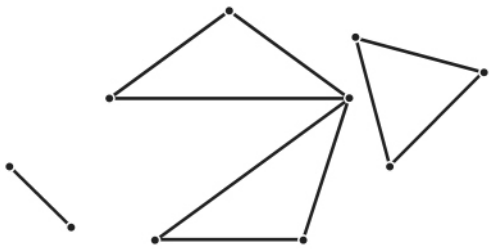
$$E_c = \{<m,n> \in V_c \times V_c \mid \left[\exists\, i \in I_c : (m \text{ käyttää } i\text{:tä}) \wedge (n \text{ käyttää } i\text{:tä})\right] \vee$$
$$(m \text{ kutsuu } n\text{:ää}) \vee (n \text{ kutsuu } m\text{:ää}) \}$$

Missä I_c on luokan instanssimuuttujien joukko. Graafin pisteet yhdistetään kaarella, jos ne käyttävät samaa instanssimuuttujaa. Jos metodi — eli graafin piste — kutsuu luokan toista metodia, näiden välille lisätään kaari.

Graafin avulla lausuttuna Hitzin ja Montazerin LCOM4 on [81]:

$$\text{LCOM4}(c) = \text{Graafin } G_c \text{ yhtenäisten komponenttien määrä} \tag{2.23}$$

Esimerkiksi kuvan 2.4 graafissa on löydettävissä kolme erillistä komponenttia. Tämän luokan LCOM4 on 3, ja sen selvästi voisi jakaa kolmeen pienempään luokkaan. Yleisesti yli kahden arvoja saavat luokat pitäisi jakaa mitan osoittamaan määrään pienempiä luokkia. Jos LCOM4 saa arvon nolla, luokassa ei ole metodeja. Arvolla 1 graafissa on vain yksi yhtenäinen komponentti, ja tämä kuvastaa kiinnevoimaista luokkaa. [13]



Kuva 2.4: Kolmiosainen graafi, jolle LCOM4 on 3.

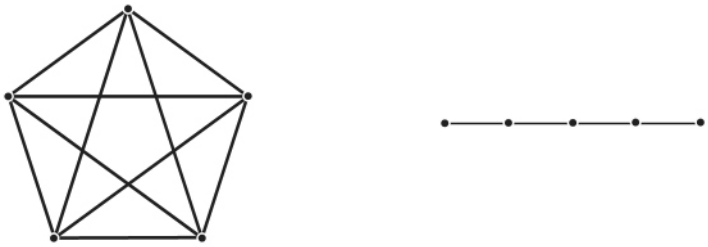
Uusi määritelmä poistaa aksessorimetodien ongelmat, mutta ei huomioi konstruktorin vaikutusta. Ratkaisuna tähän Briand kollegoineen [32] suosittelee konstruktorien jättämistä tarkastelun ulkopuolella. He myös kritisoivat mittaa — ja kaikkia LCOM:eja — normalisoinnin puutteesta, sillä LCOM4 voi saada arvoja nollasta äärettömään. Briand ym. [32] huomauttavat myös, etteivät Hitz ja Montazeri ota kantaa periytyvien piirteiden käsittelyyn.

Etzkorn kollegoineen [61] vertasi tunnettuja koheesiomittoja vasten asiantuntijaryhmän luokille antamia koheesioarvoja. Selvityksessä LCOM4 vastasi koheesion puutteen mitoista parhaiten arvioijien käsitystä kiinnevoimasta. Samassa tutkimuksessa löydettiin, yllätyksettömästi, huomattavaa korrelaatiota LCOM1–5 mittojen välille.

Hitz ja Montazeri [81] huomasivat mittansa puutteelliseksi graafin muodostuessa yhdestä komponentista. Kuvassa 2.5 on kahden luokan graafit, jotka ovat LCOM4:n mielestä hyviä. Vasemmanpuoleinen täydellinen graafi on kuitenkin selvästi oikeanpuoleista kiinnevoimaisempi. Tämän vuoksi Hitz ja Montazeri määrittelivät yhden komponentin graafeille apumitan, joka tarkastelee kuinka lähellä täydellisesti kytkeytynyttä graafia luokka on. Briand ym. [32] nimesivät mitan Co:ksi (engl. *Connectivity*). Kaavana Co on [81]:

$$\text{Co} = 2 \times \frac{|E_c| - (|V_c| - 1)}{(|V_c| - 1)(|V_c| - 2)} \tag{2.24}$$

Co:n arvot kuvautuvat välille $[0, 1]$, missä yläraja 1 saadaan sijoittamalla kaavaan täydellisen graafin pistejoukon koko $|E_c| = \frac{|V_c|(|V_c|-1)}{2}$. Vastaavasti alaraja tulee kuvan 2.5 oikeanpuoleisesta graafista, missä on muodostettu yhden komponentin graafi vähimmäismäärä-



Kuva 2.5: Yhden komponentin graafien ääritapaukset kun $|V| = 5$. Vasemman puoleinen huomattavasti kiinnevoimaisempi, mutta oikean puoleisen hajoittaminen useaksi luokaksi saattaa olla vaikeaa. [81]

lä kaaria. Sijoittamalla tämän graafin kaarijoukon koko $|E_c| = |V_c| - 1$ kaavaan 2.24 saadaan tulokseksi 0.

Apumitta on laskettavissa vain luokille, joilla on enemmän kuin kaksi metodia. Toisin kuin LCOM4:n kohdalla, Hitz ja Montazeri [81] eivät tarjonneet toimintaohjeita Co:lle. Yhtenäisen kytkösgraafin hajottaminen useammaksi luokaksi ei välttämättä ole mielekästä, sillä syntyneiden luokkien välille saattaa muodostua hyvin vahvoja kytköksiä.

2.3.2 TCC ja LCC

Bieman ja Kang [29] pitivät alkuperäisiä LCOM:eja riittämättöminä havaitsemaan hienovaraisia eroja koheesiossa. He esittelivät mitat luokan tiukalle (engl. *Tight Class Cohesion*, TCC) ja löyhälle (engl. *Loose Class Cohesion*, LCC) kiinnevoimalle, joiden pitäisi mittareina olla herkkiä luokan koheesion muutoksille. Kuten LCOM:ien, myös Biemanin ja Kangin lähestymistapa perustuu metodien yhteisten attribuuttien käyttämiseen. Toisin kuin Chidamberin ja Kemerin metriikoissa, Bieman ja Kang huomioivat myös metodien välisiä riippuvuuksia.

TCC mittaa luokan suoria ja LCC epäsuoria kytköksiä. Luokan metodit ovat suoraan kytkeytyneitä, jos ne käyttävät vähintään yhtä yhteistä attribuuttia. Metodit ovat epäsuorasti kytkeytyneet, jos löytyy jono metodipareja, jotka ovat suoraan kytkeytyneitä.

Muodollisesti ilmaistuna metodit m_1 ja m_n ovat epäsuorasti kytkeytyneet kun:

$$m_1 \, \delta' \, m_n \iff m_1 \, \delta \, m_2, \; m_2 \, \delta \, m_3, \; \ldots, \; m_{n-1} \, \delta \, m_n$$

Missä δ' kuvaa epäsuoraa ja δ suoraa kytköstä. Bieman ja Kang rajaavat tarkastelunsa ainoastaan luokan näkyville metodeille. [29]

TCC ja LCC ovat muodollisesti annettuna [29]:

$$\text{TCC}(c) = \frac{\text{NDC}(c)}{\text{NP}(c)} \tag{2.25}$$

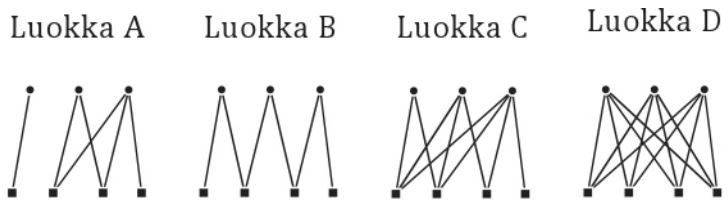
$$\text{LCC}(c) = \frac{\text{NDC}(c) + \text{NIC}(c)}{\text{NP}(c)} \tag{2.26}$$

Missä NDC(c) on luokan c suoraan ja NIC(c) epäsuoraan kytkeytyneiden metodiparien lukumäärä. Luokan suorien ja epäsuorien kytkösten maksimi on NP(c), joka on n:llä metodilla $\frac{n \times (n-1)}{2}$. Koska LCC:n määritelmä sisältää myös suorat kytkökset, on LCC aina vähintään yhtä suuri kuin TCC. Molemmat mitat ovat normalisoitu välille $[0, 1]$, missä yksi kuvaa luokan täydellistä ja nolla olematonta yhtenäisyyttä. [29]

Metriikan määrittelijät havaitsivat konstruktorien luovan näennäisiä kytkeytyneitä metodeja ja jättivät alustajat TCC:n ja LCC:n määritelmien ulkopuolelle. [29] Aksessorimetodien ongelmat on kierretty huomioimalla epäsuorat kytkennät LCC:n määritelmässä.

Bieman ja Kang [29] huomioivat perinnän vaikutuksen luokan koheesiolle ja antoivat kolme toimintamallia: Ensimmäisessä vaihtoehdossa kaikki perityt piirteet lasketaan mukaan. Toisessa vaihtoehdossa huomioidaan vain luokassa itsessään määritellyt piirteet. Kolmannessa vaihtoehdossa perityt attribuutit sisällytetään, mutta ei perittyjä metodeja. Briand, Daly ja Wüst [32] pitävät ainoastaan kahta ensimmäistä vaihtoehtoa järkevinä ja ihmettelevät, miksi Bieman ja Kang eivät perustelleet syitä kolmannen vaihtoehdon mukaan ottamisesta.

Etzkorn ym. [61] eivät löytäneet mittauksissaan huomattavaa riippuvuutta Biemanin ja Kangin metriikoiden ja LCOM:ien variaatioiden välille. He ehdottavat selitykseksi LCC:n ja TCC:n mittaavan erilaista koheesiota kuin LCOM:it, vaikka mitat perustavat



Kuva 2.6: Luokkien viittausgraafit, joissa neliö kuvaa metodia ja ympyrä attribuuttia. [43]

Taulukko 2.2: Kuvan 2.6 luokkien koheesioarvoja.

	Luokka A	Luokka B	Luokka C	Luokka D
LCOM2	0	0	0	0
LCOM4	2	1	1	1
Co	†	0	1	1
TCC	$\frac{1}{2}$	$\frac{1}{2}$	1	1
LCC	$\frac{1}{2}$	1	1	1

† Co ei ole määritelty luokille, joilla ei ole yhtenäistä graafia.
Chae ym. [43]

metodiparien attribuuttien käyttöön. Etzkorn kollegoineen pitää LCC:tä täydellisimpänä kiinnevoiman mittarina Briandin ym. [32] listaamista koheesiometriikoista.

Chae, Kwon ja Bae [43] kritisoivat Biemanin ja Kangin metriikoita edelleen liian epätarkoiksi hienovaraisten koheesioerojen mittaajiksi. He antoivat neljän luokan vastaesimerkin (kuva 2.6), jossa TCC saa kahdessa ensimmäisessä ja kahdessa viimeisessä luokassa saman arvon. Samoin LCC saa saman arvon luokissa B, C ja D (taulukko 2.2), sillä mitta saavuttaa maksimiarvon kun graafissa on vain yksi komponentti. Intuitiivisesti luokat eivät kuitenkaan ole yhtä kiinnevoimaisia. Chae ja kollegat [43] huomauttivat LCC:n myös saavan epäsuorilla kytköksillä helposti maksimiarvoja, mikä ei auta huomioimaan eroja esimerkiksi luokkien B, C ja D koheesioissa.

Bieman ja Kang [29] eivät ottaneet kantaa pitäisikö TCC:tä ja LCC:tä käyttää erillään vai yhdessä. Etzkorn ym. [61] havaitsivat — yllätyksettömästi — näiden metriikoiden kor-

reloivan keskenään ja suosittivat vain LCC:n käyttämistä, ja myös muut tutkijat [135, 172] ovat keskittyneet vaan toiseen esitetyistä metriikoista. Kuitenkin Chae ym. [43] huomasivat LCC:n saavuttavan helposti maksimiarvon, milloin TCC:n tulos saattaa osoittautua mielenkiintoisemmaksi. Biemanin ja Kangin metriikoista saa täyden hyödyn vasta käyttämällä niitä rinnakkain arvioimaan luokan kiinnevoimaisuutta.

2.3.3 Koheesion puute asiakkaissa

Koheesio kuvaa myös luokan semanttisen käsitteen yhtenäisyyttä, mutta esitellyt mitat perustuvat metodiparien yhteisiin instanssimuuttujiin ja luokan sisäisten kytkösten määrään. Ne eivät selvästi mittaa, kuinka hyvin luokan käsite on muodostettu tai kuinka yhtenäistä käsitettä luokka mallintaa, vaan keskittyvät arvioimaan luokan implementaation yhtenäisyyttä. Ongelma puhtaasti luokan sisäiseen näkymään keskittyvässä maailmankuvassa on mitan arvon liiallinen riippuvuus toteutuksesta.

Esimerkiksi luokan `Rectangle` koheesiomittojen tulokseen vaikuttaa onko suorakulmio mallinnettu kahdeksalla yksittäisellä koordinaatilla vai kahdella sivun pituudella ja yhdellä koordinaattiparilla. [134] Jos koheesioarvoja käytetään luokkien hyvyyden arvosteluperusteena, voi ohjelmoija sijoittaa kaikki attribuutit yhteen tietorakenteeseen. Tämä kasvattaa luokan kiinnevoimaa keinotekoisesti, sillä kaikki metodit käsittelisivät vain yhtä ja samaa instanssimuuttujaa. Tällöin luokka näyttää keinotekoisesti paremmalta.

Kuvan 2.6 luokka `A` on uudempien koheesiomittojen arvoilla huonosti muodostettu. Esimerkiksi LCOM4 suosittelee sen jakamista kahteen erilliseen osaan. Luokka saattaa silti muodostaa semanttisesti järkevän kokonaisuuden, jonka pilkkominen kahteen osaan ei ole mielekästä. Erityisesti jos luokan asiakkaat käyttävät yleisesti graafin molempia osia, ei pienempien osaluokkien luominen ole tarkoituksenmukaista. Mäkelä ja Leppänen [133] paheksuvatkin refaktoroinnin, eli ohjelman rakenteen siistimisen jälkikäteen, perustamisen paikallisten koheesiomittojen antamiin tuloksiin.

Mäkelä ja Leppänen [136] tarjoavat vaihtoehtoista lähestymistapaa määrittelemällä asiakas- ja kokonaisnäkymän tarkasteltavaan luokkaan perinteisen sisäisen näkymän lisäksi. Listatut koheesiometriikat edustavat juuri sisäistä näkymää, sillä niiden laskennassa huomioidaan ainoastaan luokan toteutus. Asiakkaan käyttämät piirteet muodostavat yksittäisen asiakasnäkymän ja jokaisella asiakkaalla on oma näkymänsä luokasta. Kokonaisnäkymä on kahden edellä kuvatun näkökulman yhdistelmä.

Asiakaspohjaiset metriikat perustuvat asiakas- tai kokonaisnäkymään luokasta. [136] Mäkelä ja Leppänen [133, 134] perustelevat kontekstin tärkeyttä luokan käytön mielekkyydessä ja yhtenäisyydessä, minkä vuoksi he [136] määrittelivät asiakaspohjaisen kiinnevoimametriikan: koheesion puutteen asiakkaissa (engl. *Lack of Cohesion in Client*, LCIC). Mitta kuvaa, kuinka yhtenäisesti luokan asiakkaat käyttävät tarjottuja palveluita. Perustavana oletuksena on pitää luokkaa yhtenäisesti muodostettuna, jos asiakkaat käyttävät yhtenäisesti sen koko tietosisältöä. [136]

LCIC:n laskennassa huomioidaan paikallisesti määriteltyjen piirteiden lisäksi perityt metodit ja attribuutit. [136] Muodollisesti luokan s koheesion puute asiakkaissa lasketaan kaavasta [136]:

$$\text{LCIC}(s) = \frac{\sum_{c \in C,\, Uses(c,s)} \text{LCIC}(c, s)}{|clients(s)|} \tag{2.27}$$

Missä C on kaikkien luokkien joukko ja $clients(s)$ on luokan s asiakkaiden joukko. LCIC on siis asiakaskohtaisen koheesion puutteen keskiarvo. Asiakaskohtainen LCIC saadaan kaavasta [136]:

$$\text{LCIC}(c, s) = 1 - \frac{|\{f \in s \,|\, Uses(c, f)\}|}{|\{f \in s \,|\, accessible_c(s)\}|}$$

Missä asiakasluokan suoraan tai palvelujen kautta käyttämien piirteiden määrä on jaettu asiakkaan näkemien piirteiden kokonaismäärällä, sillä käytetty rajapinta ei välttämättä tarjoa pääsyä kaikkiin palveluihin. Luokan LCIC on normalisoitu välille $[0, 1]$, missä yksi tarkoittaa täydellistä kiinnevoiman puuttumista. Jos luokalla ei ole asiakkaita tai instanssimuuttujia, sen LCIC on nolla. Jos luokalla on instanssimuuttujia, mutta ei asiakkaita, mitan arvo on yksi. [136]

LCIC yrittää olla ensimmäinen puhdas semanttisen käsitteen mittari, mutta on valitettavan riippuvainen luokan tietosisällön toteutuksesta. Esimerkiksi tietosisällön yhteen taulukkoon keskittänyt luokka saa parempia arvoja kuin perinteisemmin toteutetut. Tosin LCIC:n huomioidessa myös käytetyt metodit, ei taulukolla tehty harhautus ole yhtä merkittävä kuin sisäiseen katsantoon perustuvilla mitoilla. Toisenlainen, häiritsevämpi ongelma on huonosti toteutetut asiakkaat, sillä ne saattavat käyttää luokkaa toisin kuin on tarkoitettu. [136] Näin kiinnevoimaisesti toteutettu luokka voi saada *"väärän positiivisen"*, huonon koheesioarvon ilman todellista ongelmaa.

Mäkelä ja Leppänen [136] mittasivat LCIC:n arvoja kolmesta suuresta Java-ohjelmasta. Projektien koko vaihteli 94–161 KLOC:iin, ja jokaisessa niistä oli satoja luokkia. Kokeellisessa tutkimuksessa he onnistuivat havaitsemaan LCIC:llä luokkia, jotka selvästi tarvitsisivat refaktorointia. Ongelmaluokat esimerkiksi palvelivat useassa roolissa, kaipasivat piirteiden abstraktointia rajapinnan taakse tai toimivat välittäjänä kutsuketjuissa.

Mittauksissa Mäkelä ja Leppänen [136] löysivät tarkastellusta 1 913 luokasta vain 96 luokkaa, joilla ei ollut asiakkaita. Näiden lisäksi 151:llä luokalla oli vain yksi tai kaksi asiakasta. LCIC:lle ongelmallisia ovat luokat, joilla on vähän asiakkaita. Yhden tai kahden asiakkaan luokilta ei välttämättä saada tarpeeksi kattavaa kuvaa niiden käytöstä. Arvioituaan mittauksissa löydettyjä yhden tai kahden asiakkaan luokkia, Mäkelä ja Leppänen pitävät laskettuja LCIC:n arvoja luotettavina myös näille.

LCIC:n määrittelijät osoittivat mitan olevan hyödyllinen refaktorointitarpeen löytäjä isoissa ohjelmissa. Luotettavaan tulokseen vaaditaan kuitenkin useampi käyttäjä luokalle, mikä saattaa rajoittaa LCIC:n sopivuutta pienissä projekteissa, joista löytyy vain muutamia asiakkaita. LCIC ei myöskään sovellu luokille, joille ei tarkastelualueelta löydetä asiakkaita. Esimerkiksi kirjastoluokat pitäisi arvioida sovelmien kanssa sopivassa asiayhteydessä.

Mäkelä ja Leppänen [136] huomauttavat, ettei LCIC:n ole tarkoitus toimia ainoana koheesiomittana. Sisäiseen näkymään perustuvia koheesiomittoja tarvitaan luokkien kans-

Taulukko 2.3: Koheesiokehys ja esimerkkimittojen luokittelu.

	Koheesion kriteeri	Määrittely-alue	Epäsuorat kytkennät	Perintä	Erikois-metodit
LCOM2	yhteiset attribuutit	Luokka	ei	ei†	ei aksessoreita†
LCOM4	yhteiset attribuutit, metodien käyttö	Luokka	kyllä	ei†	ei konstrukt.†
TCC	kytkeytyneet metodit	Luokka	ei	on‡	ei konstrukt.
LCC	kytkeytyneet metodit	Luokka	kyllä	on‡	ei konstrukt.
LCIC	asiakasnäkymien piirteiden käyttöaste	Luokka	kyllä	kaikki	sisällytä

† Briandin ym. [36] suosituksia.
‡ Bieman ja Kang [29] tarjoavat kolme eri vaihtoehtoa.

sa, joilla ei ole yhtään asiakasta. On tutkittava, millaisia ongelmia ohjelmistomitat voivat löytää, ja millaisia virheitä metriikoiden yhdistelmillä voidaan havainnoida.

2.3.4 Koheesiometriikoiden luokittelukehys

Briand, Daly ja Wüst [32] loivat koheesiometriikoille kehyksen, jonka tavoitteena on helpottaa uusien mittojen määrittelemistä, validointia ja mittojen keskinäistä vertailua. Luokittelukehys tarjoaa viisi kriteeriä, joiden kohtelu pitäisi huomioida koheesiometriikoiden määrittelyssä. Teoriassa kehys auttaa näin myös käyttäjiä valitsemaan sopivan mitan tarpeisiinsa. Tässä opinnäytteessä käsitellyt koheesiomitat on esitetty kehyksen kategorioissa taulukossa 2.3. Briandin ym. kehyksen viisi kohtaa ovat [32]:

Koheesion kriteeri. Mikä yhdistää luokan piirteitä? Toisin sanottuna, mikä tekee luokasta yhtenäisen? Esimerkiksi alkuperäiset LCOM:it perustuvat metodiparien yhteisiin attribuutteihin, LCIC taas asiakasnäkymien käyttämien piirteiden määrään.

Määrittelyalue. Millä abstraktiotasolla koheesiota tarkastellaan? Briand ym. [32] huo-

mioivat viisi tasoa: attribuutti, metodi, luokka, luokkajoukko ja järjestelmä. Suurin osa — ja kaikki tässä opinnäytteessä — esitellyistä koheesiomitoista on määritelty vain luokkatasolle.

Epäsuorat kytkennät. Huomioidaanko laskennassa elementtien epäsuorat vai vain suorat kytkeytymiset? LCOM2 ja TCC huomioivat ainoastaan suorat kytkökset, muut esitellyistä tarkastelevat myös epäsuoria.

Periytyvät piirteet. Huomioidaanko laskennassa periytyneet piirteet ja miten? Periytyvistä piirteistä voidaan sisällyttää pelkät attribuutit, kaikki tai ei mitään. Bieman ja Kang [29] jättivät valinnan käyttäjälle. Mäkelä ja Leppänen [136] taas sisällyttävät tarkasteluunsa kaikki.

Erikoismetodien kohtelu. Miten laskennassa huomioidaan erikoismetodit kuten aksessorit, konstruktorit, destruktorit ja delegaatit? Aksessorit voidaan joko sisällyttää, jättää huomioimatta tai tulkita suoriksi viittauksiksi vastaaviin attribuutteihin. Konstruktorit ja destruktorit voidaan joko sisällyttää tarkasteluun tai jättää kokonaan huomioimatta. Chae ym. [43] huomasivat delegaattien toimivan kuin aksessoreiden; ne vähentävät keinotekoisesti luokan yhtenäisyyttä. Delegaattimetodit toimivat palveluiden välittäjinä ja näin käsittelevät vain yksittäistä attribuuttia.

2.3.5 Kritiikkiä ja empiirisiä tuloksia

Briand, Daly, Porter ja Wüst [36] kävivät läpi kahdeksan opiskelijaprojektin virheraportit, etsien korrelaatiota Briandin ym. [32] luettelemien koheesiomittojen ja virheiden väliltä. He olettivat koheesittomien luokkien olevan huonosti muodostettuja ja näin virhealttiimpia kuin muut. Tilastollisessa analyysissaan he löysivät vain heikkoja todisteita hypoteesin tueksi.

Briand, Wüst ja Lounis [37] toistivat mittauksen ohjelmistosuunnittelun ammattilaisten rakentamalla järjestelmällä, ja tulokset olivat samanlaisia kuin opiskelijoiden projek-

teissa. Abudakarin, AlGhamdin ja Ahmedin [11] avoimen lähdekoodin ohjelmien mittauksissa ei löydetty merkitseviä riippuvuuksia luokan virhemäärän ja koheesioarvojen välille. LCIC:n yhteyttä ulkoisiin laatutekijöihin ei ole vielä tutkittu.

Briandin ja kollegoiden [36] koheesio-oletus vastaa yleistä uskomusta kiinnevoimattomista luokista, miksi on yllättävää, ettei korrelaatiota virheisiin löytynyt. Selitykseksi he tarjoavat kahta huomiota: puutteellinen ymmärrys mitattavasta käsitteestä ja semanttisen kokonaisuuden arvioimisen vaikeus syntaktisella analyysilla. Jälkimmäistä huomiota tukee myös Henderson-Sellers [80], joka kritisoi jo alkuperäisiä LCOM:eja pyrkimyksestä mitata ainoastaan rakenteellista yhtenäisyyttä. Hänen mukaansa perinteiset koheesiometriikat eivät huomioi luokan muodostaman semanttisen kokonaisuuden järkevyyttä.

Uusia koheesiomittoja kehitetään kiivaasti, mutta ne keskittyvät usein vain arvioimaan luokan implementaatiota. Esimerkiksi Chae ym. [43] tavoittelivat täydellistä kiinnevoimametriikkaa, joka havaitsisi pienetkin erot luokkien yhtenäisyyden välillä. He määrittelivät koheesiomitan luokan jäsenten kytkeytyneisyydelle (engl. *Cohesion Based on Member Connectivity*, CBMC), joka antaa luokalle sitä huonomman arvon mitä helpommin sen viittausgraafi on jaettavissa pienempiin osiin. CBMC ei edes yritä mitata luokan käsitteen yhtenäisyyttä, vaan keskittyy puhtaasti luokan toteutuksen kritisointiin. Mitan rekursiivinen laskenta on myös huomattavan vaativa operaatio — jota voi tosin käsin nopeuttaa [67] — mikä vähentää sen sopivuutta käytännön ohjelmistokehitykseen.

Woo kollegoineen [172] valitsi toisenlaisen lähestymistavan ja uudisti keskeisten koheesiomittojen määritelmiä erottamalla attribuuttien käytön luku- ja kirjoitusinteraktioihin. Woo ym. perustelevat attribuutin kirjoittamisen lukemista kiinnevoimaisemmaksi operaatioksi, mitä ei ole perinteisissä attribuuttien yhteiskäyttöön perustuvissa metriikoissa huomioitu. He väittävät omien mittaustensa perusteella uudelleen määriteltyjen metriikoiden toimivan alkuperäisiä parempina muutosherkkyyden ennustajina. Ilmoitetut tilastollisen analyysin erot eivät tosin olleet kovin huomattavia ja osittain puutteellisesti raportoituja.

Chaen ym. [43] ja Woon ym. [172] metriikoissa täydellisen koheesion saavuttaminen on äärimmäisen vaikeaa. Silti on epäselvää, kuinka hyödyllistä suunnittelua edustaa luokka, jossa kaikki metodit lukevat ja kirjoittavat kaikkia attribuutteja. [32] Mäkelä ja Leppänen [132, 135, 136] sekä Briand ym. [37] löysivät tilastollisesti merkitsevän, mutta heikon korrelaation luokan koon, erityisesti instanssimuuttujien määrän, ja koheesiomittojen välille. Tilastollinen yhteys on helposti ymmärrettävä, sillä isojen luokkien on vaikea saavuttaa täydellistä koheesiota, jossa kaikki metodit käyttäisivät kaikkia attribuutteja. Ne antavatkin helpommin tulokseksi väärän positiivisen.

El Emam, Benlarbi, Goel ja Rai [60] saivat vielä kriittisempiä tuloksia tutkimuksessaan, jossa yksikään koetelluista mitoista ei ollut tilastollisesti merkitsevä ohjelmakoon vaikutuksen poistamisen jälkeen. He myös kyseenalaistivat aiempien tutkimusten tulokset, joissa koon vaikutusta ei oltu huomioitu. Ainakin Mäkelä ja Leppänen [135] ovat ehdottaneet ratkaisua ohjelmakoon vaikutuksen poistamiseksi. Heidän mallissaan mitan arvoa verrataan vasten attribuuttimääräkohtaiseen profiiliin luokan koon vaikutuksen poistamiseksi.

Luokan yhtenäisyyttä pidetään yleisesti keskeisenä olio-ohjelmoinnin tavoitteena, ja koheesiomittareita arvioidaan ja kehitetään raivokkaasti. Silti empiirisissä tutkimuksissa niiden hyödyllisyydestä virheiden ennustajina ei ole löytynyt näyttöä. Briand ym. [32, 33] painottavat toistuvasti metriikan olevan mielenkiintoinen ainoastaan, jos sillä voidaan osoittaan olevan yhteys johonkin järjestelmän laatutekijään. He ehdottavat metriikkasuunnittelun ensimmäiseksi askeleeksi ulkoisen ominaisuuden määrittämisen, mitä mitta yrittää mallintaa. Samoin he suosittavat mitan unohtamista, jos sillä ei voida osoittaa olevan yhteyttä ulkoisiin laatutekijöihin.

Mäkelällä ja Leppäsellä [136] oli toisenlainen lähestymistapa: he käyttivät LCIC:tä osoittamaan refaktorointia tarvitsevia luokkia. Refaktoroinnin tarkoituksena on parantaa ohjelmakoodin laatua [136], ymmärrettävyyttä [64] ja tuottavuutta [137]. Ulkoisten laatutekijöiden ja refaktoroinnin yhteyttä ei ole selvästi osoitettu [137], mutta lähestymistapa

saattaa sopia paremmin koheesiomitoille, jotka ovat toistuvasti epäonnistuneet ulkoisten laatutekijöiden indikaattoreina.

2.4 Kytkeytymisen metriikat ja luokittelukehys

Hakonen [75] määritteli olio-ohjelmoinnin koostuvan tietotyyppien lisäksi niiden välisestä vuorovaikutuksista. Luokkien ja olioiden välistä yhteistoimintaa tarvitaan, jotta järjestelmä voidaan rakentaa moduuleihin jaetuista toiminnallisuuksista. Toisaalta luokkien väliset kytkökset vaikuttavat myös järjestelmän ylläpitämiseen ja testattavuuteen, sillä mitä enemmän osat ovat kytkeytyneet toisiinsa, sitä vaikeampi niitä on kehittää ja testata erillisinä. Vuorovaikutuksen tärkeyttä kuvaa se, että suurin osa alaluvussa 3.3 esiteltävistä suunnittelumalleista keskittyy juuri olioiden väliseen yhteistoiminnan kuvaamiseen.

Kytkeytyminen (engl. *Coupling*) on koheesion ohessa tärkeä ohjelmistosuunnittelun vuorovaikutuksen ohjenuora. Kun koheesio tarkastelee luokan sisäistä yhtenäisyyttä ja yhteistoimintaa, kytkeyminen keskittyy luokkien ulkoisiin suhteisiin. Coadin ja Yourdonin klassisen määritelmän mukaan kytkeytyminen on *"olioperustaisen suunnittelun osien välinen yhteen kytkeytyneisyys"*. [51, s. 129] He korostavat, että kytkeytymiseen vaikuttaa kytkösten määrien lisäksi niiden kompleksisuus. Coadin ja Yourdonin määritelmä ei rajaa kytkeytymistä vain luokkien tai olioiden ominaisuudeksi, vaan käsitettä voidaan soveltaa myös esimerkiksi luokkajoukkojen tai komponenttien välille.

Kun luokilta odotetaan korkeaa koheesiota potentiaalisten virheiden välttämiseksi, niiltä myös toivotaan alhaista kytkeytymistä. [51] Luokkien välisten riippuvuuksien minimoimisella yritetään myös estää luokan muutosten propagoituminen muualle ohjelmaan [128], mikä vähentää ohjelmavirheiden korjauksesta syntyvää työmäärää. Korkean kytkeytymisen uskotaan myös vaikuttavan ymmärrettävyyteen, ylläpidettävyyteen ja järjestelmän kompleksisuuteen. [33] Matalan kytkeytymisen luokkien käyttäminen uudelleen on taas helpompaa kuin luokkien, jotka ovat riippuvaisia useista ulkoisista palveluista.

Koheesiota ja kytkeytymistä esitetty jopa olioanalyysin ohjenuoraksi [50], mutta käsitteet ovat kiinnostaneet enemmän metriikkamaailmaa. Erityisesti kytkeytymisen mittoja on kehitetty runsaasti, sillä Briandin ym. [33] jo viime vuosikymmenellä tekemässä kartoituksessa listattiin kolmekymmentä eri kytkeytymisen metriikkaa tai variaatiota. Silti uusia kytkeytymisen metriikoita esitellään jatkuvasti, esimerkiksi Arisholm ja kollegat [18] määrittelivät vuonna 2004 kaksitoista uutta suoritusaikaisen kytkeytymisen mittaa.

Tässä opinnäytteessä on sivuttu jo kolmea kytkeytymisen mittaa: Chidamberin ja Kemererin [46, 47] CBO:ta ja RFC:tä, sekä Abreun [8, 9, 10] COF:ää. Briand ja kollegat laajensivat Chidamberin ja Kemererin kytkeytymismetriikoita: Alkuperäinen RFC(c) määriteltiin vain luokan c kutsumien ja sen omien metodien summaksi. [47] RFC*(c) on luokan c käyttämien metodien transitiivinen sulkeuma. [33]

Chidamber ja Kemerer [47] sisällyttivät CBO:n alkuperäiseen laskentaan myös luokan esi-isät. Briandin ym. [33] CBO´(c):ssa ei oteta huomioon luokkia, joiden perillinen luokka c on. Näiden kytkeytymismittojen lisäksi tässä opinnäytteessä esitellään vielä Lin ja Henryn [109] MPC ja DAC sekä Leen ja kollegoiden tietovuohon perustuva ICP. Luvun lopussa käsitellään vielä Briandin, Dalyn ja Wüstin [33] kytkeytymismittojen luokittelukehys sekä empiiristen tutkimusten tuloksia.

2.4.1 MPC ja DAC

Li ja Henry [109] huomauttivat CBO:n pitävän kaikkia kytkeytymisen muotoja samanarvoisina ja laskevan ne yhdeksi arvoksi. Hienojakoisemman määrittelyn kytkeytymisen muodoista tarjoavat Briand, Daly ja Wüst [33], jotka luokittelivat kahden luokan väliset kytkökset yhdeksään kategoriaan. Heidän jaottelussaan löytyy mm. yhteisen data käyttö, metodiosoittimen välittäminen ja metodin parametrityyppiin perustuva kytkeytyminen.

Viestinvälitys-, perintä- ja tietotyyppikytkeytyminen riittivät Lin ja Henryn [109] tarkasteluun. He pitivät DIT:tä ja NOC:tä käypinä perintäkytkeytymisen mittareina ja määrittelivät kahdelle muulle kategorialle uudet mitat. Lin ja Henryn [109] esittelemät kyt-

keytymismetriikat ovat viestinvälityskytkentä (engl. *Message Passing Coupling*, MPC) ja kytkeytyminen tietotyyppien kautta (engl. *Data Abstraction Coupling*, DAC).

MPC määritellään luokan kutsumien ulkoisten metodien määräksi. Laskennassa huomioidaan ainoastaan luokassa toteutetut palvelut. Lin ja Henryn mielestä MPC kertoo, kuinka riippuvainen luokan lokaalien metodien toteutus on muista luokista. [109] Briand ym. [33] huomauttavat määritelmän jättävän epäselväksi lasketaanko ylikirjoitetut metodit paikallisiksi ja pidetäänkö perittyjä metodeja luokan ulkopuolisina. Jos perityt piirteet lasketaan luokan ulkopuolisiksi, ne pitäisi sisällyttää luokan käyttämien ulkopuolisten metodien summaan. Ajatus on outo, sillä hyväksyttävä tapa perityn palvelun toteuttamisessa on kutsua yliluokan vastaavaa palvelua osana suoritusta. Kun korkeaa periytymiskytkeytymistä pidetään toivottavana [51], onko tarpeellista tai järkevää sekoittaa kahta eri kytkeytymisen muotoa toisiinsa samassa mittarissa?

DAC on luokan käyttämien erilaisten tietotyyppien lukumäärä, eli niiden attribuuttien ja parametrien määrä, jotka eivät ole perustietotyyppisiä. Li ja Henry [109] perustelivat mittaa sillä, että luokan kytkeytymissuhteet ovat sitä monimutkaisempia, mitä enemmän se käyttää muita luokkia. Määritelmästä jäi tosin epäselväksi miten perittyjä piirteitä pitäisi käsitellä. Briand ym. [33] ratkaisivat tämän jättämällä periytyneet instanssimuuttujat ja piirteet laskennan ulkopuolelle.

Briand ym. [33] määrittelivät mitasta toisen version, DAC´:n, joka laskee yhteen luokan käyttämien erilaisten tietotyyppien määrän. DAC´ laskee monet samaa luokkaa edustavat attribuutit vain yhdeksi kytkökseksi, missä DAC:ssä kytköksiä tulee muuttujien määrän verran. DAC:n ja DAC´:n välinen ero kytkösten esiintymistiheyden käsittelyssä on merkittävä, sillä erilaisten kytkösten määrä edustaa hienovaraista käsitystä siitä, kuinka laajasti luokka on kytkeytynyt järjestelmään. Kytkösten kokonaismäärä taas kertoo kytkeytymisen vahvuudesta.

Hitz ja Montazeri [82] antavat esiintymistiheyden huomioimisesta kaksi ääriesimerkkiä: Luokalla c_1 on kymmenen luokan d edustajaa instanssimuuttujina. Luokalla c_2 on

kymmenen attribuuttia, mutta jokainen niistä edustaa eri tietotyyppiä. DAC on molemmille 10, mutta DAC´(c_1) on 1, kun taas DAC´(c_2) on 10. Luokka c_2 on selvästi laajemmin kytkeytynyt järjestelmään, sillä se on riippuvainen kymmenestä luokasta, c_1 kun taas yhdestä.

Toisaalta kytkös luokkien c_1 ja d välillä on vahvempi kuin yksikään c_2:n riippuvuuksista. Kytkösten vahvuus vaikuttaa niiden määrän ohella ylläpidettävyyteen ja testattavuuteen. [33] Julkisen rajapinnan muutos palvelimessa voi olla yhteen luokkaan vahvasti kytkeytyneellä c_1:llä huomattavasti työteliäämpi korjata kuin luokalle c_2. DAC´, kuten CBO ja COF, tarkastelevat kuinka moneen luokkaan kytköksiä löytyy, eivätkä huomioi kytkösten vahvuutta. Näille mitoille luokat ovat joko kytkeytyneitä tai eivät ole; kytkösten vahvuuksilla ei ole väliä.

Li ja Henry [109] mittasivat yli kolmen vuoden ajalta kahden kaupallisen järjestelmän ylläpidon vaatimaa työtä. Työmäärä arvioitiin lähdekoodirivien muutoksesta. Heidän tilastoanalyysissään MPC ja DAC toimivat yhdessä Chidamberin ja Kemererin metriikkapaketin kanssa parempina ylläpitämisen työmäärän ennustajina kuin perinteiset kokomitat.

2.4.2 Tietovuohon perustuva kytkeytyminen

Y. Lee, B. Liang, S. Wu ja F. Wang esittelivät vuonna 1995 paperissaan *Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow* tietovuohon perustuvan kytkeytymismitan (engl. *Information-flow-based Coupling*, ICP). ICP laskee metodin käyttämien luokan ulkopuolisten piirteiden kutsukertoja, painottaen niitä parametrien määrällä. Metriikan tavoitteena on mitata metodien välisen tiedonkulun määrää, ja näin arvioida kytkeytyneisyyttä. [33]

Briandin ja kollegoiden formalismilla ilmaistuna ICP(c, m) yksittäiselle luokan c me-

todille m on [33]:

$$\text{ICP}(c,m) = \sum_{m' \in PIM(m) - (M_{NEW}(c) \cup M_{OVR}(c))} (\,1 + |Par(m')|\,) \times NPI(m,m') \quad (2.28)$$

Missä $PIM(m)$ on kaikkien niiden metodien joukko, jota polymorfismin ja dynaamisen sidonnan vuoksi metodi m saattaa toteutuksessaan käyttää, M_{NEW} on luokkaan lisätyt metodit ja M_{OVR} on luokan ylikirjoittamien metodien joukko. $Par(m)$ on metodin m parametrien joukko ja $NPI(m,m')$ on metodin m polymorfisten kutsujen määrä metodille m'. [33]

ICP(c,m) laskee metodille summan sen polymorfisesti muista luokista herättämien metodien kutsukerroista, painotettuna kutsuvan piirteen parametrien määrällä, eli kuinka paljon informaatiota virtaa pois metodista. Lee ja kollegat määrittelivät kytkeytymismitan skaalautuvaksi, jolloin se voidaan laskea myös isommille kokonaisuuksille. ICP:t luokille (kaava 2.29) tai luokkajoukoille (kaava 2.30) ovat formaalisti lausuttuna [33]:

$$\text{ICP}(c) = \sum_{m \in M_I(c)} \text{ICP}(c,m) \quad (2.29)$$

$$\text{ICP}(SS) = \sum_{c \in SS} \text{ICP}(c) \quad (2.30)$$

Missä $M_I(c)$ on luokassa c implementoitujen metodien joukko ja SS on tarkasteluun valittujen luokkien joukko. Luokan ICP on siinä toteutettujen metodien tietovuokytkeytymisen summa. Vastaavasti luokkajoukolle ICP on sen luokkien arvojen summa.

Lee ym. erottelivat kytkeytymisen esivanhempiin ja ei-sukulaisiin. He esittelivät kummallekin oman tietovuohon perustuvan mitan. Kytkeytyminen muihin kuin perittyihin piirteisiin (engl. *Noninheritance-based ICP*, NIH-ICP) lasketaan vastaavasti kuin alkuperäiset [33]:

$$\text{NIH-ICP}(c,m) = \sum_{m' \in R} (1 + |Par(m')|) \times NPI(m,m') \quad (2.31)$$

$$\text{missä } R = PIM(m) \cap \Big(\bigcup_{c' \in Ancestors(c)} M(c') \Big)$$

$$\text{NIH-ICP}(c) = \sum_{m \in M_I(c)} \text{NIH-ICP}(c, m) \tag{2.32}$$

$$\text{NIH-ICP}(SS) = \sum_{c \in SS} \text{NIH-ICP}(c) \tag{2.33}$$

Missä $Ancestors(c)$ on luokan c esivanhempien joukko. Periytymiskytkeytyminen (engl. *Inheritance-based ICP*, IH-ICP) lasketaan myös samalla tavalla [33]:

$$\text{IH-ICP}(c, m) = \sum_{m' \in S} (1 + |Par(m')|) \times NPI(m, m') \tag{2.34}$$
$$\text{missä } S = PIM(m) \cap \Big(\bigcup_{c' \in C - \{c\} \cup Ancestors(c)} M(c') \Big)$$

IH-ICP(c) ja IH-ICP(SS) saadaan sijoittamalla kaavoihin 2.32 ja 2.33 ei-periytyvien mittojen tilalle periytymisen huomioivat versiot. Kahden edellä määrityn mitan välillä vallitsee tärkeä yhtälö: ICP = IH-ICP + NIH-ICP, joka pätee sekä metodeille, luokkille että luokkajoukoille. [33]

ICP laskee metodista muihin luokkiin suuntautuvaa tietovirtaa, eli metodin toimintaa palveluiden asiakkaana. Briand ym. [33] nimittävät lähestymistapaa tuontikytkeytymiseksi (engl. *import coupling*), sillä ICP tarkastelee kuinka kytkeytynyt metodi on muuhun järjestelmään. Vientikytkeytymisessä (engl. *export coupling*) painopisteenä on luokkaan tulevat kytkökset siitä lähtevien sijasta. Useimmat kytkeytymismitat tarkastelevat vain tuontikytkeytymistä, mutta CBO ja COF poikkeuksellisesti yhdistävät molemmat suunnat samaan mittaan.

Luokalle laskettava ICP on intuitiivinen, sillä ymmärrettävästi luokan informaatiovirta on sen metodien virtojen summa. Yksittäiselle metodille laskettava ICP(c, m) ei taas anna täyttä kuvaa metodin kytkeytymisestä, sillä siinä jätetään huomioimatta kytkökset saman luokan sisällä. Todennäköisesti Lee ja kollegat tarkoittivat kuitenkin ICP(c, m):n ainoastaan apumitaksi, eikä käytettäväksi yksittäisen metodin kytkeytymisen tarkasteluvälineenä.

ICP(SS) toimii käänteisesti verrattuna metodiversioon, sillä luokkajoukon tietovuokytkeytyminen laskee mukaan joukon sisäiset kytkökset. Näin ICP:tä ei voi käyttää tar-

kasteltaessa kuinka kytkeytynyt osajärjestelmä on muuhun ohjelmaan. Leen ja kollegoiden metriikan käyttäminen luokkajoukon sisäisen kytkeytymisen mittana on myös mahdotonta, sillä laskettaessa yhteen yksittäisten luokkien kytkeytymisiä huomioidaan myös luokkajoukon ulkopuoliset kytkökset. ICP(SS):n käyttäminen koko järjestelmän kytkeytymisen arvioimiseen ei ole mielekästä, sillä mittaa ei ole normalisoitu. Briand ja kollegat [32, 33] väittävät toistuvasti luokkamittojen skaalautuvan luokkajoukoille ja järjestelmille sopiviksi, mutta ainakin ICP:n summaperustainen ratkaisu jättää tarkasteluun huomattavia puutteita.

2.4.3 Kytkeytymisen luokittelukehys

Briand, Daly ja Wüst [33] määrittelivät myös kytkeytymiselle luokittelukehyksen, joka muistuttaa koheesiokehystä. Briandin ym. kehys mahdollistaa kytkeytymismittojen keskinäisen vertailun ja auttaa käyttäjää valitsemaan tarpeisiin sopivan mitan, tai määrittelemään kokonaan uuden. Taulukossa 2.3 on esitetty tässä opinnäytteessä mainitut kytkeytymismitat. Briandin ym. kehys koostuu kuudesta kriteeristä, jotka ovat [33]:

Kytkeytymisehto. Mikä mekanismi kytkee kaksi luokkaa toisiinsa? Esimerkiksi COF:n mielestä luokat ovat kytkeytyneet, jos toinen käyttää toisen luokan piirteitä. DAC:n kytkeytymisehdossa taas lasketaan attribuuttien tyyppejä.

Kytkösten suunta. Kytkeytymismitoissa voidaan tarkastella luokkaan tulevia tai lähteviä kytköksiä. COF käsittelee suunnat erikseen, mutta laskennassa se samaistaa ne. CBO taas tulkitsee molempia suuntia yhdenvertaisina eikä erottele niitä.

Hienojakoisuus. Kriteeri muodostuu kahdesta osasta: abstraktiotasosta ja kytkösten laskentatavasta. Kytkeytymistä voidaan tarkastella attribuutti-, metodi-, luokka-, luokkajoukko- tai järjestelmätasoilla. [33] Laskentatapa kertoo kuinka tarkasti kytkeytymisen vahvuus huomioidaan. Esimerkiksi CBO:n ja COF:n binäärinen laskentatapa

Taulukko 2.4: Kytkeytymiskehys ja esimerkkimittojen luokittelu.

			Hienojakoisuus				Perintä		
	Kytkeyty-misehto	Suunta	Abstrak-tiotaso	Laskentatapa	Palvelimen pysyvyys	Epäsuorat kytkökset	Tarkastelu-joukko	Polymor-fismi	Perityt piirteet
CBO CBO´	metodien käyttö, attribuuttiviittaus	molemmat	luokka	Binäärinen	–	ei	kaikki ei-sukul.	kyllä	ei
RFC RFC´	metodien käyttö	Tulevat	luokka	Metodien lukumäärä	–	ei kyllä	kaikki	kyllä	ei
COF	metodien käyttö, attribuuttiviittaus	molemmat‡	järjest.	Binäärinen	–	ei	ei-sukul.	kyllä	ei
MPC	metodien käyttö	Tulevat	luokka	Kytkösten määrä	–	ei	kaikki	ei	ei
DAC DAC´	attribuuttien tyyppi	Tulevat	luokka	Kytkösten määrä Kytkeytyneiden osien määrä	–	ei	kaikki	–	ei
ICP IH-ICP NIH-ICP	metodien käyttö	Tulevat	metodi, luokka, joukko	Yksittäisten kytkösten määrä	–	ei	kaikki sukul. ei-sukul.	kyllä	ei

† Briandin ym. [33] suosituksia

‡ COF laskee kummankin suunnan erikseen.

— luokat ovat joko kytkeytyneet tai eivät ole — on Briandin ym. [33] esittämistä tavoista karkein mahdollinen malli tarkastella kytkeytymistä.

Palvelimen muuttumattomuus. Luokka voi käyttää vakaita (esim. luokkakirjastot) tai muutosherkkiä palveluita. Kytkeytymistä muuttumattomiin piirteisiin pidetään hyväksyttävämpänä vaihtoehtona, koska todennäköisyys palvelimen muuttumisesta johtuvaan luokan päivittämiseen on pienempi. [82] Briand ym. [33] ehdottivat palvelimen pysyvyyden huomioimista kytkeytymismitoissa, mitä yksikään esitetyistä metriikoista ei tee.

Epäsuorat kytkökset. Jos luokka c kutsuu luokan d:n piirteitä, joiden toteuttamisessa d käyttää apuna luokkaa e, niin luokka c on epäsuorasti kytkeytynyt luokkaan e. Esitellyistä mitoista ainoastaan RFC´ huomioi epäsuoran kytkeytymisen.

Perintä. Kriteeri voidaan jakaa kolmeen osaan: Huomioidaanko mitassa kaikki luokat, ainoastaan esivanhemmat vai pelkästään ei-sukulaisluokat? Huomioidaanko laskennassa polymorfismin vaikutus? Sisällytetäänkö perityt piirteet tarkasteluun?

2.4.4 Kritiikkiä ja empiirisiä tuloksia

Briand ja kollegat [36] vertasivat 28 kytkeytymismitan tuloksia opiskelijaprojektien virheraportteihin. He olettivat luokan olevan sitä virhealttiimpi, mitä kytkeytyneempi se on. Samoin he uskoivat kirjastoihin kytkeytyneiden luokkien olevan vähemmän alttiimpia virheille kuin järjestelmäluokkiin kytkeytyneillä. Tämän vuoksi Briand ym. mittasivat erikseen kytkeytymiset järjestelmäluokkiin ja kirjastoihin.

Kytkeytymismitoilla oli tutkimuksessa huomattava yhteys virheraportteihin, erityisesti CBO:n, RFC:n ja ICP:n eri versiot sekä MPC osoittivat selkeää korrelaatiota luokkien virheisiin. Toisaalta luokan tuontikytkeytymisen mittojen ja virheiden välillä ei ollut merkitsevää korrelaatiota. [36] Tulos on siitä erikoinen, että samasta aineistosta Briandin, Devanbun ja Melon [34] mittauksissa samoille vientikytkeytymisen mitoille löydettiin

korrelaatio, kun tarkasteltiin järjestelmää kokonaisuutena ilman jakoa kirjasto- ja järjestelmäluokkiin.

Briand, Wüst ja Lounis [37] toistivat mittauksen käyttäen ammattilaisten rakentamaa ohjelmaa. Mitattu LALO-järjestelmä on huomattavasti opiskelijaprojekteja suurempi, minkä Briand ym. huomioivat kytkeytymismittareiden korkeammissa arvoissa. Mittauksen tulokset muistuttavat aiempaa tutkimusta, erityisesti CBO ja CBO´ korreloivat vahvasti virhealttiuden kanssa. Erona aikaisempaan myös vientikytkeytymisen metriikat korreloivat virheiden kanssa — tosin heikoimmin kuin muut kytkeytymisen mitat.

Briand ja kollegat [37] laskivat uudemmassa tutkimuksessa myös kytkeytymismittojen ja koon välisen riippuvuussuhteen. DAC, DAC´ ja RFC korreloivat metriikoista eniten kokomittojen kanssa, mutta näidenkin kytkeytymismittojen suhde oli tilastollisesti heikko. Tulos ei ole yllättävä, sillä RFC:hen lasketaan mukaan luokan omat metodit. [37] DAC ja DAC´ taas laskevat luokan attribuuttien määriä, mikä selittää riippuvuussuhteen kokoon.

Opinnäytteessä mainittujen metriikoiden lisäksi Briand, Devanbu ja Melo [34] esittelivät 18:sta C++:lle kehitettyä kytkeytymismittaa. Briandin ja kollegoiden mitat pyrkivät havainnoimaan jokaista heidän löytämäänsä kytkeytymisen muotoa. Empiirisissä tutkimuksissa mitat ovat osoittaneet riippuvuussuhdetta virheisiin, mutta heikommin kuin muut tässä opinnäytteessä esitellyt. [34, 36, 37] Selvästi kytkeytymiselle voidaan määrittää monia muotoja, mutta Briandin ym. kahdeksantoista, Arisholmin ym. [18] kahdentoista dynaamisen ja Kaungin ym. [96] kahdeksan metoditason kytkeytymismitan paketit tuntuvat liioitellun laajoilta.

LALO-järjestelmän mittauksissa [37] useat Briandin mitoista jäivät ilman tuloksia, koska järjestelmässä ei ollut käytetty kyseisen tyypin kytkeytymistä. Paketin muiden mittojen maksimiarvot vaihtelivat kahdesta 250:neen. Yksikään Briandin ym. luokittelemista kytkeytymisen muodoista ei ole mittauksissa ollut virhealttiimpi kuin muut ja kun mitat ovat vielä toimineet heikommin virheiden ennustajina, on niiden käytettävyys kyseenalai-

nen. Briandin ym. mitoille saattaisi tosin löytyä käyttöä refaktorointitarpeen tunnistajina.

Joshi ja Joshi [92] esittelivät kaksi kytkeytymismittaa, joiden tarkoituksena on mahdollistaa hienovaraisempien refaktorointipäätösten tekeminen. He määrittelivät suhteellisen kytkeytymismitan metodille (engl. *Relative Method Coupling*, RMC) ja attribuutille (engl. *Relative Inward Coupling*, RIC) ja neuvoivat näiden käyttämistä *Move method* ja *Move attribute* -refaktorointitarpeiden tunnistamiseen. RMC ja RIC sekä LCIC edustavat tuoretta ajattelua metriikkamaailmassa, missä liian monen metriikan itseisarvo on tarjota vain mahdollisuus ohjelman piirteiden mittaamiselle. Yhteys ulkoisiin laatutekijöihin [32, 33, 81] tai virheiden paikannus on monelle mitalla toissijainen tehtävä.

2.5 Metriikoiden käyttäminen

Ohjelmistometriikoiden käyttäminen ei saa olla itseisarvo, vaan niille on kehitystyössä selkeästi osoitettava käyttökohde ja -tarkoitus. Useimpien mainittujen oliospesifien mittojen olemassaoloa on oikeutettu ainoastaan virhealttiin ohjelmakoodin osoittamisella. Esitettyjen empiiristen tutkimusten valossa tehtävässä onnistuminen on ollut kyseenalaista: Osa tutkimuksista on löytänyt vahvan positiivisen riippuvuussuhteen metriikoiden ja virhemäärien välille. Kriittisemmät tutkimukset taas eivät ole löytäneet tilastollisesti merkitsevää korrelaatiota. Esimerkiksi El Emam kollegoineen [60] kritisoi oliometriikkamittauksien jättäneen järjestelmällisesti huomioimatta ohjelmakoon vaikutuksen, osan oliomitoista on osoitettu toimivan vain välittäjinä kokomittojen arvoille.

Useimmilta sisäisen laadun mitoilta puuttuvat selkeät käyttö- ja kehitysohjeet, tai ne ovat ilmaistu puutteellisesti. Suurin osa mitoista on ohjeistettu kuten Chidamberin ja Kemererin [47] metriikkapaketti, joka ei anna neuvoja mittojen käyttöön. Välimuoto on antaa ohjeet Abreun [9] tapaan, joka asettaa mitoille rajat, mutta ei tarkemmin ohjeista, miten ne pitäisi saavuttaa. Ääripäänä taas ovat Lorenzin ja Kiddin [112] mitat sekä Hitzin ja Montazerin [81] LCOM4, joille on annettu kynnysarvot ja toimintaehdotus, miten yli-

tetyt raja-arvot voi saavuttaa ohjelman laatua parantaen. Ilman ohjeita jääneet metriikat ovat niitä, joiden mittaamien ominaisuudet eivät ole ohjelmasta helposti hahmotettavissa. Tällaisista esimerkkinä on Abreun polymorfismikerroin. Rajoja tärkeämpää on suuntaa antava toimintaohje, jolla metriikoiden osoittamiin puutteisiin voi reagoida. On työajan haaskausta laskea metriikat niiden itsensä vuoksi, ja jättää puuttumatta niiden ennustamiin ongelmiin.

Empiirisissä tutkimuksissa ongelmallista on lähestymistapa, joka ei tue metriikoiden käyttöönottoa ohjelmistoprojekteissa. Suurin osa opinnäytteessä läpikäydyistä metriikka-analyyseista on laskettu projektin päätyttyä toimitetuista lähdekoodeista, eikä kehitys-syklien välituotoksista. Harvat tutkimukset on suoritettu osana tuotantoprosessia, jolloin metriikoiden arvot ovat voineet ohjeistaa kehityksen suuntaa. Esimerkiksi Karp [95] mittasi kahden Symbianilla toteutetun ohjelman eri kehitysversioissa oliometriikoiden ja virhetiheyden arvoja. Mittojen tulokset heikkenivät uusien ohjelmaversioiden myötä, vaikka järjestelmän ulkoinen laatu parani. Karpin pro gradussa ei raportoitu käytettiinkö mittojen arvoja hyödyksi ohjelman kehittämisessä, mutta todennäköisesti niitä ei huomioitu. Jos metriikoiden arvoja olisi seurattu, olisi myös sisäisen laadun mittareiden pitänyt parantua kehitysversioiden myötä. Ongelma on yleinen, sillä yhdessäkään tässä opinnäytteessä viitatussa tutkimuksessa ei ole osoitettu kiistattomasti metriikoiden käytön parantavan ohjelman sisäistä laatua.

Metriikoiden tarkoituksenmukainen ja pitkäjänteinen käyttäminen on kuitenkin avainasemassa niiden tarjoamien mahdollisuuksien hyödyntämiseen. Victor Basili [21, 23] on esitellyt yhden tavan mittojen johdonmukaiseen käyttämiseen. Hänen *tavoite-kysymys-metriikka* –mallissa (engl. *Goal-Question-Metric*, GQM) ensimmäinen askel on määritellä haluttu päämäärä. Kysymysten avulla kuvaillaan tavoitetta, ja soveltuvilla mitoilla pyritään vastaamaan esitettyihin kysymyksiin. Fentonin ja Pfleegerin [63] mukaan malli auttaa keräämään projektille tarpeelliset mittaustulokset ja samalla kysymykset kertovat mihin kerättyä dataa käytetään. Taulukossa 2.5 on esitetty käytännön esimerkkinä osa

Taulukko 2.5: Osa AT&T:n käyttämästä GQM-mallista.

Tavoite	Kysymys	Metriikka
Inspektoinnin kehittäminen	Kuinka tehokas inspektointiprosessi on?	Normaalisti löydettyjen virheiden määrä Havaittujen virheita per KLOC Inspektoinnin määrä Valmistelujen määrä Tarkastettuja lähdekoodirivien määrä
	Mikä on inspektointiprosessin tuottavuus?	Normaalisti löydettyjen virheiden määrä Työmäärä havaittu virhettä kohti Inspektoinnin määrä Valmistelujen määrä Tarkastettuja lähdekoodirivien määrä ⋮

Barnard ja Price [19]

AT&T Bell Laboratories:in GQM-analyysin tuloksia projektista, jossa selvitettiin ohjelmakoodin katselmoinnin höytyjä. [19]

Fenton ja Neil [62] väittävät metriikoita tehokkaiksi projektijohdon työvälineiksi, mikä näkyy Trienekensin ym. [159] selvityksessä. Trienekens ja kollegat haastattelivat 49:ää Philipsin CMM-kypsyysmallia (engl. *Capability Maturity Model*) noudattavaa ohjelmointiryhmää metriikoiden käytöstä. Kehitysryhmät sijaitsivat Aasiassa, Amerikassa ja Euroopassa. Noin puolella ryhmistä oli formaali metriikkaohjelma ja yli 63 % vastaajista ilmoitti käyttävänsä metriikoita ohjelmistokehityksen ohjaamiseen.

Philipsin ohjelmointiryhmien käyttämät metriikat vaihtelivat laajasti, mutta ahkerimmin seurattiin työmäärää, ohjelmakokoa ja toimitusaikaa. Erinäisiä virhemittoja hyödynsi vain kolmannes vastaajista. Trienekensin ym. [159] tutkimuksessa mainittiin sisäisen laadun mittareista ainoastaan McCaben syklomaattinen kompleksisuus, jota yksikään vastaaja ei käyttänyt. [159] Selvityksessä tarkasteltiin tosin vain ryhmien ilmoittamia virallisia mittareita, jolloin yksittäisten työntekijöiden käyttämät mitat ovat saattaneet jäädä kirjaamatta.

CMM-yritysten metriikkakäytäntöjä ovat tutkineet myös Paulk [141] ja Jalote [87].

Paulk huomasi kypsiltä yrityksiltä löytyvän metriikoihin erikoistunutta henkilökuntaa. Hänen mukaansa yritykset käyttivät pitkälti GQM-mallia, vaikka yksittäisiä työntekijöitä ja ryhmiä kannustettiin kehittämään ja käyttämään omiin tarpeisiin soveltuvia mittoja. Jaloten [87] selvitys intialaisten CMM-yritysten metriikkakäytännöistä vastaa pitkälti Trienekensin ym. [159] selvitystä, mutta Philipsin kehitysryhmiltä raportoitiin vähemmän ohjelmistomittojen käyttöä.

Metriikoiden käyttöönottoon — kuten uuteen teknologiaan yleensä — liittyy myös eettinen näkökulma. Mittaustuloksia voidaan käyttää niiden kerääjiä vastaan, unohtaen subjektiivinen katsanto. Tietäessään suoritustaan arvioitavan, työntekijät saattavat myös kaunistella mittaustuloksia. Paulk [141] kertoi tapauksesta, jossa työnjohdon painostuksesta kehittäjä esikatselmoi tuotoksensa lyhentääkseen virallisten katselmointitilanteiden virheraportteja. Yksittäisen työntekijä toiminta aiheutti näin yrityksen mittaustulostietokantaan tilastollisesti haitallista poikkeamaa, mikä saattaa vaikeuttaa prosessin kehittämistä.

Luku 3

Ohjelmistoarkkitehtuurit

Ohjelmistojen laajuuden nopeassa kasvussa, on niiden ymmärtämisen helpottamiseksi kehitetty uusia abstrahointitasoja ja -työvälineitä. Ohjelmistoarkkitehtuuri (engl. *Software architecture*) on pitkään ollut vain yksi ohjelman osituksen vaihe, mutta viime vuosikymmenellä löydettiin sen tehokkuus myös suunnitteluvälineenä. Huonosti suunniteltu arkkitehtuuri voi tuhota projektin onnistumismahdollisuudet. [69] Hyvin toteutettu arkkitehtuurisuunnittelu taas tukee siirrettävyyttä, uudelleenkäytettävyyttä ja tehokkuutta. [102]

Ohjelmistoarkkitehtuureille on annettu useita määritelmiä ja tehtäviä. Esimerkiksi D'Souzan ja Willsin [54] mielestä arkkitehtuurin tarkoitus on estää toteuttajien liiallinen luovuus. Koskimies ja Mikkonen [102] nimittivät ohjelmistoarkkitehtuuria järjestelmän ytimen perustuslaiksi, jota noudattaen ohjelmistoa kehitetään ja ylläpidetään. Heidän mukaansa arkkitehtuuriin kirjattuja päätöksiä voi muuttaa vain hyvin painavilla perusteilla, mutta ytimen ulkopuolisia osia saa vapaasti muunnella soveltuvammiksi. IEEE:n standardissa ohjelmistoarkkitehtuurit määritellään "*järjestelmän perusorganisaatioksi, sisältäen arkkitehtuurikomponentit, niiden keskinäiset suhteet ja niiden suhteet ympäristöön, sekä periaatteet joiden mukaan järjestelmää suunnitellaan ja kehitetään*". [85, s. 3]

Seuraavaksi tarkastellaan arkkitehtuurikomponentin määritelmiä. Luvussa tarkastellaan myös arkkitehtuurin kuvaustyökaluja ja dokumentointia. Lopuksi keskitytään vielä keskeisimpiin arkkitehtuurityyleihin sekä suunnittelu- ja antisuunnittelumalleihin.

3.1 Arkkitehtuurin osat

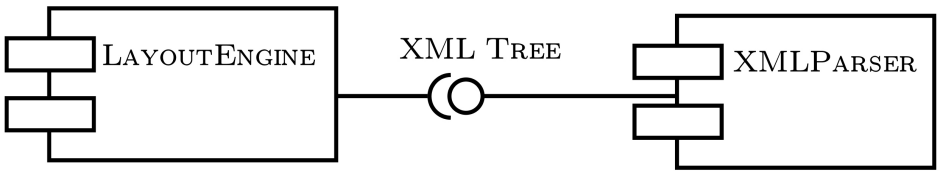
Ohjelmistoarkkitehtuurin määritelmä tai tehtävät eivät ole vielä vakiintuneet, ja ehdotelmia löytyy useita esimerkiksi lähteestä [152]. Useat määritelmät esittävät arkkitehtuurin ohjelmakomponenttien ja niiden välisten suhteiden kuvaukseksi sekä ratkaisujen perusteluiksi. Arkkitehtuurin tärkeimmän osan, komponentin, määritelmä on myös valitettavan epämääräinen, vaikka arkkitehtuurien dokumentoinnista on hyvinkin selkeitä ohjeita.

3.1.1 Komponentti — Arkkitehtuurien rakennuspalanen

Komponentit ovat arkkitehtuurien perusyksiköitä, pienimpiä arkkitehtuurisuunnittelussa käsiteltäviä ohjelmapalasia. Pelkistäen arkkitehtuurin voi väittää käsittävän ohjelman dekomposition komponenteiksi, ja näiden komponenttien välisten suhteiden hallinnaksi. Komponentit toimivatkin erottajana korkean ja matalan tason suunnittelun välillä, sillä komponenttien sisäistä rakennetta tarkastellaan vasta matalan tason suunnittelussa. [102] Koskimies ja Mikkonen määrittelevät komponentin "*itsenäiseksi ohjelmistoyksiköksi, joka tarjoaa palvelujaan hyvin määriteltyjen rajapintojen kautta*". [102, s. 54] Toiminnallisuuden perusteella rakennutetut komponentit kokoavat yhteen läheiset palvelut, jotka yleensä pohjautuvat samoihin tietorakenteisiin. Tällaisia palveluita käytetään tavallisesti samasta kontekstista, miksi niiden sijoittaminen samaan komponenttiin on perusteltua. [102]

Toimivan ohjelman rakentamiseksi ohjelmistokomponenttien pitää tehdä yhteistyötä, mutta komponenttien välisiä suoria riippuvuuksia pidetään huonona suunnitteluna. Yksinkertainen ratkaisu on määritellä ja käyttää komponenttien rajapintoja. Tällöin komponentti on riippuvainen palvelusta, eikä sen konkreetista toteutuksesta. Rajapintojen avulla palvelun toteutus voitaisiin vaihtaa helposti, jopa ajonaikaisesti. Ne ovatkin keskeinen työväline arkkitehtuurisuunnittelussa ja tärkeä osa komponentin määrittelyä. [102]

Komponentti voi pienimmällään koostua yhdestä luokasta ja suurimmillaan kokonai-



Kuva 3.1: Kaksi komponenttia, sekä vaadittu ja tarjottu rajapinta.

sesta sovelmasta, joka tarjoaa ohjelmoitavan rajapinnan. Isoimmille komponenteille voidaan laatia erikseen oma arkkitehtuuri, minkä avulla ohjelmistoarkkitehtuuri voidaan jakaa hierarkkisille tasoille. [102] Esimerkiksi Internet-selaimessa yksittäisenä komponenttina voisi olla XML-parseri, joka palvelunaan tuottaa annetusta tiedostosta selainytimen ymmärtämän tietorakenteen. Tälle komponentille olisi perusteltua myös laatia sisäinen arkkitehtuurikuvaus ja -dokumentaatio, jotka tukevat komponentin uudelleenkäyttämistä.

Kuvassa 3.1 on esitetty komponentit `LayoutEngine` ja `XMLParser` UML-notaatiolla. `XMLParser` on toteuttanut rajapinna `XMLTree`, ja tarjoaa palveluitaan sen kautta. Komponentti `LayoutEngine` taas vaatii toimiakseen samaisen rajapinnan palveluita. Kuvassa komponentit ovat yhdistetty toisiinsa `XMLTree`-rajapinnan kautta. Tällöin komponentit eivät ole riippuvaisia toisistaan —ainoastaan sovitusta rajapinnasta. Jos kohdetiedosto on toteutettu jollain toisella kuvauskielellä kuin XML:llä, voi ohjelma vaihtaa käytettävän parserin toiseen helposti. Komponentti `LayoutEngine` on kiinnostunut ainoastaan rajapinnan palveluista, ei niiden tuottajista.

Itsenäisiä, tarkan toiminnallisuuden toteuttavia, ohjelmistopalasia voidaan uudelleen käyttää lähes rajattomasti. Esimerkiksi `XMLParser` on kopioitavissa kaikkiin järjestelmiin, jotka käsittelevät XML:n mukaista tietoa. Uudelleenkäytettävyyden haasteena on rajapintojen standardisointi ja komponenttien räätälöitävyys. Komponenttisuunnittelussa on mahdollistettava arkkitehtuuripalasen mukauttaminen käyttöyhteyden tarpeisiin, joko sisäisen tilan muuttamisella, periytymisellä tai rajapintojen toteuttamisella. [102]

Komponenttien räätälöintiä suurempi haaste uudelleenkäytettävyydelle on rajapintojen standardisointi. Palvelurajapintojen ja komponenttien vaatiman infrastruktuurin yh-

tenäistämisellä pystyttäisiin avaamaan julkiset komponenttimarkkinat, joilla valmistajat tarjoaisivat tuotteitaan ja toteuttajat kilpailuttaisivat palveluita. [102] Esimerkiksi kuvan 3.1 `XMLParser`-komponentin uudelleenkäytettävyyttä tukisi `XMLTree`:n vaihtaminen DOM1-standardirajapinnan mukaiseksi. [166]

Arkkitehtuurikirjallisuudessa komponenttien määritelmä jätetään usein valitettavan avonaisiksi. Esimerkiksi Koskimiehen ja Mikkosen [102] määritelmässä komponentin koko vaihtelee yksittäisestä luokasta kokonaiseen järjestelmään. Komponenttien mittaamista varten määritelmä on kuitenkin kiinnitettävä konkreettisesti havainnoitaviin piirteisiin. Alaluvussa 4.1.1 komponentin määritelmälle esitetään konkreettinen sidonta, jota käytettiin komponenttimittojen empiirisessä arvioinnissa.

3.1.2 Arkkitehtuurin kuvaaminen ja dokumentointi

IEEE:n [85] määritelmässä keskeinen osa ohjelmistoarkkitehtuuria ovat suunnitteluperiaatteet, joiden perusteella järjestelmää kehitetään. Standardi määrittelee arkkitehtuurikuvauksen (engl. *Architectural description*, AD) kokoelmaksi arkkitehtuuria mallintavia artefakteja. Arkkitehtuurin kuvauksen yksityiskohdille annetaan useita ehdotuksia, mutta sen tarkkaa muotoa ei määrätä. Arkkitehtuuridokumentti konkretisoi kuvaukset ja dokumentti on tärkeä väline sidosryhmien välisessä kommunikaatiossa. Erilaisia dokumenttityyppejä on useita, mutta keskeisimpiä ovat alustava, järjestelmä- ja alijärjestelmäarkkitehtuuridokumentti. [102]

Maier, Emery ja Hilliard asettavat kolme minimivaatimusta arkkitehtuurikuvauksille [113]: Ensimmäisenä järjestelmään liittyvät sidosryhmät ja huolenaiheet on mainittava kuvauksessa. Toiseksi yksi tai useampi arkkitehtuurinäkymä järjestelmään on esiteltävä. Kolmanneksi tehtyjen valintojen perustelut ja suunnittelun periaatteet on kirjattava esille. Nämä ovat arkkitehtuurien kuvaamisen vähimmäisvaatimukset, mutta kuvaus voi myös sisältää selvityksen hylätyistä valinnoista tai analyysin arkkitehtuurinäkymien johdonmukaisuudesta. [85]

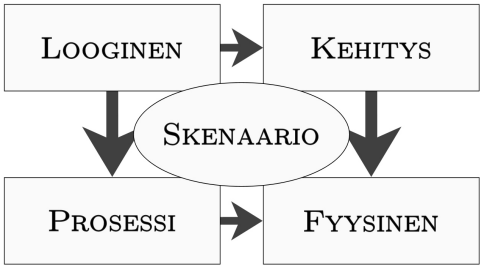
Maier ym. [113] korostavat kuitenkin useasti, ettei järjestelmän arkkitehtuuri ole sama kuin sen dokumentaatio. He painottavat arkkitehtuurin olevan järjestelmän kompleksinen ominaisuus, eikä yksittäinen artefakti. Heidän mukaansa jokaisella järjestelmällä on arkkitehtuuri, on se dokumentoitu tai ei. Esimerkiksi standardi kuvailee tilannetta, jossa ohjelmistolla ei ole arkkitehtuuria, mutta se voidaan luoda takaisinmallintamalla (engl. *Reverse engineering*). [85]

Koskimies ja Mikkonen [102] kannattavat huomattavasti tiukempaa lähestymistapaa. He mieltävät järjestelmäarkkitehtuuridokumentaation ohjelmistoprojektin avainartefaktiksi ja kyseenalaistavat takaisinmallintamalla luodut arkkitehtuurikuvaukset. Kärkevimpänä väitteenä he kiistävät järjestelmän arkkitehtuurin olemassaolon, ellei arkkitehtuuria ole dokumentoitu. Samoin heidän mielestään projektien, joiden arkkitehtuuridokumentaatio puuttuu, ei pitäisi antaa jatkua. Koskimies ja Mikkonen perustelevat näkökantaansa kehityksen muuttumisesta anarkiaksi perustuslain puuttuessa, mikä rämettää rakennettavan järjestelmän laatua sisältäpäin.

Konservatiivista, säntillistä dokumentaatiota kannattavien Koskimiehen ja Mikkosen ajatusmalli on eriskummallinen. Ohjelmistoprojektin avainartefakti on aina toimitetun tuotteen binäärikoodi. Esimerkiksi suosittujen ja menestyneiden ketterien menetelmien manifestissa korostetaan toimivaa tuotetta yli kattavan dokumentaation. [25] Samoin ketterän kehityksen periaatteissa painotetaan keskustelua ylivertaisena tiedonlevittämistapana. Tätä roolia Koskimies ja Mikkonen tarjoavat arkkitehtuuridokumentaatiolle.

Arkkitehtuurinäkymät

Riippumatta käytettävästä lähestymistavasta arkkitehtuuridokumentaatioon, näkökulmat (engl. *Viewpoints*) ja näkymät (engl. *View*) ovat tärkeä osa arkkitehtuurin konkretisointia. Näkökulma on järjestelmästä riippumaton tapa kuvata jotain arkkitehtuurin ominaisuutta. Näkymä on jonkin näkökulmaa mukainen kuvaus yksittäisen järjestelmästä. [102] Kruchtenin [103] esittelemät “4+1”-mallin näkökulmat ovat esitetty kuvassa 3.2.



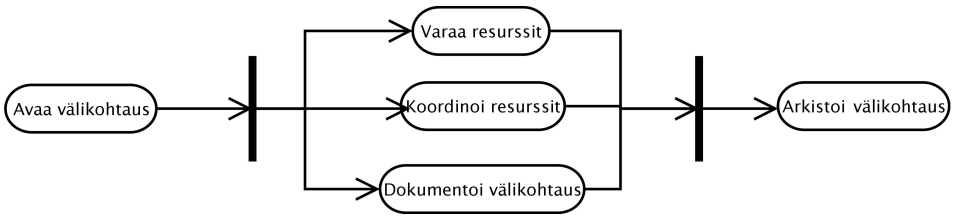
Kuva 3.2: Kruchtenin "4+1"-mallin mukaiset arkkitehtuurinäkökulmat. [103]

Kruchtenin viisi näkymäänsä ovat [103]:

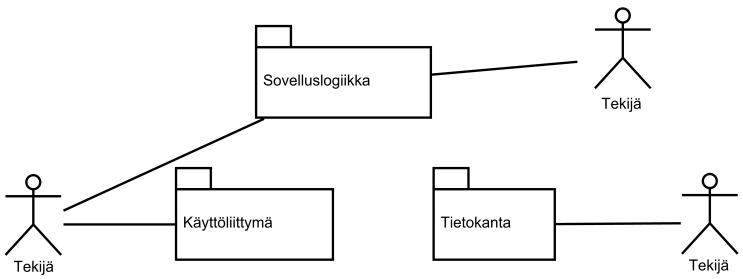
Looginen näkymä (engl. *Logical view*). Näkymä kuvaa järjestelmän komponenttien velvollisuudet ja keskeiset suhteet. Loogisia näkymiä voidaan kuvata UML:n luokka-, sekvenssi- ja kommunikaatiokaavioilla. [101] Esimerkiksi kuvassa 3.1 on esitetty kahden komponentin suhde luokkakaaviolla.

Prosessinäkymä (engl. *Process view*). Näkymä kuvaa millaisiin prosesseihin järjestelmän suoritus on jaettu, ja miten nämä kommunikoivat keskenään. Näkymä ottaa kantaa ohjelman suorituksen synkronointiin ja samanaikaisuuteen, ja sitä voidaan kuvata UML:n aktiviteettikaaviolla. [101] Kuvassa 3.3 on esitetty välikohtauksen — suorituksen ongelmatilanteen — hoitaminen UML:n aktiviteettikaaviolla.

Kehitysnäkymä (engl. *Development view*). Näkymä kuvaa millaisiin osiin järjestelmä on jaettu, jotta sitä voidaan kehittää erillisinä yksikköinä. UML:n pakettikaaviota



Kuva 3.3: UML:n aktiviteettikaavio, jossa on kuvattu välikohtauksen hoitaminen. Kohtaus koostuu viidestä prosessista. Tummennetut pystyviivat toimivat *portteina*, joista suoritus ei jatka ennen kun kaikki prosessit ovat valmiina. [39]



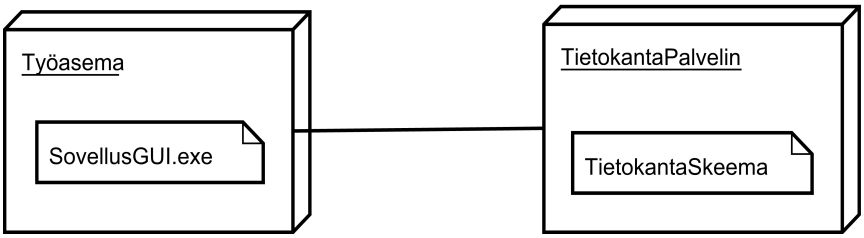
Kuva 3.4: UML:n pakettikaavio, johon on merkitty toteuttajat.

käytetään apuna näkymän mallintamisessa. [101] Kuvassa 3.4 on tästä yksinkertainen esimerkki.

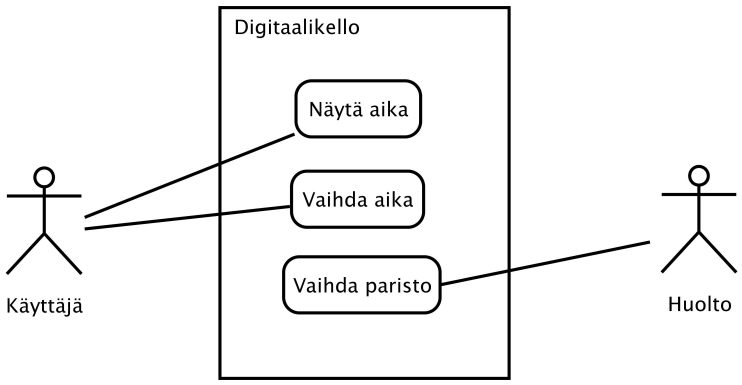
Fyysinen näkymä (engl. *Physical view*). Näkymä mallintaa miten järjestelmä on jaettu fyysisesti eri prosessoreiden kesken, ja miten nämä kommunikoivat keskenään. Fyysisellä näkymällä voidaan myös kuvata miten eri resurssit on jaettu laskentayksiköiden kesken, esimerkiksi miten tiedostot ja tietokannat on sijoiteltu. UML:n sijoittelukaaviota käytetään näkymän kuvaamisessa. [101] Kuvassa 3.5 on esitelty kahdelle eri laitteelle jaettu järjestelmä.

Skenaarionäkymä (engl. *Scenario view*, *Scenarios*). Näkymä kuvaa järjestelmän ulkoiset rajapinnat, eli kuinka käyttäjät tai muut järjestelmät ovat vuorovaikutuksessa kuvauksen kohteen kanssa käyttötapauksessa. Käyttötapauskaavio auttaa näkymän mallintamisessa. [101] Kuvassa 3.6 on esitetty yksinkertainen kellon käyttötapauskaavio.

Tarpeelliset näkökulmat ovat projektikohtaisia, esimerkiksi fyysinen näkökulma on



Kuva 3.5: UML:n sijoittelukaavio, jossa järjestelmä on jaettu kahdelle eri laitteelle.



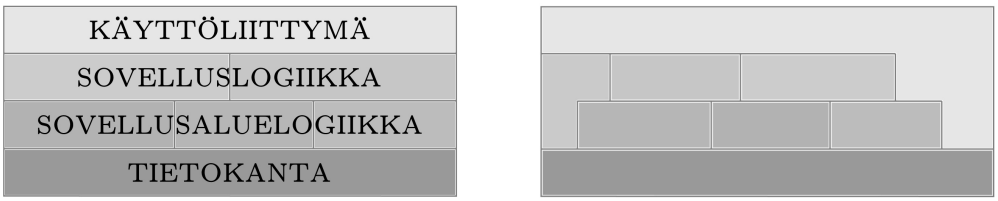
Kuva 3.6: Digitaalisen kellon käyttötapauskaavio, jossa on kaksi *aktoria* ja kolme käyttötapausta. Aktorit ovat järjestelmän ulkopuolisia tahoja, jotka kommunikoivat sen kanssa. [39]

hyödytön, jos järjestelmää ajetaan vain yhdellä suorittimella. Kruchtenin mallin lisäksi saatetaan tarvittaessa käyttää myös erikoisempia näkymiä. Esimerkiksi Koskimies ja Mikkonen [102] ovat määritelleet tuoterunkoarkkitehtuureille soveltuvan *Muuntelunäkymän* (engl. *Variability view*), joka kuvaa järjestelmän variaatiopisteet. Muuntelunäkymä siis kuvaa millainen järjestelmä on toteutusalustana uusille sovelluksille.

3.2 Arkkitehtuurityylit

Arkkitehtuurityylit (engl. *Architectural style* ja *Architectural pattern*) ovat yleisesti tunnettuja, hyväksi todettuja tapoja organisoida järjestelmän komponentit. Tyylit esiintyvät sovelluksissa harvoin puhtaina, sillä useimmiten järjestelmä on rakennettu soveltamalla useampaa tyyliä eri tarkoituksiin. [102] Jokainen arkkitehtuurimalli tuo mukanaan etuja ja haittoja [69], esimerkiksi tyyli saattaa edesauttaa modulaarisuutta ja uudelleenkäytettävyyttä tehokkuuden kustannuksella.

Arkkitehtuurityylien luettelo ei ole vakiintunut, ja esimerkiksi Malli-näkymä-ohjain -suunnittelua nimitetään sekä arkkitehtuurityyliksi että suunnittelumalliksi. [102] Samoin tapoja kategorisoida arkkitehtuurityylejä on esitelty useita. Seuraavassa käytetään Koskimiehen ja Mikkosen [102] esittelemää ryhmittelyä kolmeen tyylikategoriaan. Samalla tarkastellaan keskeisimmät arkkitehtuurityylit näistä kategorioista.



Kuva 3.7: Kerrosarkkitehtuurin kaaviokuva ilman ohituksia ja ohituksilla. Ohituksettomaan suunnitelmaan on merkitty esimerkki kerrosten tarkoituksesta. [102]

3.2.1 Ryhmittelyperustaiset tyylit

Ryhmittelyperustaiset arkkitehtuurityylit auttavat hahmottamaan järjestelmän jäseniä ja niiden vastuita. Kerros- ja tietovuoarkkitehtuurityyleissä joukko komponentteja ovat arkkitehtuurin kannalta aina samassa roolissa. Esimerkiksi kerrosarkkitehtuurissa ylemmät tasot tukeutuvat aina alempiin ja tietokantakomponentit sijoitetaan matalimmalle tasolle. [102]

Kerrosarkkitehtuuri

Kerrosarkkitehtuurissa (engl. *Layered systems*) komponentit ovat ryhmitelty eri kerroksiin niin, että korkeampi taso on toteutettu alemman abstraktiotason tarjoamilla palveluilla. [69] Kuvassa 3.7 on esitetty perinteinen nelikerroksinen arkkitehtuuri, sekä kerrosarkkitehtuuri ohituksilla. Ohituksia käytetään tyylissä välittämään palvelukutsu suoraan alimmille tasoille, sillä abstraktiotasojen ohittaminen parantaa arkkitehtuurin tehokkuutta. Ohituksia pidetään yleisesti hyväksyttävinä kerrosarkkitehtuurissa, mutta niiden liiallinen käyttäminen tuhoaa arkkitehtuurityylin tarkoituksen. [102]

Kerrosarkkitehtuuri auttaa kompleksisten ongelmien ratkaisemisessa [69] ja järjestelmän ymmärtämisessä [102], sillä vaikeampien ongelmien toteutus perustuu matalampien, konkreettisempien ja yksinkertaisempien tasojen toiminnallisuuteen. Arkkitehtuurityyli siis jakaa monimutkaisen ongelman pienempiin osaongelmiin, kunnes helpoimmat niistä on yksinkertaista toteuttaa alimmalla tasolla. Kompleksinen ongelma saadaan ratkaistua

näiden pienten osaongelmien ratkaisujen avulla.

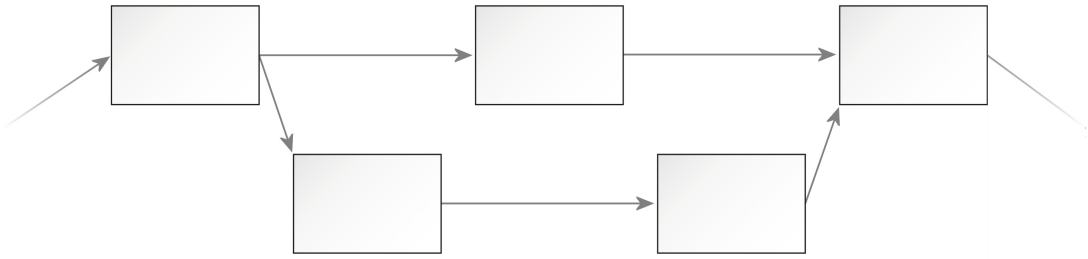
Kerrosarkkitehtuuri tukee uudelleenkäytettävyyttä [69], ylläpidettävyyttä ja muutettavuutta [102], sillä ideaalimuodossa abstraktiotaso on riippuvainen vain alemman kerroksen rajapinnasta. Tällöin koko kerroksen toteutus voidaan vaihtaa, ilman vaikutuksia ylempään tai alempiin abstraktiotasoihin. Kerroksia voidaan myös uudelleen käyttää joko yksittäisinä tai rakentamalla uusi sovellus alimpien kerrosten päälle.

Arkkitehtuurityylin ongelmana on tehottomuus, sillä palvelupyyntöjen siirtäminen aina alemmalle tasolle kuluttaa turhaan laskenta-aikaa. Pahimmillaan välitettyjen parametrien muotoa joudutaan tulkitsemaan arkkitehtuurin jokaisella tasolla erikseen. [102] Toinen ongelma on komponenttien sijoittaminen. Aina ei ole itsestään selvää, mille abstraktiotasolla komponenttin kuuluu. Tämä voi tuottaa ylimääräistä työtä. [69] Kolmas ongelma on virheidenkäsittely, sillä alemmalta tasoilta nousevat poikkeukset käsitellään vasta korkeammalla abstraktiotasolla. Tällöin niiden korjaaminen voi olla mahdotonta, sillä virheiden syntymisen syitä alemmalla tasolla ei tiedetä tai niihin ei voida vaikuttaa. [102]

Lähes kaikki järjestelmät voidaan kuvata kerrosarkkitehtuurina. [102] Eräs tapa jakaa esimerkiksi liiketoimintasovellus eri kerroksiin on esitetty kuvassa 3.7. Pankkimaailmassa sovellusaluelogiikkakerros voisi sisältää esimerkiksi käsitteiden `Tili` ja `Asiakas` toteutukset. Tällöin sovelluskerros sisältäisi eri tilityyppien toimintalogiikan. Garlan ja Shaw [69] listaavat kerrosarkkitehtuurin yleisimmiksi käyttökohteiksi tietokanta- ja käyttöjärjestelmät sekä erityisesti kerroksiset tiedonsiirtoprotokollat.

Tietovuoarkkitehtuuri

Tietovuoarkkitehtuurit (engl. *Pipes and filters*) koostuvat aktiivisista tietoa käsittelevistä komponenteista ja näitä yhdistävistä passiivisista tietoa kuljettavista väylistä. Komponentit ovat tilattomia ja itsenäisiä yksiköitä, jotka ovat riippuvaisia ainoastaan syötteistä. Ideaalitoteutuksessa käsittelijöiden järjestyksellä ei ole vaikutusta järjestelmän lopputulokseen. [69] Kuvassa 3.8 on hahmoteltu yksinkertaista tietovuoarkkitehtuurisovellusta, jossa



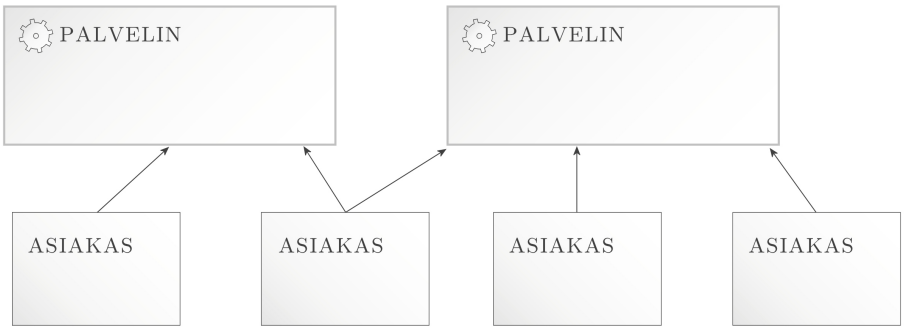
Kuva 3.8: Tietovuoarkkitehtuuri komponenteilla ja tietovirroilla. [102]

tietovuo haarautuu kahteen eri osaan kunnes yhdistyvät.

Arkkitehtuurityyli tukee uudelleenkäytettävyyttä, sillä mikä tahansa komponentti voidaan vaihtaa toisen tilalle, kunhan ne ymmärtävät samaa syötettä ja tuottavat samanmuotoisen tulosteen. [102] Tietovuoarkkitehtuurin komponentteja on helppo käsitellä ja ylläpitää, sillä muutokset ovat paikallisia ja riippuvuudet muuhun järjestelmään on minimoitu. Itsenäisyyden vuoksi komponentit ovat myös helppo hajauttaa omiksi prosesseikseen rinnakkaisiin järjestelmiin. [69]

Tietovuoarkkitehtuurilla on kuitenkin muutamia ongelmia. Ensinnäkin tyyli ei sovellu interaktiivisille järjestelmille, sillä käyttäjä ei voi seurata tai vaikuttaa sovelman tilaan. [102] Toiseksi rinnakkaisiin tietovirtoihin liittyvien virheiden jäljittäminen saattaa olla hankalaa. Kolmanneksi universaaleihin syötteisiin varautuvien käsittelijöiden kirjoittaminen on monimutkaista. Arkkitehtuurityylin suurin ongelma on kuitenkin syötteiden ja tulosten jäsentämisen vaatima laskentateho, sillä pahimmassa tapauksessa jokainen komponentti joutuu muuttamaan esimerkiksi XML-syötteen käsiteltäväksi informaatioksi ja takaisin XML:ksi. [69]

Arkkitehtuurityyli soveltuu tehtäviin, joissa tietoa jalostetaan askel kerallaan. Esimerkiksi ohjelmointikielten kääntäjät on perinteisesti toteutettu tietovuoarkkitehtuurin mukaisesti. Tyyli sopii myös hajautetuille järjestelmille ja signaalinkäsittelysovelmille. [69]



Kuva 3.9: Asiakas-palvelin -arkkitehtuuri. [102]

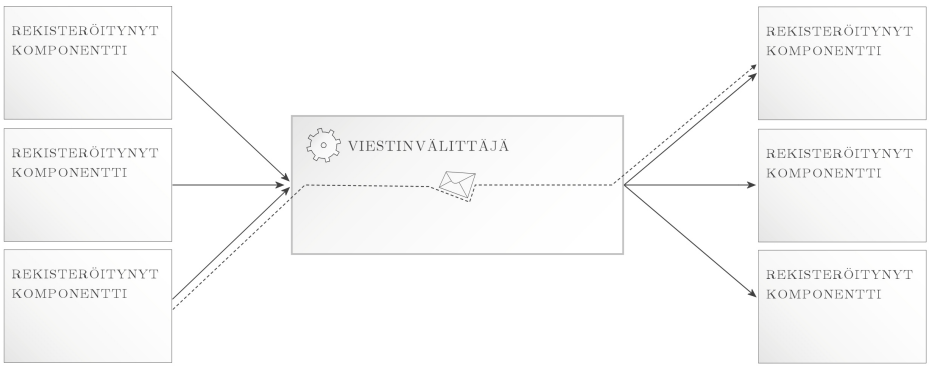
3.2.2 Palveluperustaiset tyylit

Palveluperustaisissa arkkitehtuurityyleissä komponentit esiintyvät kahdessa roolissa: palvelun tarjoajia ja niiden kuluttajia. Jonkin palvelun asiakas voi toimia toisen palvelun tuottajana, joten roolit eivät välttämättä ole kiinteitä. Yleensä palveluntarjoajilla on jokin resurssi, esimerkiksi tietokanta, minkä käytöstä komponentti määrää. [102]

Asiakas-palvelin -arkkitehtuuri

Asiakas-palvelin -arkkitehtuuri (engl. *Client-server architecture*) on olioparadigmaa mukaileva tyyli, jossa palvelin on kapseloinut hallittavan resurssin niin, ettei asiakkaiden tarvitse tietää resurssiin liittyvistä teknisistä yksityiskohdista tai muista sitä käyttävistä asiakkaista. Palvelin odottaa yleensä passiivisesti asiakkaan kutsua, yleensä omassa säikeessään tai prosessissaan, erillään palvelun käyttäjistä. [102] Kuvassa 3.9 on esitetty kaaviokuva asiakas-palvelin -arkkitehtuurista.

Tyylin selkein hyöty on työnjako, jossa komponenteilla on selkeät vastuut, rajapinnat ja kommunikointimallit. Suunnitteluparadigma tukee hajautettavuutta, sillä lähtökohtaisesti asiakkaat ja palvelimet toteutetaan omina prosesseina. Pienille resursseille tyylin soveltaminen voi olla liioiteltua, sillä etäpalvelupyynnöt ovat raskaampia suorittaa kuin paikalliset rutiinikutsut. Asiakas-palvelin -arkkitehtuuri mahdollistaa virheiden käsittelyn itsenäisesti palvelun osapuolissa, mutta esimerkiksi asiakkaiden tulee varautua palveli-



Kuva 3.10: Kuvaus viestinvälitysarkkitehtuurista, jossa komponentit ovat rekisteröityneet viestinvälittäjälle. Katkoviivalla on merkitty erään viestin reittiä komponentilta toiselle. [102]

men kaatumiseen tai käyttökatkokseen. [102]

Koskimies ja Mikkonen [102] pitävät asiakas-palvelin -tyyliä mahdollisesti yleisimpänä arkkitehtuurimallina. Resurssien, kuten tietokantojen ja levyasemien, lisäksi mallia on sovellettu palveluiden jakamiseen. Esimerkiksi Symbian OS -käyttöjärjestelmässä resurssien ja palveluiden käyttö on kapseloitu palvelimien taakse. Käyttöjärjestelmästä löytyykin niin tiedosto-, socket- kuin fonttipalvelin. [129]

Viestinvälitysarkkitehtuuri

Viestinvälitysarkkitehtuuri (engl. *Message dispatcher architecture* ja *Message bus architecture*) soveltuu järjestelmille, joissa tuntematon määrä erilaisia komponentteja kommunikoi keskenään. Tyylin keskiössä toimii viestinvälittäjä, joka tulkitsee viestin standardimuotoisen osoitetiedon ja välittää sen vastaanottajille. Postinjakajan ainoa tarkoitus on välittää viestejä, eikä se ole kiinnostunut asiakkaiden toiminnallisuudesta tai viestien sisällöistä. [102] Kuvassa 3.10 viestinvälitysarkkitehtuurin kuvaus, jossa yksittäisen viestin kulkua komponentilta toiselle on merkitty katkoviivalla.

Tyylissä komponenteilla on yhteinen rajapinta, jonka kautta ne vastaanottavat viestejä. Viesteille on staattinen rajapinta niiden välittämiseksi ja käsittelemiseksi. Komponentteja ja viestejä voidaan lisätä järjestelmään helposti, kunhan ne toteuttavat vaadittavat metodit.

Uudentyyppiset viestit pystyvät muuttamaan järjestelmän toiminnallisuutta huomattavasti, ilman muutoksia viestinvälittäjään tai muihin komponentteihin. Tyylissä komponentteja voidaan vielä lisätä ja poistaa ajonaikana järjestelmän siitä kärsimättä. [102]

Arkkitehtuurityyli mahdollistaa viestien vastaanottamisen synkronisesti ja asynkronisesti. Poikkeustilanteissa voidaan lähettää komponenttien välillä erityisiä virheviestejä. Tyylin suurin hyöty on komponenttien vapaa määrä ja laatu, mikä tosin haittaa järjestelmän ymmärrettävyyttä. Virheiden syiden etsiminen vaihtelevasta määrästä komponentteja on hankalaa. Virhetilanteissa vaikuttavat myös viestien sisältö ja rakenne, mikä vaikeuttaa ongelmien paikallistamista. Viestien välittäminen, lukeminen ja kirjoittaminen heikentävät myös järjestelmän tehokkuuteen. [102]

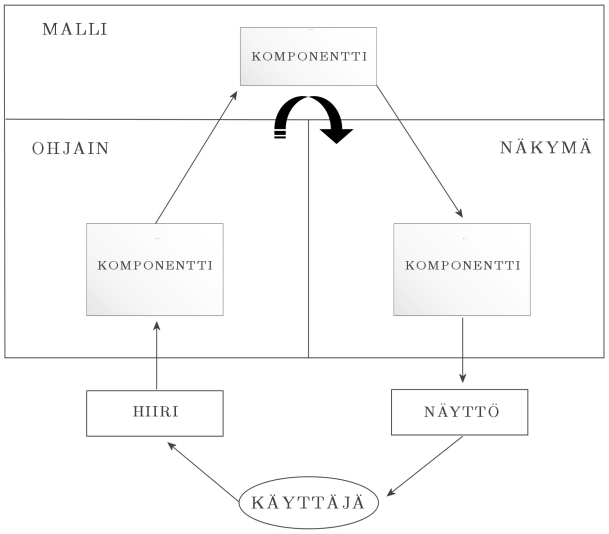
3.2.3 Sovellusalaperustaiset tyylit

Koskimiehen ja Mikkosen [102] kolmas kategoria on sovellusalaperustaiset arkkitehtuurityylit, joihin kuuluvat sovellusalansa standarditoteutuksiksi muodostuneet arkkitehtuurimallit. Esimerkiksi Malli-näkymä-ohjain on yleinen tapa rakentaa helposti muokattava käyttöliittymä, joko koko järjestelmän tai yksittäisen komponentin sisäisenä arkkitehtuurina. Sovellusalaperustaiset tyylit voivat olla hyvinkin erikoistettuja. Esimerkiksi tulkkiarkkitehtuuri, jossa tulkki lukee toiminallista kuvausta ja suorittaa sitä jonkin toteutusalustan päällä. [102]

Malli-näkymä-ohjain -arkkitehtuuri

Malli-näkymä-ohjain -arkkitehtuuri (engl. *Model-View-Control*, MVC) on perinteinen ratkaisu käyttöliittymän erottamiseksi sovelluslogiikasta ja -informaatiosta. Suunnittelutyyli mahdollistaa useiden erilaisten, ajan tasalla pysyvien näkymien rakentamisen saman tietosisällön ympärille. Erottamalla käyttöliittymä sovelluslogiikasta mahdollistetaan myös siirrettävyys toiselle graafiselle alustalle. [102]

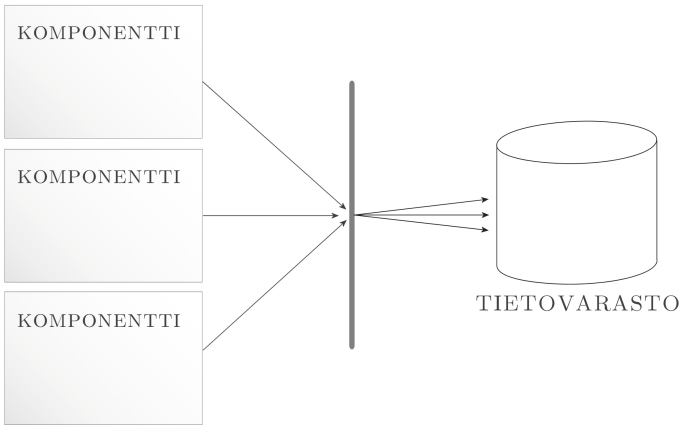
MVC-arkkitehtuuriin kuuluvat jäsenkomponentit jaetaan sovelluslogiikan mukaisiin



Kuva 3.11: Eräs Malli-näkymä-ohjain -arkkitehtuurin kuvaus, jossa näkymän komponentit piirtävät tietokoneen näytölle ja ohjain seuraa käyttäjän syötteitä. Nuoli kuvaa muutosten kulkemista järjestelmässä. [151]

malleihin, käyttöliittymän *näkymiin* ja näiden välisinä sovittajina toimiviin *ohjaimiin*. Sovellusalueen mallien velvollisuutena on ilmoittaa näkymille tilanmuutoksista, näkymät pitävät käyttäjän ajan tasalla ja ohjaimet muuttavat käyttäjän komennot sovellusalueen käskyiksi. [151] Kuvassa 3.11 on esitetty kaaviona MVC-arkkitehtuurin toiminta järjestelmässä, jossa käyttäjä saa järjestelmän tulosteen näytölle ja ohjaa toimintaa hiirellä.

Tyyli mahdollistaa mallien uudelleenkäyttämisen ja käyttöliittymän helpon muunneltavuuden. Tosin se myös monimutkaistaa järjestelmää ja lisää tarvittavien luokkien määrää. Mallit saattavat myös lähettää useita päivityspyyntöjä näkymille, joita muutokset eivät koske. Näkymäkomponenttien tekemät turhat tarkistuspyynnöt kuluttavat ylimääräistä laskenta-aikaa. Samoin tehoja kuluu useiden näkymien pyytäessä päivitetty sovellusdataa samanaikaisesti. Vaikka mallit ovat uudelleenkäytettäviä, niin ohjaimien ja näkymien käyttäminen uudelleen yhdessä tai erikseen on haastavaa niiden sovelluskohtaisuuden vuoksi. [102]



Kuva 3.12: Tietovarastoarkkitehtuurin hahmotelma, jossa komponentit käyttävät tietovarastoa yhteisen rajapinnan kautta.

Tietovarastoarkkitehtuuri

Tietovarastoarkkitehtuuri (engl. *Repository architecture*) on periaatteiltaan vain Asiakas-palvelin -mallin erikoistus. [102] Arkkitehtuurityylissä itsenäiset komponentit ja järjestelmät ylläpitävät tilaa yhteisessä, keskitetyssä tietovarastossa. Järjestelmän jäsenet eivät kommunikoi suoraan, vaan ainoastaan tietovaraston kautta. [69] Kuvassa 3.12 on esitetty arkkitehtuurityyli kaaviona.

Tyyli tukee hajautettuja ja rinnakkaisia järjestelmiä. Uusien komponenttien lisääminen ja poistaminen on helppoa, sillä ne ovat riippuvaisia ainoastaan tietovarastosta. Järjestelmän komponentit hajauttaa eri prosessoreille riippuvuuksien puuttumisen vuoksi. Tietovarastoarkkitehtuurin etuna kaikilla komponenteilla on käytössä sama informaatio. Tosin keskeisen varaston rajapinnan muuttaminen on vaikeaa, sillä kaikki ulkopuoliset komponentit ovat riippuvaisia siitä. Pienetkin muutokset vaatisivat koko järjestelmän refaktoroimista. Tämä hankaloittaa resurssin rajapinnan suunnittelua, sillä sen pitäisi olla tarpeeksi tehokas, mutta myös yleinen myöhemmin lisättävien komponenttien tarpeisiin. Arkkitehtuurityyliin ongelma on mahdollinen tehottomuus, sillä ulkoiset komponentit joutuvat jatkuvasti odottamaan tietovaraston sisältöä verkon yli ja jokainen komponentti voi joutua tulkitsemaan tiedon itselleen sopivaan muotoon. [102]

Tietovarastoarkkitehtuuria käytetään erityisesti ohjelmointiympäristöissä [69], sekä teksti- ja julkaisujärjestelmissä [102]. Näissä sovelmissa järjestelmän työkalut käsittelevät keskeiseen tietovaraston dataa ja sen esitysmuotoa. Monet liiketoimintajärjestelmät, kuten laskutus- ja varastosovelmat, pohjautuvat myös tietovarastoarkkitehtuurin. [102]

3.3 Suunnittelu- ja antisuunnittelumallit

Gamman, Helmin, Johnsonin ja Vlissidesin määritelmän mukaan suunnittelumallit (engl. *Design patterns*) ovat "*kuvauksia keskenään vuorovaikutuksessa olevista olioista ja luokista, jotka on muotoiltu ratkaisemaan tietyssä yhteydessä esiintyvä yhteinen suunnitteluongelma*". [68, s. 3] Suunnittelumallit ovat sekä hyväksi havaittuja että käytettyjä ratkaisuja. Esimerkiksi Gamma ym. hyväksyivät kirjaansa ainoastaan malleja, joita oli käytetty useammin kuin kerran onnistuneesti eri sovelluksissa.

Arkkitehtuurityylien ja suunnittelumallien välinen ero on hyvin pieni. Arkkitehtuurityylit voidaan nähdä suunnittelumallien idean yleistyksenä koko järjestelmälle. [69] Esimerkiksi MVC -suunnittelua nimitetään sekä arkkitehtuurityyliksi että suunnittelumalliksi riippuen käyttökohteesta. Tyylit ja mallit voidaan kuitenkin erottaa koon ja esiintymistiheyden mukaan. [102] Järjestelmällä on vain yksi määräävä tyyli, mutta suunnittelumalleja voi esiintyä useita. Arkkitehtuurityyli koskee kaikkia järjestelmän jäseniä, mutta suunnittelumalli käsittää yleensä vain muutaman luokan tai komponentin väliset suhteet.

Antisuunnittelumallit (engl. *Anti-patterns*) ovat edellisten vastakohtia, yleisesti esiintyviä huonoja ratkaisuja ja toimintatapoja. Ne ovat suunnittelumalleja huomattavasti laajempi käsite, sillä antimalleja on tunnistettu myös projektin hallinnasta ja suunnittelusta. [38] Huonoja toimintatapoja on useita, ja antimallien kuvaukset ovatkin huomattavasti yleisempiä ja vähemmän yksityiskohtaisempia kuin suunnittelumallien. Antimallien idea on parantaa järjestelmän laatua tunnistamalla sitä heikentäviä tekijöitä. Tämän vuoksi antimallien kuvaukseen kuuluu ehdotus ongelman korjaamiseksi. [102]

3.3.1 Suunnittelumallit

Gamman ym. [68] esittelemät 23 suunnittelumallia keskittyivät matalan tason suunnitteluun, luokkien ja olioiden välisiin suhteisiin. He antoivat myös klassisen kaksiakselisen jaon suunnittelumalleille. Mallit voidaan jakaa toiminnallisuuden perusteella kolmeen kategoriaan: luontimalleihin, rakennemalleihin ja käyttäytymismalleihin. Suunnittelumallit voidaan jakaa vaikutuskohteen perusteella olio- tai luokkamalleihin. Seuraavaksi esiteltävät suunnittelumallit on luokiteltu Gamman ym. kategorioissa taulukossa 3.1.

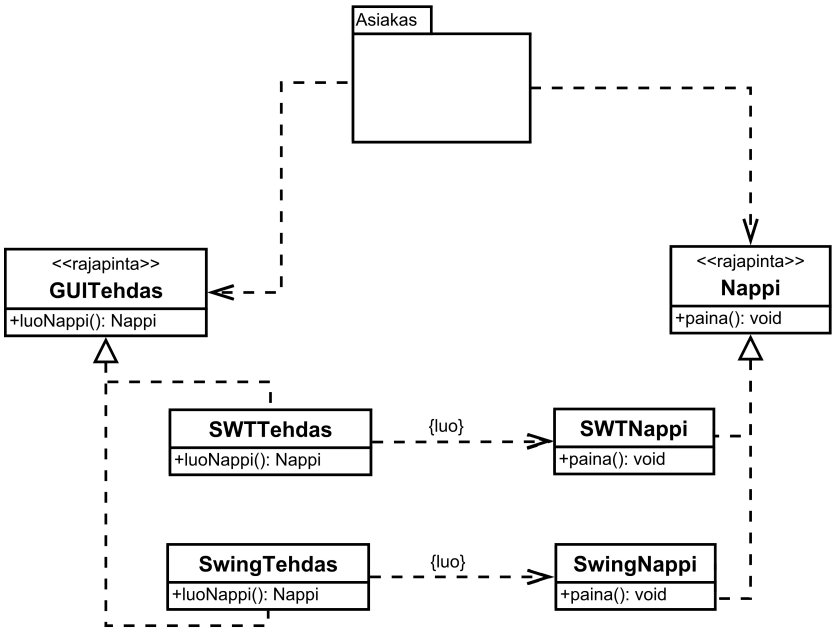
Esimerkkinä annettavat suunnittelumallit *Abstrakti tehdas*, *Sovitin* ja *Välittäjä* ovat esitetty luokkien ja olioiden välisten suhteiden hallitsemiseksi. Niiden taustalla olevia ratkaisuja voidaan kuitenkin soveltaa myös komponenttien tai luokkajoukkojen vastaavissa ongelmissa. Esimerkiksi *Abstrakti tehdas* poistaa luokkien välisiä suoria riippuvuuksia ja vastaavasti komponenttien riippuvuuksia voidaan poistaa suunnittelumallin yleistyksellä arkkitehtuuritasolle.

Abstrakti tehdas (engl. *Abstract Factor*). Suunnittelumalli soveltuu järjestelmiin, joissa on useita tuoteperheitä, mutta vain yksi voi olla käytössä kerrallaan. Abstraktin tehtaan avulla voidaan luoda tuoteperheen olioita, tietämättä mikä konkreetti luokka on kyseessä. Tehdas kuuluu selvästi luonti- ja oliomalleihin, sillä sen avulla hallitaan olioiden luontia. Kuvassa 3.13 on hahmoteltu abstraktin tehtaan erästä toteutusta. [68]

Suunnittelumallin etu on konkreettien luokkien piilottaminen ja tuoteperheen helppo vaihdettavuus. Abstrakti tehdas myös valvoo, että vain yhden tuoteperheen olioita käytetään samanaikaisesti. Uusien tuotteiden lisääminen on suunnittelumallissa

Taulukko 3.1: Esimerkkisuunnittelumallien sijainti Gamman ym. luokittelussa.

	Luontimalli	**Rakennemalli**	**Käyttäytymismalli**
Luokkamalli		Sovitin	
Oliomalli	Abstrakti tehdas		Välittäjä



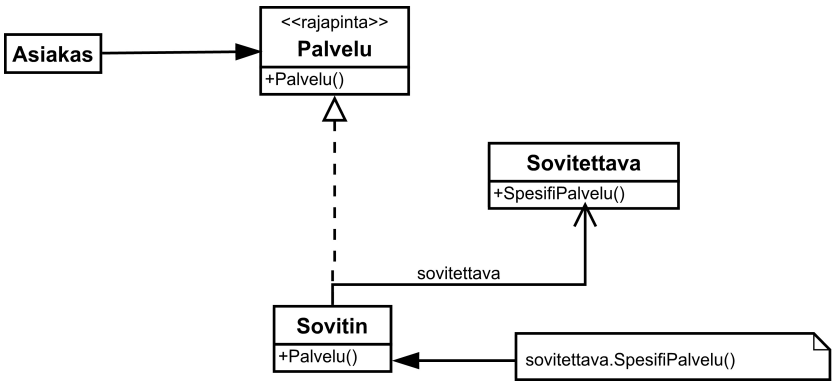
Kuva 3.13: Abstraktitehdas -suunnittelumalli, jossa `GUITehdas` luo käyttöliittymäkomponentteja kuten painonappeja. `SWTTehdas` ja `SwingTehdas` luovat konkreettiset oliot. Kuvassa asiakas luo `Nappi`-rajapinnan olioita abstraktin tehtaan kautta. [68]

työlästä, sillä tehtaan rajapintaa ja sen kaikkia toteuttajia pitää päivittää. [68]

Sovitin (engl. *Adapter*). Suunnittelumalli muuntaa luokan rajapinnan järjestelmän tarpeisiin sopivaksi. Tämä mahdollistaa suoranaan yhteen sopimattomien luokkien käyttämisen yhdessä. Sovitin tukee uudelleenkäytettävyyttä, sillä se mahdollistaa olemassa olevien luokkien käyttämisen, vaikka niiden rajapinta ei olisi suoraan soveltuva. [68] Kuvassa 3.14 on esimerkki mallin käytöstä. Sovitin-suunnittelumallista on olemassa myös olioversio, mutta sitä ei käsitellä tässä opinnäytteessä.

Välittäjä (engl. *Mediator*). Suunnittelumallia käytetään ison luokkajoukon keskinäisen kommunikoinnin yksinkertaistamisessa. Kuvassa 3.15 on esitetty Välittäjä-suunnittelumallin yleinen luokkakaavio. Kollegaoliot antavat viestinsä välittäjälle, joka toimittaa viestin toiselle kollegalle. Näin kollegoiden ei tarvitse tietää toisiaan. [68]

Suunnittelumalli korvaa monta-moneen –suhteet yksi-moneen –suhteilla, mikä yk-



Kuva 3.14: Sovitin-suunnittelumalli, jossa `Sovitin` toteuttaa rajapinnan `Palvelu`-rutiinin käyttämällä `Sovitettava`-luokan `SpesifiPalvelu`-rutiinia. [68]

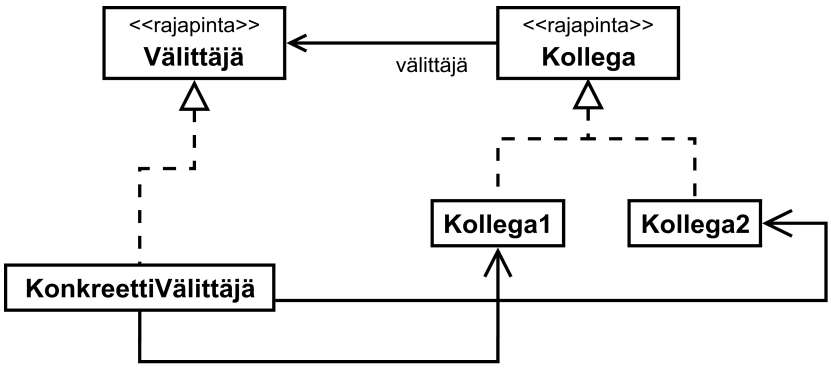
sinkertaistaa kollegaolioiden toteutetusta. Suunnittelumallin ongelmana on, että välittäjä-luokasta saattaa muodostua monimutkainen, monoliittinen ja vaikeasti ylläpidettävä. [68]

3.3.2 Antisuunnittelumallit

Brown, Malveu, McCormick ja Mowbray [38] jakoivat antisuunnittelumallien esiintymiset kolmeen kategoriaan: lähdekoodi, arkkitehtuuri ja projekti. Leikkaa ja liimaa – ohjelmointi (engl. *Cut-and-Paste Programming*) on klassinen esimerkki lähdekoodissa esiintyvästä antisuunnittelumallista. [102] Esimerkiksi projektinhallintaan liittyvä antitoimintamalli on *Kuoliaaksi suunnitteleminen* (engl. *Death by Planning*), missä pitkällinen suunnittelu myöhästyttää projektia ja tuottaa hyödyttömiä dokumentteja. Antisuunnittelumalleihin liittyy ratkaisutapa tunnistettujen toimintamallien kehittämiseksi. Esimerkiksi liiallista suunnittelua voidaan hallita iteratiivisella kehityksellä. [38]

Arkkitehtuurien antimalleja on lueteltu kymmeniä, käsittäen korkean tason suunnitteluvirheiden lisäksi vaiheeseen liittyviä huonoja toimintatapoja ja ratkaisuja. Muutamia Brownin ym. luettelemia antisuunnittelumalleja ovat [38]:

Valmistajariippuvuus (engl. *Vendor Lock-in*). Järjestelmässä on kriittinen riippuvaisuus ulkopuolisen valmistajan toimittamasta komponentista tai tekniikasta. Muutokset



Kuva 3.15: Välittäjä-suunnittelumalli, jossa `Kollega`-rajapinnan toteuttavat luokat käyttävät keskinäiseen viestintään `KonkreettiVälittäjä`-luokkaa. Välittäjä-malli vähentää luokkien välisiä riippuvuuksia ja parantaa näin kollegoiden uudelleenkäytettävyyttä. [68]

komponentissa vaativat toistuvaa järjestelmän päivittämistä. Ratkaisu ongelmaan on eristää ulkopuolisen komponentin palvelut ylimääräisen kerroksen taakse.

Sveitsiläinen linkkuveitsi (engl. *Swiss Army Knife*). Komponentin rajapinnat ovat monimutkaisia ja liioiteltuja. Suunnittelijalta on puuttunut selkeä kuva komponentin tarkoituksesta, minkä vuoksi se yrittää palvella kaikki mahdollisia asiakkaita. Ongelman ratkaisuksi ehdotetaan erityisen profiilin luomista, mikä määrittelisi komponentin keskeisimmät metodit sekä käyttöesimerkin.

Komiteasuunnittelu (engl. *Design by Committee*). Järjestelmän suunnitelma on monimutkainen ja sisältää useita ylimääräisiä ominaisuuksia. Vaatimusten toteuttaminen käytettävillä resursseilla on mahdotonta tai niiden testaamiseen käytettävästä ajasta on karsittava. Keskeinen ongelma on yhteisen näkemyksen puuttuminen, jolloin jokainen esitetty idea yritetään toteuttaa. Ongelman ratkaisuehdotus on uudistaa arkkitehtuurisuunnittelua vähentämällä suunnittelijoiden määrää ja formalisoimalla suunnitteluprosessia.

Kyhäelmä (engl. *Stovepipe System*). Osajärjestelmien arkkitehtuuri on rakennettu *Ad hoc* -suunnittelun varaan ja ilmaantuvia ongelmia korjataan ilman johdonmukaisuutta

tai suunnitelmaa. Kyhäelmäjärjestelmää voidaan parantaa refaktoroimalla komponenttien palvelut abstraktien rajapintojen taakse.

Luku 4

Arkkitehtuuritason metriikat

Ohjelmistometriikat voidaan luokitella niiden käyttökohteen perusteella eri abstraktiotasoille. Tässä opinnäytteessä on jo esitelty joukko metodi- ja luokkametriikoita, jotka nimensä mukaisesti ovat käytettävissä metodi- tai luokkatasolla. Arkkitehtuuritasolta voidaan poikkeuksellisesti tunnistaa vielä kaksi erillistä abstraktiokerrosta: komponentti- ja järjestelmämetriikat.

Järjestelmämitat tarkastelevat sovelmaa yhtenä kokonaisuutena, jolle lasketaan yksittäisiä tunnuslukuja. Esimerkiksi Abreun [9] COF, joka laskee järjestelmän suhteellista kytkeytymistä, on yksi järjestelmämitoista. Komponenttimitat taas keskittyvät yksittäisen arkkitehtuurin osan ominaisuuksiin. Sivulla 46 esiteltyä tietovuohon pohjautuvaa ICP(SS):tä voidaan esimerkiksi käyttää komponenttimetriikkana.

Korkealla tasolla korostuu erityisesti Briandin ym. [33] painottama riippuvuus metriikan ja laatutekijöiden välillä. Käsitteet muuttavat abstrakteiksi, mutta silti uusien mittareiden pitäisi pystyä ennustamaan joko ulkoista laatua [32] tai suunnittelun ongelmakohtia [114, 115]. Arkkitehtuuritasolla mittojen määrittely vaatii erityistä huolellisuutta.

Tässä luvussa tarkastellaan ensin komponentin ja järjestelmän määritelmiä, minkä jälkeen esitellään arkkitehtuuritasolle määriteltyjä ohjelmistomittoja. Metriikoista tarkastellaan erityisesti Martinin [120], Ducassen ym. [57] sekä Ponision [144] mittoja. Luvun lopussa pohditaan arkkitehtuurin mitattavia ominaisuuksia ja näille soveltuvia mittoja.

4.1 Arkkitehtuuritason määritelmä ja osat

Taulukossa 4.1 on esitelty Briandin, Dalyn ja Wüstin määrittelemät viisi eri abstraktiotasoa koheesio- [32] ja kytkeytymismetriikoille [33]. Määrittelyalueiden mukainen jako soveltuu myös muille sisäisten ominaisuuksien tuotemetriikoille kuten esimerkiksi kompleksisuus- tai kokomitoille. Kaksi korkeinta tasoa — luokkajoukko ja järjestelmä — ovat arkkitehtuurimittojen kannalta mielenkiintoisimmat. Järjestelmämitat soveltuvat suoraan arkkitehtuuritasolle, mutta terminä luokkajoukko (engl. *Set of Classes*) ei ole tarpeeksi sitova määrittely arkkitehtuurimetriikoille.

Luokkajoukko voi tarkoittaa mielivaltaisesti valittuja järjestelmän luokkia, joilla ei ole yhdistäviä tekijöitä. Satunnaisten luokkien tai luokkajoukon ominaisuudet eivät ole kiinnostavia arkkitehtuurisuunnittelussa. Ohjelmistokomponentin osilla on taas tarkoitus ja syy yhtenäisyydelle, niitä yhdistää tavoite yhtenäisten palveluiden tuottamiseksi. [116] Näin komponentti on mielekkäämpi määrittelyalue arkkitehtuurimetriikoille kuin sekalaisten luokkien joukko.

Ei ole selvää tarkoittivatko Briand ym. [32] luokkajoukollaan komponenttia vastaavaa käsitettä vai joukkoa riippumattomia luokkia. Metriikoiden tarkan määrittelyn nimissä on tehtävä ero kahden käsitteen välille, sillä myös luokkajoukoille voidaan esitellä mielenkiintoisia mittoja. Esimerkiksi kahden tai useamman luokan välistä kytkeytymistä voidaan tarkastella vain joukkoina. Lisättäessä komponenttitaso Briandin ym. määrittelyalueisiin, sijoittuisi se järjestelmäkerroksen ja luokkajoukon välille.

4.1.1 Komponentti

Ohjelmistoarkkitehtuureista puhuttaessa komponentin määritelmä jää valitettavan epäselväksi. Yleisyys tosin sallii komponentin määritelmän vaivattoman liukumisen yksittäisestä luokasta kokonaiseen järjestelmään. Formaalisti ilmaistuille ohjelmistometriikoille epämääräisyydet tulevat kuitenkin ongelmallisiksi. Esimerkiksi satojen luokkien yh-

Taulukko 4.1: Briandin, Dalyn ja Wüstin metriikoiden abstraktiotasot.

Määrittelyalue	**Tasolle määriteltyjä mittoja**
Attribuutti	RIC [92]
Metodi	RMC [92]
Luokka	LCOM, WMC
Luokkajoukko	ICP(SS)
Järjestelmä	COF

Briand ym. [32, 33] ellei toisin mainittu.

tenäisyyden arvioiminen muutaman luokan komponentille tarkoitetulla koheesiomitalla antaisi huomattavan vääristyneitä tuloksia.

Abreu [3] määritteli mittojensa yhteydessä ohjelmistometriikoille muutamia reunaehtoja. Yhtenä seitsemästä kriteeristä hän vaati metriikoiden helppoa automatisoitavuutta. Mitta siirtyy akateemisesta tarkastelusta teollisuuden käyttöön vain, jos sen laskenta on helposti toistettavissa ja tapahtuu kohtuullisessa ajassa. Ohjelmakomponentin tulee tämän vaatimuksen mukaan olla ohjelmallisesti helposti ja toistettavasti tunnistettavissa.

Tässä opinnäytteessä komponentti tullaan hahmottamaan ohjelmointikielten antamien abstraktioiden mukaisesti. Esimerkiksi Javassa komponentin osaksi tulkitaan `package`-avainsanalla merkityt, samaan pakettiin kuuluvat luokat ja rajapinnat. Javan paketit voivat koostua muista paketeista, mutta samoin suurilla komponenteilla voi olla oma arkkitehtuuri ja sisäinen rakenne alikomponentteineen. [102] Ongelmallista määritelmässä on, etteivät kaikki ohjelmointikielet tue Javan pakettien kaltaisia rakenteita. Tällaiset kielet kuitenkin käsitellään tapauskohtaisesti tämän opinnäytteen esimerkeissä, ja valittu komponentin rajaus ilmoitetaan erikseen.

Ohjelmistokomponentti koostuu luokkien ja rajapintojen määrittelyistä [144], mutta kaikki osat eivät ole jokaiselle mitalle samanveroisia. Pitäisikö esimerkiksi Javassa laskea komponentin luokkien määrään mukaan sisä- tai anonyymit luokat? Erityisesti käyt-

töliittymäkomponenttien luokkamäärät moninkertaistuisivat jos nimettömän luokat huomioitaisiin kokonaismäärässä. Toisaalta sisäluokkien yleinen unohtaminen komponenttimetriikoissa vääristäisi kytkeytymismittojen antamaa kokonaiskuvaa komponentin riippuvuuksista.

Ohjelmistokomponenteille ja luokille voidaan antaa *rooleja*, joiden pohjalta metriikoiden tarkastelualue voidaan määritellä. Esimerkiksi kokomitat huomioisivat ainoastaan ulkoluokan roolin mukaiset luokat. Roolit eivät ole toisiaan poissulkevia, vaan yksittäisellä luokalla voi olla useita rooleja. Esimerkiksi MVC-arkkitehtuurissa luokka yleisesti toteuttaa sekä ohjaimen että näkymän vastuut. Vastaavasti tällaiselle luokalla olisi määritelty kummatkin roolit. Laskennan automatisoitavuuden vuoksi roolit tulisi tunnistaa staattisessa analyysissa joko luokan sisäisestä rakenteesta tai sen asiakkaiden perusteella. Esimerkiksi palveluntarjoan roolissa luokkaan kohdistuvia riippuvuuksia on huomattavasti enemmän kuin lähteviä.

4.1.2 Järjestelmä

Nykypäivän suuret ohjelmistot koostuvat miljoonista riveistä ja tuhansien ihmisten työmäärästä. Esimerkiksi, maailman suurimmaksi ohjelmaksi arvioidussa [16], Debian 4.0 -käyttöjärjestelmässä on $10\ 106$ komponenttia, jotka koostuvat yhteensä $288,5$ miljoonasta fyysisestä lähdekoodirivistä. [70] Informaatiotulva niin ihmiselle kuin metriikoille on valtaisa, ja järjestelmää kokonaisuudessa arvioivien mittojen näkemiä yksityiskohtia tulisikin rajoittaa. Ponisio [144] ja Abdelmoez ym. [2] ovat kummatkin ehdottaneet korkeampien abstraktiotasojen näkemien yksityiskohtien rajoittamista, ja ehdotukset muistuttavat Lakshmi Narasimhan ja Hendradjayan [106] esittelemää graafimallia.

Ohjelmistoa voidaan mallintaa suunnattuna graafina, jossa soveltuvat elementit toimivat solmuina ja kaaret ovat riippuvuuksia näiden välillä. [126] Järjestelmätasolla solmuja olisivat ohjelmakomponentit [106], 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. Tällaisessa laskennassa ylimääräisen informaation karsiminen nopeuttaa tuloksen saamista.

Toinen saavutettava hyöty on vastaavuus UML:n luokka- ja komponenttikaavioiden kanssa. Luokkakaavio kuvaa ohjelman tai komponentin rakennetta luokilla ja olioilla, ja kaaviossa kahden luokan relaatio on kuvattu ilman tietoa fyysisestä toteutuksesta. [39] Vastaavasti komponentti- tai pakettikaavioissa järjestelmää mallinnetaan vain komponenteilla tai paketeilla, jotka kuvaavat ryhmiteltyjä luokkia tai käyttötapauksia. [15] Abstraktiotasojen yhteneväisyyden perusteella voidaan metriikoiden laskenta automatisoida suunnittelutyökaluihin, ja ensimmäiset 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 olisi esimerkiksi yksittäisten metodikutsujen tai viittauksen summat. Eri relaatiotyypeille, kuten perintä ja viittaussuhteille voitaisiin määritellä omat painofunktiot tai käyttää hypergraafia.

Järjestelmällä tarkoitetaan tässä opinnäytteessä joko ohjelmiston kaikkia komponentteja tai soveltuvaa osaa niistä. *Alijärjestelmä* on joukko komponentteja, jotka toteuttavat yhdessä määritellyn toiminnallisuuden. Suurempi järjestelmä voidaan jakaa useammaksi alijärjestelmäksi, vastaavasti kuin monimutkaisen rutiinin jakaminen alirutiineiksi. Järjestelmälle esiteltävät mitat soveltuvat myös alijärjestelmissä käytettäviksi.

4.2 Arkkitehtuurimetriikat

Briand ja kollegat [32, 33] väittävät toistuvasti alemman abstraktiotason metriikoiden, kuten luokkatason koheesiomittojen, skaalautuvan suoraan ylemmille tasoille kuten komponentille ja järjestelmälle. Esimerkiksi lähdekoodirivien lukumäärä on ohjelmistomitta, jota voidaan soveltaa lähes kaikilla tasoilla. Ohjelmakoko on kuitenkin helppo, tarkastelutasolta toiselle muuttumaton ja helposti ymmärrettävä käsite. Briandin ym. lähestymistavan suoraviivaista yleistämisestä kuitenkin kritisoitiin tässä opinnäytteessä jo ICP:n yhteydessä sivulla 49. Yksinkertainen vastaesimerkki kiistää Briandin ym. väitteen yleistämisen, ja jokaisen mitan kohdalla on abstraktiotason korottamisen tarkastelu suoritettava erikseen.

Komponenttitasolle on määritelty myös muutamia mittoja. Opinnäytteessä esitellään näistä tunnetuin, kehittäjänsä mukaan nimetty Martinin metriikka [120], sekä Ducassen ym. [57] ja Ponision [144] ohjelmakomponenttien visualisoimisen avuksi kehittämät mitat. Viimeisenä esitellään lyhyesti kumulatiivinen riippuvuus [105], Hautuksen [79] komponenttien laatumitta sekä Meltonin ja Temperon [124] CRSS. Luvussa 4.2.4 on listattu lyhyesti muita kirjallisuudessa esitetty komponenttitasolle soveltuvia mittoja.

4.2.1 Martinin metriikka

Robert C. Martin [116] esitteli yhden ensimmäisistä komponenttimetriikoista jo vuonna 1994. Martinin metriikan (engl. *Martin's metric*) kehittäjä kutsui mittaa suorasukaisesti laatumitaksi [116], vaikka mitta arvioi ainoastaan komponentin *abstraktiuden* ja *stabiiliuden* suhdetta. Martin ei myöskään yrittänyt osoittaa yhteyttä metriikan ja ulkoisten laatutekijöiden välille, mitä Briand ym. [32] korostivat metriikkasuunnittelun lähtökohdaksi.

Metriikka perustuu kahteen Martinin esittelemään ohjelmakomponentin suunnittelun ohjenuoraan. Näistä ensimmäisen, *stabiilin riippuvuuden periaatteen* (engl. *Stable Dependencies Principle*, SDP) mukaan komponentin pitäisi olla riippuvainen ainoastaan

sitä vakaammista komponenteista. Epästabiilien osien muuttuessa niistä riippuvaisia komponentteja joudutaan muuttamaan ja päivittämään yhteensopiviksi uudempien versioiden kanssa. [118] Tämän vuoksi komponentin tulisi olla riippuvainen vain vakaista palveluista, jolloin komponenttia ei tarvitse refaktoroida jatkuvasti palveluntarjoajan muuttuessa.

Stabiilius ei tarkoita Martinille niinkään ohjelmakomponentin muutosten epätodennäköisyyttä, vaan muutoksen tekemisen vaativuutta. [118] Hän määritteli stabiilin komponentin sellaiseksi, jolla on paljon vastuuta ja vain vähän syitä muuttua. [119] Jokainen luokka, joka on riippuvainen komponentin palveluista lisää komponentin vastuita. Vastaavasti jokainen luokka, josta komponentti on riippuvainen on mahdollinen syy muutokselle.

Martin [116] lähestyy vakauden mittaamista käänteisesti määrittelemällä spatiaalisen epästabiiliuden mitan, joka pohjautuu komponenttiin tuleviin ja lähteviin riippuvuuksiin:

$$C_a \text{ (engl. } \textit{Afferent Coupling}) = \text{Komponentin ulkopuolisten luokkien lukumäärä, jotka ovat riippuvaisia moduulin sisäisistä luokista.} \quad (4.1)$$

$$C_e \text{ (engl. } \textit{Efferent Coupling}) = \text{Komponentin ulkopuolisten luokkien lukumäärä, joista moduulin sisäiset luokat ovat riippuvaisia.} \quad (4.2)$$

$$I \text{ (engl. } \textit{Instability}) = \frac{C_e}{C_a + C_e} \quad (4.3)$$

Epästabiiliuden, I:n, arvot vaihtelevat välillä $[0, 1]$. Arvolla nolla komponentti on stabiili, sillä se ei ole riippuvainen yhdestäkään toisesta paketista. Vastaavasti arvolla yksi, komponenttiin tulevia riippuvuuksia ja vastuita ei ole, jolloin komponentti on epästabiili.

Martinin toisen ohjenuora, *vakaan abstraktiuden periaate* (engl. *Stable Abstractions Principle*, SAP), vaatii stabiileista komponenteista abstrakteja. SDP:n perusteella rakennettujen vakaiden komponenttien muuttamisesta on tehty vaikeata, sillä näistä riippuvia komponentteja on paljon ja niiden refaktoroiminen työlästä. Silti komponenttien palve-

luita pitäisi pystyä laajentamaan. Martinin ehdottama ratkaisu on tehdä stabiileista komponenteista abstrakteja. Tällaisten komponenttien rajapintoja on helppo kasvattaa, sillä komponentin palvelut eivät voi laajennuksesta muuttua. [119]

Abstraktisuudelle määriteltiin myös oma mitta [119]:

$$N_c \text{ (engl. } \textit{Number of Classes in the Package}) = \text{Komponentin luokkien lukumäärä} \quad (4.4)$$

$$N_a \text{ (engl. } \textit{Number of Abstract Classes in the Package)} = \text{Komponentin abstraktien luokkien lukumäärä.} \quad (4.5)$$

$$A \text{ (engl. } \textit{Abstractness}) = \frac{N_a}{N_c} \quad (4.6)$$

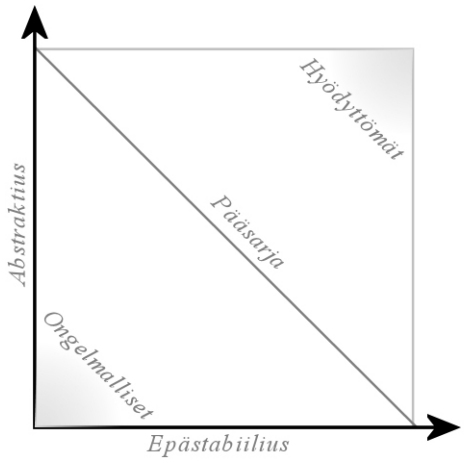
Abstraktisuuden, A:n, arvot ovat myös skaalattu välille $[0, 1]$, missä arvolla nolla komponentissa ei ole yhtään abstraktia luokkaa. Komponentin kaikkien luokkien ollessa abstrakteja saa A arvon yksi. Abstraktin luokan määritelmä on kieliriippuvainen, esimerkiksi Javassa N_a:n arvoon voidaan huomioida sekä rajapinnat että abstraktit luokat. [171]

Martinin metriikan arvo komponentille p on järjestetty pari (I_p, A_p). [116] Kuvassa 4.1 on esitelty karteesinen IA-koordinaatisto, johon sijoittamalla komponenttien arvoja metriikkaa voidaan havainnollistaa. Martinin [116] teorian mukaan pisteiden $(1, 0)$ ja $(0, 1)$ välille piirretyn suoran eli *pääsarjan* läheisyyteen sijoittuvat komponentit ovat tasapainossa vastuiden ja velvoitteiden suhteen. Suoran ääripisteet edustavat SDP:n ja SAP:n mukaisia suunnitteluja: Pisteessä $(1, 0)$ komponentti on täysin abstrakti ja vakaa. Periaatteiden mukaan kaikki riippuvuudet tulisi suunnata tällaisiin komponentteihin. Pisteessä $(0, 1)$ komponentti on taas täysin konkreetti ja epästabiili, eikä sillä olisi vastuita muille komponenteille.

Komponenttien suunnitteluperiaatteiden mukaiset ääripäät ovat harvinaisia tilanteita, joten Martin määritteli mitan etäisyydelle suorasta [116]:

$$D \text{ (engl. } \textit{Distance}) = \frac{|A + I - 1|}{\sqrt{2}} \quad (4.7)$$

$$D' \text{ (engl. } \textit{Normalized Distance}) = |A + I - 1| \quad (4.8)$$



Kuva 4.1: Martinin metriikan IA-koordinaatisto. Järjestelmää voidaan kuvata sijoittamalla sen komponentit koordinaatistoon. [120]

Kaavan 4.7 etäisyys vaihtelee välillä $[0, \frac{\sqrt{2}}{2}]$ kun kaavan 4.8 normalisoitu arvo on välillä $[0, 1]$. Etäisyyden arvolla nolla komponentti on IA-kuvaajan diagonaalilla. Mittojen yläraja taas kuvaa tilannetta, jossa komponentti on hyödyttömällä (engl. *Zone of Uselessness*) tai ongelmallisella (engl. *Zone of Pain*) alueella. [120]

Lähellä pistettä $(1, 1)$ sijaitsevat komponentit ovat abstrakteja ja niistä riippuvia luokkia on vain muutama. Selvästi tällaiset komponentit ovat hyödyttömässä asemassa arkkitehtuurissa, sillä niiden abstraktisuus lisää kompleksisuutta. Samalla komponenttien yleisyyden tuomia etuja ei hyödynnetä. Samoin pisteen $(0, 0)$ läheiset komponentit edustavat Martinin mielestä huonoa suunnittelua. Nämä ohjelmakomponentit ovat täysin konkreettisia ja samalla vastuussa monelle taholle. Martinin perustelee niiden muuttamisen olevan työlästä suuren vastuun takia ja konkreettisuuden tekevän niiden laajentamisesta hankalaa. [119]

Martinin metriikan suurin ongelma on, ettei määrittelyjä ole annettu formaalisti. Sanallinen kuvaus ei tarkenna mitä riippuvuuksia C_a:n ja C_e:n laskennassa pitäisi huomioida. [33] Esimerkiksi epäselväksi jää onko perimissuhde samanarvoinen kuin käyttösuhde, tai pitäisikö argumenttien tyypit huomioida riippuvuussuhteeksi? Formalismin puuttuessa mittojen teoreettista taustaa ei ole testattu. Esimerkiksi Briand ym. [35] esittelivät viisi

kytkeytymisen teoriaan pohjautuvaa väitettä, jotka mittaehdokkaiden tulisi läpäistä. Tällä voidaan todistaa metriikoiden mittaavan väitettävää ominaisuutta.

Toinen keskeinen ongelma on, ettei Martin tarjoa metriikan hyödyistä selkeää kuvaa. Mitat avustavat suunnittelunuorien noudattamista, mutta periaatteet edustavat suunnittelun puhtaita ääripäitä. Martin [119, 120] itse vakuuttaa käyttäneensä mittaa useasti menestyksellä komponenttien riippuvuuksia selvitellessään, mutta ei tarjoa tilastollista analyysiä metriikan hyödyllisyydestä. Mitoille on tosin annettu käyttöohje. Huonoja arvoja saaneita komponentteja voidaan refaktoroida kääntämällä riippuvuussuhteet rajapintojen avulla tai jakaa komponentti abstraktiin ja konkreettiin osaan. [120]

Martin [120] antaa lyhyesti esimerkin, jossa hän refaktoroi 15:sta komponentin *ad hoc* -arkkitehtuurin metriikoiden avulla. Neljän komponentin arvot poikkesivat huomattavasti pääsarjasta. Näiden normalisoitu etäisyys oli välillä 0.67—$0,86$. Luokkien siirron ja riippuvuuksien kääntämisen avulla Martin vähensi komponenttien määrän kahteentoista, jolloin suurimmaksi normalisoiduksi etäisyydeksi jäi 0.25. Esimerkissä metriikoiden arvot laskettiin käsin, mutta Martin suositteli automaattisen laskennan tarkastelemaan C++:n `#include`- tai Javan `import`–lauseita.

Martin määritteli myöhemmin alkuperäiseen pakettiin kuulumattoman koheesiomitan [120]:

$$H \text{ (engl. } \textit{Relational Cohesion}) = \frac{R+1}{N_c} \tag{4.9}$$

Missä R on komponentin luokkien välisten riippuvuuksien lukumäärä. Mitta laskee siis komponentin relaatioiden määrän luokkaa kohti. Osoittajan ylimääräisen yhteenlaskun tarkoituksena on estää H:n arvo 0 kun komponentissa on vain yksi luokka. [120] Mitassa on kuitenkin muutama huomattava ongelma. Martin ei ohjeistanut luokkien riippuvuuksien laskemisessa. Epäselväksi jää niin kytkeytymisehto kuin kytkösten suunnan huomioiminen. Jos komponentin molemmat luokat käyttävät toisiaan, onko riippuvuuksien lukumäärä 1 vai 2? Mitalla ei myöskään ole Briand ym. [32] edellyttämiä kiinteitä ylä- tai alarajoja. H ei esimerkiksi voi saada koskaan arvoksi nollaa.

4.2.2 Perhoskuvaajat

Ducassen, Lanzan ja Ponision [56, 57] lähestymistapa ohjelmakomponenttien mittaamiseen on huomattavan erilainen, sillä heidän ensisijaisena tarkoituksenaan oli määritellä apuvälineitä järjestelmän visualisointiin. Ohjelmakomponentit ovat monimutkaisia, useista osista koostuvia rakenteita, joiden ymmärtäminen vaatii tutustumista useisiin lähdekooditiedostoihin. Ducasse ym. esittelivät ohjelmakomponenttien ymmärtämiseksi kaksi yksinkertaisiin metriikoihin perustuvaa *perhoskuvaajaa* (engl. *Butterfly*). [56] Esitetyt kuvaajat ja metriikat on tarkoitettu erityisesti jo olemassa olevan järjestelmän refaktoroinnin apuvälineeksi. [57] Perhoskuvaajat ovat laskettavissa vasta implementaatiovaiheessa, kun esimerkiksi Martinin metriikan alustavia arvoja saadaan jo korkean tason suunnitelmista.

Absoluuttinen perhonen (engl. *Global Butterflyn*), jota on hahmoteltu säteittäiskaaviolla kuvassa 4.2, on kontekstiriippuvainen. Sen mitat ovat absoluuttisella asteikolla ja niiden arvot lasketaan käyttäen koko järjestelmää. *Suhteellinen perhonen* (engl. *Relative Butterflyn*) on kuvaajaltaan vastaava, mutta mitat ovat normalisoituja versiota absoluuttisen perhosen vastaavista. Perhosten vasen siipi kuvaa miten asiakkaat käyttävät komponenttia ja oikea siipi miten komponentti käyttää muiden palveluita. Kuvaajan alaosa osoittaa miten perintää on komponentissa käytetty. [56]

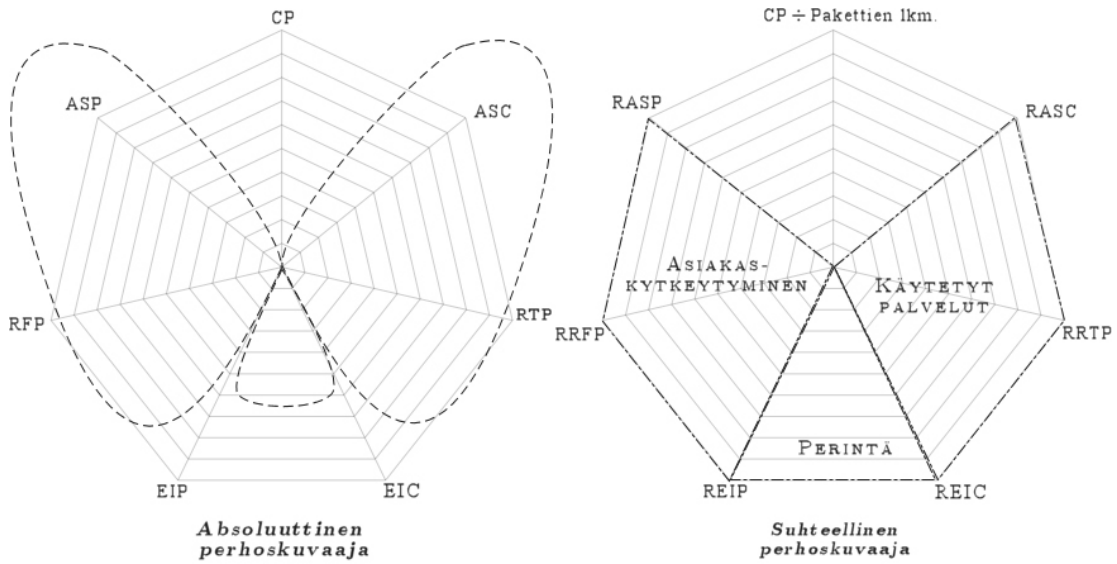
Opinnäytteessä ei tilarajoituksen vuoksi käsitellä mielenkiintoista ja tärkeätä ohjelmakomponenttien visualisointia, mutta perhoskuvaajissa käytetyt yksinkertaiset komponenttimitat on listattu taulukossa 4.2. Ducassen ym. metriikat perustuvat kolmeen luokkarelaatioon [56]: Ensimmäinen on *Perintä*, jossa luokka on toisen lapsi- tai yläluokka. *Luokkaviittauksessa* eksplisiittisesti hyödynnetään toista luokkaa. *Esivanhempien muuttujan käytössä* luokka käsittelee perittyjä instanssimuuttujia.

Ducasse ym. [57] väittävät valinneensa minimaaliset mitat olio-ohjelmoinnin komponenttien perimmäisen olemuksen esittämiseksi tiiviissä paketissa. Kuvaajat perustuvatkin hyvin yksinkertaisiin relaatioiden lukumääriä laskeviin mittoihin. Perintään ja viittauksiin perustuvat mitat ovat vastaavia mitä Martin käytti laskiessaan komponentin stabiiliutta.

Taulukko 4.2: Perhoskuvioissa esiintyvät komponenttimitat.

Metriikka		**Kuvaus**
PP	*Number of Provider Packages*	Käytettävien komponenttien lukumäärä
CP	*Number of Client Packages*	Asiakaskomponenttien lukumäärä
RTP	*Number of Class References to Other Packages*	Luokkaviittausten lukumäärä komponentin ulkopuolisiin luokkiin
RRTP	*Relative Number of Class References to Other Packages*	RTP jaettuna RTP:n ja sisäisten luokkaviittausten summalla
RFP	*Number of Class References from Other Packages*	Ulkoisista luokista komponenttiin tulevien luokkaviittausten määrä
RRFP	*Relative Number of Class References from Other Packages*	RFP jaettuna RFP:n ja sisäisten luokkaviittausten summalla
PIIR	*Number of Internal Inheritance Relationship*	Paketin sisäisten perintärelaatioiden lukumäärä
RPII	*Relative Number of Internal Inheritance Relationship*	$\frac{\text{PIIR}}{\text{PIIR}+\text{EIP}}$
EIC	*Number of External Inheritance as Client*	Perintärelaatioiden määrä, joissa yliluokka on komponentin ulkopuolella
EIP	*Number of External Inheritance as Provider*	Perintärelaatioiden määrä, joissa perijä on komponentin ulkopuolella
REIP	*Relative Number of External Inheritance as Provider*	$\frac{\text{EIP}}{\text{EIP}+\text{PIIR}}$
ASC	*Number of Ancestor State as Client*	Komponentin ulkopuolisten yliluokkien muuttujien käyttömäärä
RASC	*Relative Number of Ancestor State as Client*	ASC jaettuna ASC:n ja komponentin sisäisen perinnän yläluokkien muuttujien käyttömäärällä
ASP	*Number of Ancestor State as Provider*	Komponentin ulkopuolelle perittyjen muuttujien käyttömäärä
RASP	*Relative Number of Ancestor State as Provider*	ASP jaettuna ASP:n ja komponentin sisäisen perinnän yläluokkien muuttujien käyttömäärällä
CC	*Number of Class Clients*	Komponentista riippuvien (perintä, luokkaviittaus tai yläluokan muuttujan käyttö) ulkopuolisten luokkien lukumäärä
NCP	*Number of Classes in a Package*	Luokkien lukumäärä

Ducasse, Lanza ja Ponisio [57]



Kuva 4.2: Vasemmalla absoluuttinen ja oikealla suhteellinen perhoskuvaaja. Oikean kuvaajan päälle on merkitty metriikoiden edustamat sektorit. [57]

Samoin luokkien ja komponenttien lukumäärät toistuvat myös aiemmassa metriikassa. Paketin erikoisuus on esivanhempien instanssimuuttujien käyttö.

Ducasse ym. [56, 57] eivät perustelleet jakoa perimysrelaatioon ja yliluokan instanssimuuttujien käyttösuhteeseen. Jälkimmäinen relaatio ei ole mahdollista ilman perintäsuhdetta, eivätkä määrittelijät erotelleet instanssimuuttujien suoraa käyttöä epäsuorasta. Onko aksessorin käyttäminen eri asemassa kuin suora viittaus instanssimuuttujaan? Ongelmana on, ettei mitan mallintamaa ominaisuutta ole selitetty, eikä näin mitan tarkoituskaan ole selvillä. Kuitenkin käytettäessä tai laajennettaessa perittävän luokan palveluita, eivät aksessorien tai instanssimuuttujien käyttötiheys vaikuta mielenkiintoiselta — etenkään komponenttitasolla, jolla luokkia on kymmeniä.

Ponisio [144] tunnisti perhoskaavioiden avulla useita kategorioita, joihin komponentit voidaan luokitella niiden tarkoituksen tai rakenteen perusteella. Hän myös oletti visualisoinnin auttavan löytämään komponentit, jotka rikkovat arkkitehtuurin vaatimuksia. Nämä erottuisivat kuvaajaltaan muista alijärjestelmän komponenteista. Ponisio ei kuitenkaan esitellyt tarkemmin kaavioiden hyödyntämistä ongelmakomponenttien etsimiseen. Ducassen ym. [57] mitat ovatkin erikoisia siinä, etteivät ne pyri havainnollistamaan

tai ohjeistamaan sisäisen laadun parantamisessa. Niiden tarkoitus on ainoastaan auttaa ymmärtämään miten komponentti on vuorovaikutuksessa järjestelmän kanssa.

4.2.3 Komponenttien kontekstiriippuvainen koheesio

Ponision ja Nierstraszin [143, 144] esittelemä komponenttien kontekstiriippuvan koheesio (engl. *Contextual Package Cohesion*, CPC) on ajatukseltaan hyvin lähellä Mäkelän ja Leppäsen [136] LCIC:tä ja luokan asiakasnäkymää. CPC:ssä komponentti oletetaan kiinnevoimaiseksi, jos sen asiakaskomponentit käyttävät kokonaisuudessaan tarjottua rajapintaa. [143] Ponisio ja Nierstrasz käyttivät CPC:tä myös visualisoidessaan komponenttien rakennetta. [142] Mitan arvo yhdistettynä luokan *tyytyväisyyteen*, paketin sisäiseen käyttöasteeseen, auttoi tutkijoita löytämään empiirisessä kokeessaan muutaman huonosti sijoitetun luokan.

CPC käsittelee komponenttia joukkona luokkien määrittelyitä. [144] Tutkijat eivät tarjonneet konkreettisia kielisidoksia, mutta esimerkiksi Javan rajapinta soveltuu luokan määrittelyn käsitteeseen. Koheesiomitta komponentille P on muodollisesti ilmaistuna [144]:

$$
\begin{aligned}
\text{CPC}(P) &= \sum_{a,b \in I(P)} \frac{f(a,b)}{\#Pairs} \qquad (4.10)\\
&\text{Missä:}\\
I(P) &= interface(P)\\
&= \{d \in P \mid clients(d) - P \neq \emptyset\}\\
\#Pairs &= \frac{|I(P)| \times |I(P) - 1|}{2}\\
C(a,b) &= clients(a) \cap clients(b)\\
f(a,b) &= \begin{cases} 1 & \text{jos} \quad C(a,b) \neq \emptyset \\ 0 & \text{muulloin} \end{cases}
\end{aligned}
$$

Komponentin P "*rajapinta*" $I(P)$ muodostuu luokista, joilla on komponentin ulkopuolisia asiakkaita. Ponision rajapinnan käsite eroaa totutusta, sillä siihen ei kuulu tarjotut vaan

käytetyt palvelut. Komponentin rajapintaan kuuluvan luokan a asiakkaita — $clients(a)$ — ovat ne luokat, jotka ovat perintä-, viittaus-, viestinvälitys- tai perityn muuttujan käyttö -relaatiossa a:n kanssa. [143] Yliluokkien muuttujien käyttösuhde määriteltiin jo luvussa 4.2.2. CPC siis laskee miten laajasti komponentin asiakkaat käyttävät sen rajapinnasta löytyviä palveluita. CPC:n arvo on normalisoitu välille $[0, 1]$, missä nolla tarkoittaa komponentin yhtenäisyyden täydellistä puuttumista. Arvolla yksi asiakkaat käyttävät kaikkia rajapinnan luokkia.

CPC:n määritelmässä huomioidaan viestinvälitysriippuvuus, jossa lähetetty viesti herättää jonkun toisen luokan metodin. Polymorfismia ja dynaamista sidontaa tukevissa kielissä viestin vastaanottajaa ei voida tietää staattisessa tarkastelussa. Ponision ja Nierstraszin [143] lähestymistavassa riippuvuussuhde lasketaan tarkasteltavan luokan ja jokaisen mahdollisen kandidaatin välille. He erottelivat CPC:stä vielä riippuvuussuhteille omat osamitat, esimerkiksi CPC_{INH} perintä- ja CPC_{REF} viittausrelaatiolle.

CPC:n aikaisemmassa versiossa Ponisio ja Nierstrasz lisäsivät asiakkaille kertoimet, jotka painottivat itsenäisiä komponentteja. Komponentit, jotka käyttivät laajasti järjestelmän palveluita, eivät olleet yhtä arvokkaita asiakkaita kuin ainoastaan tarkasteltavaa komponenttia käyttävät. Painot määriteltiin asiakkaan tulevien ja lähtevien riippuvuuksien summan käänteisarvoksi. [143] Määritelmässä oli tosin ristiriita, sillä myös komponentit, joita käytettiin runsaasti, saivat pienempiä painoja, vaikka ne käyttäisivät vain laskettavan komponentin palveluita. Ponisio [144] ei perustellut väitöskirjassaan painojen poisjättämistä.

Ponisio ja Nierstrasz muokkasivat CPC:lle vertailumitoiksi muutamasta luokkien sisäisen näkymän koheesiomitoista komponenteille sopivat versiot. Henderson-Sellersin [80] LCOM5:n komponenttiversio ILCO (engl. *Internal Lack of Cohesion*) on [143]:

$$\begin{aligned} \text{ILCO} &= \frac{(\frac{1}{m}\sum_{j=1}^{m}\mu(A_j)) - n}{1-n} \\ m &= |internals(P)| \\ &= |P - interface(P)| \end{aligned} \tag{4.11}$$

$$
\begin{aligned}
n &= |interface(P)| \\
\mu(A) &= |\{B \mid B \in interface(P) \wedge B \in clients(A)\}|
\end{aligned}
$$

Missä $internals(P)$ on komponentille P niiden luokkien joukko, joilla ei ole ulkopuolisia asiakkaita. Mitan arvo vaihtelee välillä $[0, \frac{n}{n-1}]$.

Fentonin ja Pfleegerin [63] koheesiokerroin (engl. *Cohesion Ratio*, CR) määritellään komponenteille [143]:

$$\text{CR}(P) = \frac{\text{kiinnevoimaisten luokkien määrä}}{\text{luokkien kokonaismäärä}} \tag{4.12}$$

Missä luokka on kiinnevoimainen, jos sen TCC:n arvo on suurempi kuin 0.7. CR:n arvo komponentille on välillä $[0, 1]$.

Sisäisten riippuvuuksien suhde (engl. *Internal Dependencies Ratio*, IDR) on komponentin luokkien keskinäisten suorien riippuvuuksien määrä jaettuna kaikilla mahdollisilla. Ponisio ja Nierstrasz erottelevat vielä neljä eri kytkeytymisen muotoa riippuvuuksien maksimissaan. IDR määritellään [143]:

$$\text{IDR}(P) = \frac{\text{Suorien riippuvuuksien määrä}}{\frac{|P|\,(\,|P|-1\,)}{2} \times 4} \tag{4.13}$$

Mitan arvo on välillä $[0, 1]$.

Komponentin luokkien koheesioaste (engl. *Degree of Cohesion of Objects*, DCO) määritellään luokista riippuvien komponenttien keskiarvoksi [143]:

$$\text{DCO}(P) = \sum_{P} \frac{\text{Luokkaan tulevat komponenttiriippuvuudet}}{|P|} \tag{4.14}$$

Ponisio ja Nierstrasz [143] arvioivat käsin 41 komponentin kiinnevoimaa ja vertasit komponenttikoheesiomittojen arvoja intuitiiviseen käsitykseen. Yllätyksettömästi CPC pärjäsi huomattavasti paremmin kuin neljä luokkamittojen karkeaa käännöstä. Samoin Ponisio ja Nierstrasz onnistuivat tunnistamaan CPC:llä ja sen variaatioilla kiinnevoimaisia komponentteja, jotka perinteiseen sisäiseen näkymään pohjautuvat mitat ohittivat.

Asiakasnäkymän soveltaminen komponenttikoheesiomittoihin on äärimmäisen tärkeää. Komponentin kiinnevoiman arvioiminen pelkästään sen osasten yhteistoiminnalla ei

anna todellista kuvaa esimerkiksi sovelluskehyksissä. [143] Valitettavasti Ponisio ja Nierstrasz eivät kuitenkaan suorittaneet tilastollista analyysiä tuloksista. Samoin kahden tutkijan binäärinen koheesion arviointi ei anna riittävän laajaa kuvaa edes intuitiivisesta kiinnevoimasta. Perhoskuvaajat tai CPC eivät selitä riippuvuuksien jakamista neljään kategoriaan, ja erityisesti perittyjen muuttujien käytön listaamista omaksi riippuvuusmuodoksi.

Vain kolmannes tarkastelluista komponenteista oli sekä intuitiivisesti että luokkakoheesiomittojen käännöksillä kiinnevoimainen. Osin tulos selittyy mittojen suoraviivaisella tulkitsemisella arkkitehtuuritasolle, sillä esimerkiksi koheesiokertoimen TCC:n rajaarvoa 0.7 ei perusteltu. Samoin DCO:n tai CR:n väittämistä komponentin sisäisen yhtenäisyyden asteeksi ei perusteltu.

DCO voidaan ilmaista asiakaskomponenttien kumulatiivisen summan keskiarvoksi — mikä ei ole riippuvainen paketin sisäisestä rakenteesta tai yhtenäisyydestä. Fentonin ja Pfleegerin sovellettu koheesiokerroin ei huomioi komponentin rakennetta tai sisäisiä suhteita, miksi sen soveltuvuus koheesiomitaksi voidaan kyseenalaistaa. Vastaavasti CR:lle komponentti on yhtenäinen, jos sen luokat ovat kiinnevoimaisia. Komponentti voidaan rakentaa useista toisistaan riippumattomasta, mutta hyvin muodostetusta, luokasta. Mitalle tällainen komponentti olisi tällöin yhtenäinen, mikä on vastoin intuitiivista käsitystä. Selvästi myöskään CR ei mittaa komponentin todellista koheesiota.

4.2.4 Muita komponenttimetriikoita kirjallisuudessa

Edellä kuvattujen mittojen lisäksi kirjallisuudessa esiintyy muutamia komponenttimetriikoita. Useimmat niistä ovat yksinkertaisia, lukumääriä ja suhteita laskevia mittoja. Esimerkiksi Baxter ja kollegat [24] esittelivät mitan komponentin julkisten luokkien määrälle. Samoin Ducassen ym. [1, 55, 58] komponenttien visualisointiin apuvälineet ovat hyvin yksinkertaisia esiintymiskertoihin perustuvia mittoja.

Luokkametriikoita on myös sovellettu komponenteille. Esimerkiksi Lindvall ja kollegat [111] määrittelivät CBO:sta kaksi eri versiota korkeammalle tasolle: komponenttien

välisen (engl. *Coupling Between Modules*, CBM) sekä komponenttien ja luokkien välisen kytkeytymisen (engl. *Coupling Between Module-Classes*, CBMC). Bengtsson [27] määritteli Chidamberin ja Kemererin metriikat, DAC:n sekä MPC:n komponenteille. Samoin Yacoub ym. [174] sovelsivat matalammalle tasolle suunniteltua syklomaattista kompleksisuutta arkkitehtuuritasolla komponenttien monimutkaisuuden ja riskialttiuden arvioimiseen. Suoraviivaiset yleistykset kuitenkin kärsivät samoista, opinnäytteessä aiemmin esitellyistä, puutteista kuin alkuperäiset.

Allen, Khoshgoftaar ja Chen [14] ehdottivat informaatioteoriaan pohjautuvia kiinnevoiman ja kytkeytymisen mittoja moduuleille. Esimerkissään he tulkitsivat ohjelmakooditiedostot moduuleiksi, mutta riippuvuusgraafi voidaan muodostaa myös arkkitehtuurikomponenttien välille. Mišić [131] ei myöskään sitonut koheesiometriikkansa määritelmää yksittäiselle abstraktiotasolle, ja tarjoaa mittaa niin Javan paketeille, luokille kuin metodeillekin. Ymmärrettävästi mitat eivät kuitenkaan huomioi luokkien tai komponenttien erityispiirteitä, miksi spesifisemmät komponenttimetriikat käsittelevät laajemmin erilaisia kytkeytymismuotoja.

Van Belle [26] esitteli ohjelmistojen evoluutioteoriaan perustuvia metriikoita, joilla arvioitiin modulaarisuutta, kapselointia ja muutosherkkyyttä. Mitat käyttävät laskennassaan hyödyksi komponentteihin aiemmin tehtyjen muutosten määriä, eivätkä ne näin sovellu ilman komponentin historiatietoja tehtävään analyysiin. Van Belle ei sitonut evoluutiomittojensa määritelmiä tietylle abstraktiotasolle, mutta käytti esimerkeissään Javan paketteja moduuleina.

Zhou ym. [175] ehdottavat myös ympäristön huomioivaa koheesiomittaa paketeille. Yhteisen kontekstin kiinnevoima (engl. *Similar Context Cohesiveness*, SCC) — toisin kuin CPC — huomio vain viittausrelaatiot. SCC sisällyttää laskentaan myös epäsuorat kytkökset. Erikoisesti Zhoun ym. määrittelemään kontekstiin kuuluu komponenttien yhteisten asiakkaiden lisäksi niiden käyttämät yhteiset palvelut. Mitta yrittää tavoitella äärimmäisen hienovaraista koheesioanalyysia, joka havaitsee pienimmätkin erot. Tällaisten

mittojen mielekkyyttä pohdittiin jo alaluvussa 2.3.5. Esitelty mitta on kuitenkin lupaava ehdotus, mutta selkeästi vielä keskeneräinen.

Sarkar, Kak ja Rama [149] ehdottivat joukkoa moduulin rajapintaan pohjautuvia metriikoita. Ehdotuksien joukossa oli mm. särkyvän yliluokan — esitelty esimerkiksi Koskimiehen ja Mikkosen kirjassa [102] — todennäköisyysindeksi (engl. *Base-Class Fragility Index*, BCFI). Sarkarin ym. mitat ovat määritelty superkomponenteille, jotka koostuvat useista sadoista luokista. Korkeintaan muutaman kymmenen luokan kokoiselle arkkitehtuurin peruskomponentille mitat ovat liian karkeita.

Metriikoita on myös kehitetty paljon *off-the-shelf*-komponenteille, kuten Kharbin ja Singhin [99] viisi komponenttien kompleksisuusmittaa. Heidän tietovirtoihin perustuvat mitat pyrkivät havainnollistamaan komponentin monimutkaisuutta syötteiden ja tulosteiden suhteilla, ilman tietoa komponentin rakenteesta tai toteutuksesta. Samoin Washizakin ja kollegoiden [167] viisi JavaBeans-komponenteilla koeteltua mittaa pohjautuvat komponentin käsittelyyn mustana laatikkona.

Valmiskomponenteille on määritelty myös laatukehyksiä [65], mutta tässä opinnäytteessä on kuitenkin keskitytty sisäisiin, lähdekoodista laskettaviin tuotemetriikoihin. Tilarajoitusten vuoksi *Component-Off-The-Shelf* (COTS) –mitat ohitetaan. Esimerkiksi Goulão ja Abreu [72] sekä Poulin [145] ovat tarkastelleet teoksissaan COTS-metriikoita tarkemmin.

Seuraavassa esitellään vielä lyhyesti Lakoksen [105] ja Hautuksen [79] syklisiä riippuvuuksia havainnollistavat metriikat. Lisäksi kuvataan lyhyesti Meltonin ja Temperonin [124] CRSS, joka arvioi komponenttien luokkajaon hyvyyttä.

Komponentin kumulatiivinen riippuvuus

John Lakos [105] esitteli vuonna 1996 mitan komponenttien kumulatiivisille riippuvuuksille (engl. *Cumulative Component Dependency*, CCD). Hän määritteli mitasta myös normalisoidun (engl. *Normalized Cumulative Component Dependency*, NCCD) ja keskiar-

voisen (engl. *Average Component Dependency*, ACD) version [105]:

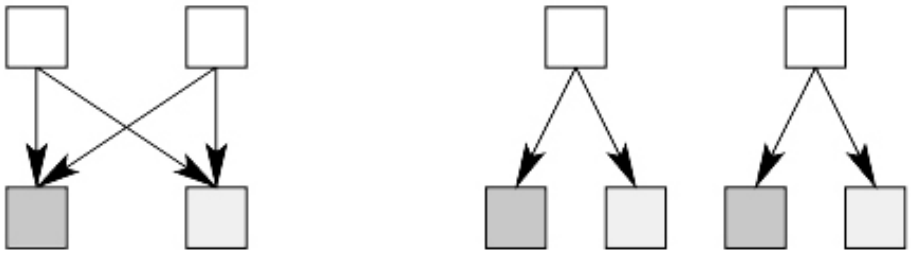
$$
\begin{aligned}
f_{link}(C_i) &= C_i\text{:n kääntämiseen tarvittavien} \\
&\quad \text{komponenttien määrä} \\
\text{CCD}(S) &= \sum_{C_i \in S} f_{link}(C_i) && (4.15) \\
\text{NCCD}(S) &= \frac{\text{CCD}(S)}{\text{CCD}_{BBT}(S)} && (4.16) \\
\text{ACD}(S) &= \frac{\text{CCD(S)}}{|S|} && (4.17)
\end{aligned}
$$

Missä S on alijärjestelmä ja $\text{CCD}_{\text{BBT}}(S)$ on alijärjestelmän CCD:n arvo kun komponenttien riippuvuusgraafi muodostaa tasapainotetun binääripuun. Lakos [105] perustelee binääripuuta joustavana, uudelleenkäytettävänä ja matalan kytkeytymisen suunnitteluna. Hän käyttää sitä mitan yhteydessä ihanteellisena tapauksena, johon mitattavia suunnitelmia verrataan.

Lakos [105] määritteli komponentin arkkitehtuurin pienimmäksi uudelleenkäytettäväksi yksiköksi, mutta rajasi samalla sen koostuvan vain yhdestä rajapinnasta (C++:ssa `.h` -tiedosto) ja sen toteutuksesta (`.c`-tiedosto). Komponenttien riippuvuusgraafiin perustuvan mitan määritelmä on kuitenkin kieliriippumaton ja sovellettavissa myös suuremmille komponenteille.

CCD ei ole komponenttien tai järjestelmän laadun mittari, mutta Lakos väittää sen soveltuvan ylläpidettävyyden ja uudelleenkäytettävyyden mittaamiseen. [105] Esimerkiksi NCCD:n suurilla arvoilla järjestelmästä saattaa löytyä syklisiä riippuvuuksia, joita pidetään yleisesti huonona suunnitteluna. [120] Lakos antaakin yhdeksi suunnitteluperiaatteeksi CCD:n arvon minimoimisen järjestelmässä, mutta painottaa riippuvuuksien hallinnan olevan numeerista arvoa tärkeämpää.

Mitan normalisointi tasapainotettua binääripuuta vasten ei välttämättä sovellu kaikille arkkitehtuurityyleille. Esimerkiksi tietovuo- ja asiakas-palvelin -arkkitehtuuri voidaan kuvata puumaisena kerrosarkkitehtuurina [102], mutta onko yksinkertaisen tasapainote-



Kuva 4.3: Alkuperäinen, vasemmanpuoleinen suunnitelma hajoitettuna pienempiin moduuleihin oikealla. Riippuvuuksien kokonaismäärä säilyy samana, mutta alkuperäisen suunnitelman CCD on 8 ja NCCD on 1.05 kun uuden riippuvuusgraafin CCD on 10 ja NCCD on 0.73. [105]

tun binääripuun riippuvuuksien määrä aina soveltuva mittari kaikille järjestelmille?

Mitta ei myöskään huomio komponentin sisäisen koon [125] tai komponenttien määrän vaikutusta. Esimerkiksi kuvassa 4.3 NCCD:n ja ACD:n arvo pienenee kun alimman tason komponentit jaettiin kahteen osaan, mutta järjestelmän riippuvuuksien kokonaismäärä säilyi. Alkuperäinen suunnittelu saattaa silti olla uudelleenkäytettävämpi ja ylläpidettävämpi ratkaisu kuin pienempiin osiin hajotettu vaihtoehto.

PASTA

Edwin Hautus [79] esitteli PASTA-ohjelman (engl. *Package Structure Analysis Tool*), jonka tarkoitus on auttaa komponenttien järjestämisessä kerrosarkkitehtuurin mukaisiin tasoihin. Hautus esitteli ohjelmassa myös PASTA-metriikan komponenttien rakenteen arvioimiseksi. PASTA mittaa sellaisten komponenttien suhteellista osuutta järjestelmästä, joita muuttamalla komponenttien riippuvuusgraafista saadaan syklitön. Mitta määritellään komponentille p ja järjestelmälle[1] S [79]:

$$\text{PASTA}(p) = \frac{\text{Epätoivottujen riippuvuuksien paino alikomponenteissa}}{\text{Alikomponenttien riippuvuuksien kokonaispaino}} \quad (4.18)$$

[1]Hautus [79] määritteli mitan järjestelmälle toivottujen (engl. *desirable*) riippuvuuksien suhteeksi kaikkiin. *Toivottavan riippuvuuden* määritelmä on kuitenkin epäselvä, eikä Hautus tarkentanut väitettään. Mittojen yhteneväisyyden vuoksi oletetaan hänen tarkoittaneen epätoivottuja (engl. *undesirable*) riippuvuuksia.

$$\text{PASTA}(S) = \frac{\text{Epätoivottujen riippuvuuksien paino komponenteissa}}{\text{Komponenttien riippuvuuksien kokonaispaino}} \quad (4.19)$$

Missä epätoivotut riippuvuudet ovat sellaisia relaatioita, joiden poistamisella saadaan syklitön riippuvuusgraafi. Graafissa kaaren *paino* on komponentin viittausten määrä toiseen komponenttiin. Paino kuvaa riippuvuuden poistamisen työmäärää. [79]

Laskettaessa PASTA:n arvoa järjestelmälle samat riippuvuussuhteet saattavat tulla lisättyä mukaan monta kertaa, sillä suurten ohjelmakomponenttien riippuvuudet muodostuvat alikomponenttien relaatioista. [124] Hautus [79] puolustaa järjestelmämitan määritelmää väittämällä arkkitehtuurin korkeampien tasojen olevan tärkeämpiä kuin matalampien.

PASTA:n arvo kuvaa kuinka suuri osa ohjelmasta on muutettava, jotta riippuvuussyklit saadaan poistettua. [79] Hautus ei kuitenkaan esittänyt metriikkaa formaalisti, minkä vuoksi kahden paketin välisen riippuvuuden määritelmä on epäselvä. Riippuvuus voidaan laskea esimerkiksi kahden luokan välille binäärisesti tai käytettyjen tyyppien määrinä. [124] Samoin, onko perintärelaatio samanarvoinen kuin tyyppiviittaus parametrissa? Perintäriippuvuuden poistaminen on kuitenkin huomattavasti työläämpää kuin yksinkertaisten viittausten.

Hautus [79] suosittelee Martinin [119] esittelemien refaktorointimenetelmien käyttämistä riippuvuuksien suuntien kääntämiseksi. Hän muistutti kuitenkin yksinkertaisista komponenttirefaktoroinneista, kuten esimerkiksi luokan siirtämisestä komponentista toiseen. Nämä saattavat soveltua Martinin listaamia raskaita refaktorointeja huomattavasti paremmin syklisten riippuvuuksien poistamiseen.

CRSS

Hayden Melton ja Ewan Tempero [124, 125] perustavat komponentin uudelleenkäytettävyyden ja testattavuuden komponenttisuunnitteluperiaatteiden noudattamiseen. Kokorajojen lisäksi he painottivat erityisesti komponentin itsenäisyyttä. Määritelläkseen komponenttimitan Melton ja Tempero [124] esittelivät ensin luokkien riippuvuusgraafiin (engl.

Class Dependency Graph, CDG), jossa luokat ovat solmuja ja käännösriippuvuudet suunnattuja kaaria. Luokan c saavutettavuus (engl. *Class Reachability Set Size*, CRSS) on niiden luokkien lukumäärä, joihin päästään solmusta c. [124] Toisin ilmaistuna CRSS(c) on transitiivinen sulkeuma niistä luokista, joita luokan c kääntäminen vaatii. [125]

Mitan tarkoituksena on parantaa komponentin rakennetta, erityisesti pitämällä ne hallittavan kokoisina ja itsenäisinä. CRSS laajennetaan yksittäisestä luokasta koko järjestelmälle tarkastelemalla luokkien arvojakaumaa. Jos mitan histogrammi kallistuu oikean laidan korkeisiin arvoihin, komponentit muodostavat riippuvuussyklejä. Tällöin yksittäisen luokan kääntäminen vaatii koko järjestelmän kääntämistä. Melton ja Tempero uudistivat onnistuneesti ongelmallisten luokkien rajapintoja *extract interface* -refaktoroinnilla. [124]

Melton ja Tempero [124] kutsuvat CRSS:ää luottavaisesti komponenttisuunnittelun laatumetriikaksi. Samoin he pitävät Hautuksen lähestymistapaa liian komponenttikeskeisenä. CRSS on kuitenkin oudosti määritelty, sillä se ei tarkastele riippuvuuksia erikseen komponentin sisälle tai ulkopuolelle. Melton ja Tempero [124] painottavat CRSS-histogrammin paljastavan huonon komponenttisuunnittelun, mutta mitta ei yhdessäkään vaiheessa huomioi luokkien jakoa osajoukkoihin. Järjestelmän histogrammi ei muuttuisi vaikka luokkajako uudistettaisiin tai jätettäisiin tekemättä. Silti Melton ja Tempero [124] ehdottivat CRSS:n käyttämistä luokkajaon lähtökohtana. Mitta toimii kuitenkin paremmin luokkametriikkana kuin komponenttimittana, sillä jopa suositeltu refaktorointi uudistaa vain luokan rakennetta eikä ohjelmakomponenttia.

4.3 Mitattavat ominaisuudet

Ohjelmistomittojen tarkoitus on luokitella ohjelmamoduulit paremmuusjärjestykseen yhdistämällä ominaisuudet numeroarvoihin tai symboleihin yksinkertaisten sääntöjen avulla. [63] Ominaisuudet voivat olla konkreettisia, suoraan mitattavia suureita kuten ohjel-

man koko. Ne voivat olla myös abstrakteja, joiden mittaamisesta ei välttämättä ole yhtenäistä käsitystä. Esimerkiksi alaluvussa 2.3 esitetyt koheesiomitat perustuvat hyvinkin erilaisiin tapoihin hahmottaa abstraktia käsitettä. Ohjelmakomponentille voidaan määritellä lukuisia ominaisuuksia, mutta kuten luokkatasolla on osoitettu, vain harvat niistä vaikuttavat kiistattomasti ohjelman ulkoiseen laatuun.

Edellä esitetyissä komponenttimetriikoista on havaittavissa luokkien ominaisuuksien tulkitsemisesta korkeammalle abstraktiotasolle. Ohjelmakomponentti ei kuitenkaan ole luokan yleistys [57], jolle kaikki alemman tason ominaisuudet soveltuisivat. Seuraavaksi pyritään tunnistamaan arkkitehtuuritason ominaisuuksia, joilla arkkitehtuurin ja komponenttien laatua pystyisi mittaamaan. Samalla arvioidaan miten nykyiset komponenttimetriikat ja erityisesti Martinin esittelemät mitat soveltuvat ominaisuuksille.

4.3.1 Komponentti

Tässä luvussa listataan komponenttien ominaisuuksia, joita tasolle määritellyt metriikat pyrkivät mittaamaan tai mitä komponenteista voisi havaita. Lähestymistapa keskittyy luettelemaan komponentin hyviä ohjelmointitapoja ja ominaisuuksia, sekä esittelemään miten mittojen arvoja voitaisiin hyödyntää. Toisenlainen tarkastelutapa on esimerkiksi suunnittelun ongelmien [120] tai Meyerin [128] viiden modulaarisuuden kriteerin mittaaminen. Huonoja suunnitteluvaihtoehtoja on kuitenkin huomattavasti enemmän kuin hyviä, ja esimerkiksi jäykkyyden (engl. *rigidity*) eli muutosten tekemisen vaikeuden [120], arvioiminen yksittäisellä mittarilla on hyvin hankalaa.

Toivottavien ominaisuuksien mittojen, kuten koheesion, huonon arvon ei ole osoitettu riippumattomasti vaikuttavan ulkoisten laatukriteereiden heikentymiseen. Ongelmaksi on arvioitu niin riippuvuutta kokoon [60], kuin käsitteiden huonoa ymmärrystä [36]. Suunnittelun ongelmien mittaaminen olisi kuitenkin yksi keino kehittää ohjelmistometriikoita, sillä huonon ohjelmakoodin refaktoroiminen yrittää parantaa ainakin järjestelmän ylläpidettävyyttä ja ymmärrettävyyttä. [64] Käytettävästä lähestymistavasta huolimatta

esiteltyjen mittojen ja ominaisuuksien riippuvuudet virhe- tai muutosmääriin tulisi aina tarkastaa empiirisillä tutkimuksilla.

Tässä luvussa listataan useita ominaisuuksia, mutta komponenttien mittaaminen voidaan karkeasti tiivistää vain muutaman ominaisuuden ympärille. Esimerkiksi Melton ja Tempero [125] väittävät komponenttien määrän, koon, kapseloinnin, kytkeytymisen ja koheesion ohjaavan ohjelmakomponenttien suunnittelua. Näistä erityisesti kaksi viimeistä ovat määrääviä tekijöitä.

Van Belle [26] piti modularisointia kahden evoluutiomitan, laajuuden (engl. *breadth*) ja taakan (engl. *weight*), optimoinnin ongelmana. Hänen ajatuksiaan mukaillen järjestelmän osittamisen komponenteiksi voidaan määritellä tehtäväksi, missä koheesio pyritään maksimoimaan ja kytkeytyminen minimoimaan. Järjestelmän koheesioarvojen maksimissa jokainen luokka muodostaisi yhden komponentin. Kytkeytymisen minimissä järjestelmä olisi vain yksi suuri komponentti. Koheesion ja kytkeytymisen optimoinnin avulla pitäisi järjestelmän jakautua itsenäisiin ja yhtenäisiin komponentteihin.

Kiinnevoima ja kytkeytyminen ovat komponenteille tärkeitä ominaisuuksia, mutta keskittyminen ainoastaan näihin jättää huomioimatta olioparadigman muut piirteet. Matalammilla tasoilla koheesio- ja kytkeytymismetriikat ovat suoriutuneet kohtuullisesti virheiden osoittajina [37], mutta toiset ominaisuudet saattavat toimia niitä paremmin komponenttitasolla. Erilaisten ominaisuuksien mitoilla voidaan tunnistaa erilaisia ongelmia. Esimerkiksi riippuvuussyklien havaitseminen koheesio- tai kytkeytymismitoilla on hankalaa ellei mahdotonta.

Seuraavassa on käsiteltyä joukkoa komponenttien ominaisuuksia, joita voitaisiin mitata. Listauksen ohessa on arvioitu ominaisuuksien soveltuvuutta arkkitehtuurin ja komponenttien laadun tunnistajina. Samassa on myös käsitelty ominaisuudelle ehdotettuja mittoja ja mahdollisia kehitysehdotuksia. Listalla esiintyy samoja ominaisuuksia, joita Berard [28] ehdotti olioparadigman mitattaviksi piirteiksi.

Abstraktius

Abstraktius 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 konkreetteja luokkia olevan enemmän kuin abstrakteja. Tosin tulokseen tulkintaan vaikuttavat myös kokonaismäärän laskennassa käytetyt luokkaroolit. Esimerkiksi sisäluokkien huomioiminen pienentäisi joidenkin komponenttien abstraktiutta.

Abstraktiuden käyttämistä suorana mittana on vaikea perustella, sillä todennäköisesti ominaisuudella ei ole vaikutusta komponentin ulkoiseen laatuun. Martin [116] esitteli ominaisuudelle mitan, jota tarkasteltiin sivulla 86. Hän hyödynsi abstraktiutta osana johdettua mittaa, eikä käsitellyt sen käyttämistä yksinään komponenttien laadun arvioimisessa. Ominaisuudella saattaa olla käyttöä johdetuissa mitoissa, mutta ei puhtaana laadun arviojana.

Kapselointi ja informaation kätkeminen

Informaation piilottaminen ja kapselointi ovat modularisoinnin perusperiaatteita. Tarkoituksena on paljastaa ainoastaan palveluiden vaatima tieto, mutta salata toteutuksen yksityiskohdat. [128] 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 särkyvän yliluokan ongelmassa. Tämän vuoksi huonosti kapseloidut komponentit olisi hyvä pystyä tunnistamaan ajoissa.

Van Belle [26] mittasi kapseloinnin onnistumista komponentin muutoshistoriatietojen

avulla, mutta lähestymistapa ei sovellu puhtaasti staattiseen analyysiin. Abreu [6] yritti mitata informaation piilottamista yksityisten osien suhteena julkisiin. Koska sama toiminnallisuus voidaan jakaa monelle eri tekijälle tai toteuttaa vain yhdessä komponentissa, ei suhteisiin perustava laskenta sovellu suoraviivaisesti yksityiskohtien kätkemisen arvioimiseen.

Martin ei määritellyt kapselointia tai informaation piilottamista arvioivia mittoja. Hänen määrittelemistään mitoista voidaan kuitenkin johtaa metriikka kapseloinnin onnistumisen mittaamiseksi. Käytettävässä näkökulmassa lasketaan abstrakteille rajapinnoille saapuvien riippuvuuksien suhde kaikkiin. Abstraktien rajapintojen käyttöaste (engl. *Interface Utilization Rate*, IUR) määritellään:

$$\text{IUR} = \frac{C_a^{abs}}{C_a} \tag{4.20}$$

Missä C_a^{abs} on komponentin abstrakteista rajapinnoista riippuvien ulkopuolisten luokkien lukumäärä. Mitan arvo vaihtelee välillä $[0, 1]$, missä nolla kuvaa abstraktien rajapintojen täydellistä käyttämättömyyttä. C_a^{abs} jättää avoimeksi riippuvuuden määritelmän, kuten myös Martinin alun perin ehdottama C_a, joka esiteltiin sivulla 85. Määritelmä on kuitenkin konkreettisesti sidottava ennen mittauksia. Eräs tulkinta olisi käyttää käännösaikaisia luokkariippuvuuksia. Tämä tarkoittaisi toisaalta myös perinnän huomioimista, mikä ei välttämättä ole kaikissa tapauksissa toivottavaa.

Rajapintojen käyttöastetta laskettaessa pitää myös huomioida asiakasluokkien roolit. Esimerkiksi *Abstraktin tehtaan* luokat ovat riippuvaisia komponentin konkreettisista osista, mutta suunnittelumalli auttaa vähentämään riippuvuuksien kokonaismäärää. Roolijaon kehittämisellä ja hienovaraistamisella mitan arvoja keinotekoisesti huonontavia tekijöitä voidaan vähentää. On huomattava myös, ettei mitta sovellu kaikille komponenttirooleille. Esimerkiksi *plug-in* -arkkitehtuurissa saattaa esiintyä ulkoisia rajapintoja toteuttavia komponentteja, joiden ainoat konkreettiset asiakkaat ovat suunnittelumallien mukaisia tehtaita.

Mitan arvon ollessa lähellä nollaa, komponentin ulkopuoliset luokat ovat riippuvai-

sia suoraan palvelujen konkreettisesta toteutuksesta. Rajapintojen käyttöasteen huono arvo ei kuitenkaan sovellu suoraan ulkoisten laatutekijöiden mittariksi, sillä kyseessä on enemmänkin hyvä ohjelmointitapa kuin selkeästi laatua heikentävä virhe. Sisäisiin laatutekijöihin, kuten ymmärrettävyyteen ja ylläpidettävyyteen, ominaisuus kuitenkin vaikuttaa. Näiden perusteella voidaan olettaa huonon arvon saaneiden komponenttien olevan virhealttiimpia kuin muut, sillä vaikeasti ylläpidettäviin ja ymmärrettäviin komponentteihin oletettavasti kertyy enemmän virheitä kuin muihin. Oletus pitäisi kuitenkin varmistaa empiirisillä mittauksilla ennen mitan hyväksymistä.

Kapselointi ja informaation piilottaminen ovat laajoja käsiteitä, joista abstraktien 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. Tässä opinnäytteessä ei kuitenkaan käsitellä näitä näkökulmia.

Koheesio

Koheesio kuvaa komponentin yhtenäisyyttä [63] tai tarkoitusta [131], ja kytkeytymisen ohessa se on toinen tärkeimmistä ominaisuuksista arkkitehtuuritasolla. Koheesio vaikuttaa niin komponentin uudelleenkäytettävyyteen, ymmärrettävyyteen kuin ylläpidettävyyteen. [32, 144, 172] Komponenteille on esitelty Ponision ja Nierstraszin [143, 144] CPC, Zhoun ym. [175] SCC sekä Martinin [120] H koheesiomitoiksi. Näiden lisäksi Ponisio ja Nierstraszin [142] määrittelivät uusiksi muutaman luokkatason koheesiomitan komponenttitasolle. Myös Allenin ym. [14] ja Mišićin [131] määrittelemiä informaatioteoriaan perustuvia kiinnevoiman mittoja voidaan soveltaa komponenttitasolla.

CPC ja SCC perustuvat asiakasnäkökulmaan. Näistä tosin kumpaakaan ei ohjeistettu käyttämään yhdessä muiden koheesiomittojen kanssa. Asiakasnäkökulmaan keskittyvien mittojen ongelma on informaation puute komponentin sisäisestä rakenteesta. Esimerkiksi CPC:llä ei pystytä havaitsemaan komponenttia, jolla on käyttämättömiä luokkia. Selvästi

tällaiset komponentit eivät ole kiinnevoimaisia. Mäkelä ja Leppänen [136] ehdottivat asiakasnäkymiin pohjautuvien koheesiomittojen käyttämistä perinteisen sisäistä eheyttä tarkastelevien mittojen tukena, sillä yhteen näkökulmaan perustuva mitta ei pysty antamaan täyttä kuvaa komponentista.

Ponision ja Nierstraszin [142] uudelleenmäärittelevät muutamia luokkien sisäisen näkymän koheesiometriikoita komponenttitasolle. Valitettavasti ne kärsivät samoista ongelmista kuin alkuperäiset mitat. Esimerkiksi ILCO on Henderson-Sellersin [80] LCOM5:n versio, jossa metodit on korvattu komponentin asiakkaiden käyttämillä luokilla. Luokkatason mitan käyttämät attribuutit on korvattu luokilla, joilla ei ole ulkoisia asiakkaita. Mitta on yhdistelmä sekä sisäistä että ulkoista näkymää.

Briand ym. [32] osoittamat ongelmat LCOM5:ssä koskevat myös ILCO:a. Ponisio ja Nierstrasz eivät myöskään perustelleet komponenttien luokkien jakamista kahteen eriarvoiseen ryhmään. Tästä seuraa ongelmia esimerkiksi komponenteille, joiden kaikkia luokkia käytetään ulkoa käsin. Tällaisille komponenteille ILCO:n arvoa ei ole määritelty, sillä joukko $internals(P)$ on tyhjä ja kertoimen nimittäjäksi tulisi nolla.

Ponision ja Nierstraszin [142] ehdottivat myös kolmea sisäistä koheesiomittaa komponentilla. CR:llä ja DCO:lla oli hyvin vähän tekemistä komponentin yhtenäisyyden mittaamisessa, ja näiden kahden ongelmia tarkasteltiin jo aiemmin. Sivulla 94 esitelty IDR soveltuu paremmin komponenteille, mutta mitta laskee jokaisen riippuvuussuhteen erikseen. Kaksi luokkaa ovat mitalle täysin kytkeytyneitä vain, jos niiden väliltä löytyvät sekä perintä-, viestinvälitys-, käyttö- että perityn muuttujan käyttö -relaatiot. Tilanteet, joissa luokat ovat kytkeytyneet edes kolmella mainitulla tavalla, ovat harvinaisia. Tämän vuoksi IDR tulee antamaan matalia koheesioarvoja suurimmalle osalle arvioitavista komponenteista. Ei myöskään ole selvää, onko mitan ihannekoheesion tilanne toivottava, vai pitäisikö näin tiukasti kytkeytyneet osat yhdistää yhdeksi suureksi luokaksi.

Martinin [120] määritteli alkuperäisen metriikkapakettinsa ulkopuolella H:n, joka on luokan sisäiseen näkymään perustuva koheesiomitta. H tosin kärsii useista ongelmista,

sillä sen määritelmä on jätetty avonaiseksi eikä mitalla ole kiinteitä ylä- tai alarajoja. Mitan ongelmia on tarkasteltu tarkemmin alaluvussa 5.1.1. Selvästi H ei ole riittävä komponentin koheesiomitaksi, eikä Martinin paketista löydy muita soveltuvia mittoja komponenttien koheesiolle. Seuraavassa tarkastellaan millaiset koheesiomitat paketeille sopisivat. Tarkastelua on jaettu kahteen osaan: sisäiseen ja asiakasnäkymään.

Sisäinen näkymä Yksinkertaisempi keino tarkastella komponentin koheesion laskemista on hyödyntää alaluvussa 4.1.2 esiteltyä graafiteoreettista mallia. Komponentin riippuvuusgraafille voidaan soveltaa alaluvussa 2.3.1 esiteltyä LCOM4:ää. Määritellään LCOP(P) (engl. *Lack of Cohesion in Package*) komponentille P sen riippuvuusgraafin G_P:n yhtenäisten osien määräksi. Graafissa voidaan myös käyttää komponenttien epäsuoria riippuvuuksia, jolloin tarkastelusta nousee esille mahdollisesti käyttämättömät luokat. Hitzin ja Montazerin [81] sivulla 32 esiteltyä Co:ta voidaan soveltaa komponenttien riippuvuusgraafien tiheyden arvioimiseksi.

Kuten kapseloinnin yhteydessä, myös LCOP vaatii riippuvuuden määritelmän kiinnittämistä. Graafissa on kyse toisiaan käyttävistä luokasta, eikä yksittäisten kytkösten laadusta tai vahvuuksista. Näin käännösaikainen riippuvuus soveltuisi myös koheesion laskemiseen, sillä siinäkin tarkastellaan vain löytyykö kahden luokan väliltä kytkös.

Asiakasnäkymä Ponision ja Nierstraszin [143, 144] CPC tai Zhoun ym. [175] SCC soveltuvat hyvin kontekstipohjaiseksi koheesiomitaksi. SCC käyttää niin yhteisiä palveluita ja asiakkaita kuin epäsuoria riippuvuuksia määritellessään komponentin kiinnevoimaisuutta. Zhou ym. sisällyttivät tarkasteluunsa luokkien käyttämät yhteiset palvelut. Lähestymistapa on erikoinen, sillä kahden luokan käyttämät yhteiset palvelut eivät luo kytköstä niiden välille. Luokkien käyttämät yhteiset palvelut eivät myöskään tuota yhtenäistä komponenttia. Esimerkiksi CPC tarkastelee ainoastaan komponentin julkisen rajapinnan käytön yhteneväisyyttä, ja noudattaa

näin enemmän puhdasta asiakasnäkymää. Näin CPC saattaisi olla SCC:tä soveltuvampi komponentin asiakaspohjaisen koheesion arvioimiseen.

Koko

Melton ja Tempero [125] listasivat koon hallitsemisen yhdeksi modularisoinnin rajoitteista, sillä komponentin pitäisi olla sitä rakentavan työryhmän hallittavissa. Samalla toinen periaate, kytkeytymisen minimoiminen, ajaa luokkia yhdeksi suureksi komponentiksi. [26] Koko on selkeästi yksi keskeisistä mitattavista ominaisuuksista ja sitä voidaan havainnollistaa esimerkiksi lähdekoodirivien, metodien tai luokkien lukumäärällä. Kuitenkin luokan piirteisiin perustuvat mitat kertovat enemmän luokista kuin komponentista, eivätkä esitetyn informaatiorajauksen perusteella metodit kuuluneet komponenttitasolle.

Luokkien määrän mittauksessa voidaan ottaa huomioon kaikki [57, 120] tai vain julkiset luokat [24]. Luokkaroolien huomioiminen on koon laskennassa keskeistä, sillä esimerkiksi Javan käyttöliittymäkomponenteissa käytetään paljon nimettömiä luokkia. Sisäluokkien rajaaminen pois antaa paremman yleiskuvan komponentista, mutta voi vääristää komponentin rakentamiseen tarvittavan työmäärän arvioimista. Komponentin koon laskennassa roolijako voidaan tehdä hyvinkin hienovaraiseksi, ja valita soveltuvat roolit käytettäväksi tarpeen tai tilanteen mukaan.

Martin [120] antoi kaksi komponenttien kokomittaa paketissaan. N_c laskee komponentin kaikkien luokkien ja N_a abstraktien luokkien määrää. Mitoille ei ole määritelty huomioitavia rooleja, joita erityisesti N_c:n laskemisen yhdenmukaistamisessa tarvittaisiin. Näistä mitoista N_c on soveltuva yleiseksi kokomitaksi.

N_c:n ohessa komponenttien kokomittana kannattaa käyttää myös LOC:tä, sillä piirteisiin perustuvat kokomitat toimivat lähdekoodirivien lukumäärää huonompana ennustajana luokkatasolla. Näin LOC:llä on ollut osansa niin työmäärän kuin virhetiheyden arvioinneissa jokaisella abstraktiotasolla. Koska sekä fyysisen että loogisen lähdekoodirivin laskeminen on helposti automatisoitavissa ja useimpien kehitystyökalujen tukema toiminto,

LOC:n laskeminen myös komponenteille on perusteltua.

Kompleksisuus

Kompleksisuus tarkoittaa moduulin, tässä tapauksessa ohjelmakomponentin, ymmärtämisen vaikeutta. [80] Kompleksisuus on abstrakti ominaisuus, jossa osatekijöinä on useita muita ominaisuuksia. Esimerkiksi Ponisio [144] ja Zhou ym. [175] väittävät koheesion vaikuttavan suoraan komponentin monimutkaisuuteen. Komponenttitasolla kuitenkin tarkastellaan luokkia ja niiden välisiä suhteita, jolloin komponentin monimutkaisuus muodostuu luokkien keskinäisistä suhteista. Metriikoiden tulisi mitata alkuperäistä ominaisuutta, ei sen heijastetta [63], miksi Ponision väitteen perusteella on parempi mitata komponentin koheesiota kuin sen kompleksisuutta.

Perinteiset kompleksisuusmitat ovat myös usein epäonnistuneet virheiden tai muutosten ennustajina [150], 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 suunnitteluongelmat ja kokomitoilla hallitsemattoman suureksi paisuvat komponentit.

Martin ei esitellyt kompleksisuusmittaa, mutta komponenttien monimutkaisuutta voidaan arvioida hyödyntämällä esimerkiksi hänen kytkeytymis- ja koheesiomittojaan. Komponenttitasolle on ehdotettu muutamia kompleksisuusmittoja, mutta nämä ovat olleet vain yleistyksiä alemmilta tasoilta. Esimerkiksi alaluvussa 2.1.2 käsitellyn McCaben syklomaattisen kompleksisuuden määritelmää on kritisoitu jo metoditasolla, ja mitan suoraviivainen tulkitseminen ylemmälle tasolle ei poista taustalla olevia ongelmia.

Kytkeytyminen

Koheesion ohessa kytkeytyminen on tärkeimpiä mitattavista komponentin ominaisuuksista, sillä kytkeytyminen vaikuttaa osaltaan niin uudelleenkäyttävyyteen, ymmärrettävyyteen kuin ylläpidettävyyteen. [33] Kytkeytyminen tarkoittaa yksittäisen komponentin kytkösten määrää järjestelmään, mikä vaikuttaa esimerkiksi komponentin uudelleenkäytettävyyteen ja testattavuuteen.

Meltonin ja Temperon [124] CRSS yrittää havainnollistaa komponentin itsenäisyyttä, mutta mitta keskittyy enemmän luokkien ominaisuuksiin. Lindvall ja kollegat [111] määrittelivät CBM:n komponenttien väliselle ja CBMC:n komponentin ja luokkien väliselle kytkeytymiselle. Näiden lisäksi Allenin ym. [14] informaatioteoriaan pohjautuvaa mittaa voidaan käyttää kytkeytymisen arvioimisessa.

Kytkeytymistä voidaan tarkastella jakamalla lähtevät ja tulevat riippuvuudet erilleen. [33] Esimerkiksi Martinin [116] C_a ja C_e laskevat saapuvia ja lähteviä luokkariippuvuuksia. Perhoskuvaajissaan Ducasse ym. [57] käyttivät vastaavasti CP:tä ja PP:tä komponenttien välisille lähteville ja saapuville riippuvuuksille. Ducassen ym. mitoissa kaksi pääkategoriaa jakautuu vielä erikseen käyttö- ja perintäriippuvuuksiin.

Ducasse ja kollegat [57] määrittelivät komponenttien riippuvuuden tarkasti käyttö- ja perintärelaatioilla. Martin puolestaan ei ottanut kantaa siihen, mikä kytkee komponentin ja luokan toisiinsa. Määritelmäksi C_a:lle ja C_e:lle voisi soveltaa esimerkiksi perhoskuvaajien käyttämää mallia. Toisena vaihtoehtona on käyttää käännösriippuvuutta, jossa mikä tahansa käännösyksikön toiseen yhdistävä riippuvuus lasketaan kytkökseksi. Kytkeytymistyyppien samaistamista yhden mitan alle arvosteltiin jo CBO:n yhteydessä [109], mutta tarkasteltaessa komponentteja uudelleenkäytettävyyden yksiköinä ei kytkösten tyypillä ole väliä. Riippumatta käyttötiheydestä tai tavasta, kytkeytyneet komponentin on toimitettava aina uudelleenkäytettäviksi yhdessä.

Kytkeytymisen tarkastelussa pitäisi myös huomioida relaatioiden vahvuus. [33] Esimerkiksi Martin [116] laski kytköksiä luokkien tarkkuudella. Ducasse ym. [57] taas käyt-

tivät binääristä tulkintaa komponenttien välillä. Kummallakin laskentatavalla on etunsa. Kahden komponentin välisen kytkeytymisen lujuutta voidaan arvioida luokkariippuvuuksien määrällä, mutta komponentin kytkeytymistä koko järjestelmään voidaan tarkastella komponenttiriippuvuuksilla. Suuressa järjestelmässä luokkamääräinen tarkastelu sisältää huomattavan määrän informaatiota, mikä saattaa olla ongelmallista joissain tapauksissa.

Esimerkiksi komponentin P_1 kymmenen luokkaa ovat riippuvaisia P_2:n palveluista. Komponentin P_1 lähtevien komponentti-komponentti -riippuvuuksien määrä on yksi, mitä voidaan pitää matalana. Toisaalta P_1:n luokka-komponentti -riippuvuuksien määrä on 10, minkä perusteella komponentti vaikuttaa tiiviisti järjestelmään kytkeytyneeltä. Laskentatavan valinta on huomioitava määriteltäessä ja käytettäessä kytkeytymismittoja.

Seuraavassa tarkastellaan tarkemmin komponenttien kytkeytymismittoja. Tarkastelu on jaettu kolmeen osaan: asiakkaiden ja käytettyjen palveluiden määriin sekä kokonaiskytkeytymiseen. Asiakkaiden määrä tarkastelee komponenttiin saapuvia kytköksiä ja vastaavasti käytettyjen palveluiden määrä komponentista lähteviä kytköksiä. Kokonaiskytkeytyminen yhdistää kaksi edellä mainittua näkökulmaa.

Asiakkaiden määrä. Asiakkaiden lukumäärä kertoo järjestelmän riippuvuudesta komponenttiin, mikä Martinin [120] mukaan vaikuttaa sen stabiiliuteen. Hän perusteli jokaisen asiakkaan kasvattavan komponentin vastuuta, eli lisäävän syitä muuttumattomuudelle. Pahimmassa tapauksessa vastuulliseen komponentin muuttamisen työmäärään on lisättävä sen jokaisen asiakkaan refaktorointi. Samoin vastuullinen, suuren asiakasmäärän, komponentti on todennäköisesti keskeisessä asemassa arkkitehtuurissa, miksi sen pitäisi nousta testauksen priorisoinnissa korkealle. Asiakkaiden lukumäärä ei siis suoranaisesti mittaa ohjelman tai komponentin laatua, mutta auttaa keskittämään testausresurssit järjestelmän keskeisimpiin komponentteihin.

Martin [120] määritteli asiakkaiden määrän laskemiseen C_a:n. Ducassen ym. [57] luokka-asiakkaiden määrää laskevaa CC vastaa Martinin määrittelemään mittaa. Tosin Ducasse ym. antoivat myös komponenttiasiakkaiden määrää laskevan CP:n.

Näistä Martinin ja komponentti-komponentti -kytköksiä laskeva CP ovat riittäviä komponenttiin tulevien kytkösten mittareita. Vaihtoehtoisesti ominaisuudelle voitaisiin määritellä mitta myös suhteellisena, sillä molemmat mainitut mitat ovat järjestysasteikolla. Eri järjestelmän komponenttien keskinäinen vertaaminen on tällä asteikolla hankalaa. Suhteellinen mitta voidaan määritellä esimerkiksi solmuun tulevien kaarien suhteeksi graafin kaarien lukumäärään tai maksimiin. Tällaisiin variaatioihin ei kuitenkaan tässä opinnäytteessä syvennytä tarkempaa.

Martinin mitalle ongelmallisia ovat Julkisivun (engl. *Façade*) kaltaiset suunnittelumallit, sillä ne kätkevät komponentin käyttäjät yksittäisen luokan taakse. Ratkaisuna tähän laskennassa voisi huomioida epäsuorat kytkennät, mutta silloin keskeisten komponenttien arvot siirtyisivät niiden hyödyntämille palveluille. Toinen vaihtoehto olisi hyödyntää komponentti- ja luokkarooleja lisäämällä suunnittelumallien välittämät riippuvuudet niiden piilottaminen komponenttien arvoihin.

Käytettyjen palveluiden lukumäärä. Käytettyjen palveluiden määrä kertoo komponentin suhteesta muuhun järjestelmään. Martinin [120] perusteluiden mukaan jokainen palvelu, josta komponentti on riippuvainen, on mahdollinen syy muutokselle. Muiden tuottamista palveluista riippumaton komponentti on itsenäinen ja helposti kierrätettävä. Ominaisuus kertoo osaltaan myös komponentin asemasta arkkitehtuurissa, esimerkiksi kerrosarkkitehtuurissa korkeimman tason komponentin käyttämien palveluiden lukumäärän pitäisi olla suurempi kuin alimman tason.

Ulkoiset riippuvuudet ovat myös tärkein mitta komponentin itsenäisyydestä. Jos komponentti on riippuvainen ainoastaan järjestelmäkirjastoista, se on helposti siirrettävissä toiseen ympäristöön. Toisaalta jos ulkoisten riippuvuuksien määrä on suuri, komponentin uudelleenkäyttäminen saattaa vaatia kokonaisen alijärjestelmän kierrättämistä. Komponenttiroolit ovat tärkeässä osassa riippuvuuksien tunnistamisessa, sillä palveluiden lukumäärässä tulisi huomioida erikseen riippuvuudet ulkoisiin kirjastoihin ja muihin järjestelmiin. Esimerkiksi käytettyjen palveluiden lu-

kumäärässä stabiilit palvelut voitaisiin painottaa eri tavalla kuin epävakaat.

Martin [116] ehdotti lähtevien kytkösten laskemiseksi C_e:tä, joka tarkastelee komponentin riippuvuuksia ulkopuolisiin luokkiin. Ducasse ja kollegat [57] käyttivät ohjelmakomponenttien visualisoinnissa PP:tä, joka laskee käytettyjen komponenttien lukumäärää. Ducasse ym. esittelivät PP:stä vielä erilliset mitat lähteville perintä- ja käyttökytköksille. Komponenttitasolla tosin kytkösten tyyppi ei ole niin mielenkiintoinen kuin niiden määrä. Esimerkiksi käyttösuhteella voidaan toteuttaa osa perinnän toiminnallisuudesta [163], miksi perintä voidaan käsitellä vain yhtenä kytkeytymisen muotona.

Kokonaiskytkeytyminen. Kolmas vaihtoehto on kahden edellä kuvatun ominaisuuden summa. Briand ja kollegat [33] arvostelivat jo CBO:ta kytkeytymissuuntien yhdistämisestä, sillä tulevien ja lähtevien kytkösten määrät kertovat erilaista tietoa luokan asemasta järjestelmässä. Sama ongelma koskee Lindvallin ym. [111] ehdottamaa CBM:ää ja CBMC:tä. Ominaisuuksien raaka yhteen laskeminen kadottaa käyttökelpoista tietoa, ja saatu tulos kertoo vain komponentin kautta kulkevan liikenteen aktiivisuudesta. Tässä opinnäytteessä noudatetaan Briandin ym. suositusta, ja tarkastellaan kytkeytymistä kahtena erillisenä osana.

Modulaarisuus

Modulaarisuus kuvaa kuinka hyvin ohjelma on jaettu itsenäisiin ja yhtenäisiin komponentteihin. [26] Ominaisuus on yksi arkkitehtuurisuunnittelun tärkeimmistä tavoitteista [31], miksi modulaarisuudelle on kehitetty myös omia mittoja. Tosin muodostettujen komponenttien itsenäisyyttä voidaan arvioida kytkeytymisellä, yhtenäisyyttä koheesiolla ja hallita kokoa kokomitoilla. Ominaisuuden arvioiminen erillisellä mitalla ei ole tarpeellista, sillä jokaista sen seurausta pystytään mittaamaan yksittäisillä mitoilla. Esimerkiksi modulaarisuutta voidaan osaltaan arvioida käyttämällä Martinin kytkeytymis-, koheesio- ja kokomittoja.

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. [26] Virheen leviämisen todennäköisyys on riippuvainen komponentin muista ominaisuuksista, ja on tehokkaampaa keskittyä näihin ominaisuuksiin kuin uuden johdetun mitan kehittämiseen. Esimerkiksi muutoksen propagoitumista voidaan arvioida osin Martinin kytkeytymismetriikalla ja sivulla 105 määritellyllä kapseloinnin mittarilla.

Perintä

Luokkajoukoille perintä ei varsinaisesti tarkoita mitään, sillä yksittäisen luokan ulkopuolelle perintä näkyy vain tavallista voimakkaampana kytkeytymisenä. Siitä huolimatta Ducasse ym. [57] jakoivat komponenttitasolla perinnän tarkastelun kolmeen osaan: komponentin sisäiseen hierarkiaan, perimiseen komponentin ulkopuolisesta yliluokasta ja periytymiseen komponentin ulkopuoliselle lapselle.

Periminen ja periytyminen komponentin ulkopuolelle ovat vain yksi kytkeytymisen muoto. Ducasse ym. [57] määrittelemien periytymismittojen tarkoitus oli havainnollistaa komponentin tarkoitusta yksinkertaisessa säteittäiskaaviossa. Niiden ei ollut alun perinkään tarkoitus toimia ulkoisen laadun mittareina. Näiden kahden kategorian mitat voidaan korvata esimerkiksi Martinin [116] lähtevien ja saapuvien kytkösten mitoilla.

Ducassen, Lanzan ja Ponsion [57] kolmas komponenttitason perintäkategoria on komponentin sisäinen perintähierarkia. He ehdottivat komponentin sisäisten perintärelaatioiden määrän tarkasteluun PIIR:iä. Empiirisissä tutkimuksissa perintämetriikat ovat kuitenkin menestyneet huonosti alemmalla tasolla, ja niiden uskotaan heijastavan enemmän suunnittelijoiden olioymmärrystä kuin virhealttiutta. [37] Samoin on epäselvää mitä komponentin perintärelaatioiden määrä kertoo. Perintärelaatioiden määrän perusteella komponentteja ei voida järjestää laadun tai virhealttiuden mukaiseen järjestykseen. PIIR:n kor-

kea arvo voi kertoa komponentin hyvästä kiinnevoimasta, mutta myös tätä ominaisuutta pystytään arvioimaan tehokkaammin komponentin koheesiometriikoilla. Perintämetriikoille ei näyttäisi löytyvän tarkoitusta tai käyttöä komponenttitasolla.

Riskialttius

Riskialttius arvioi komponentin muutoksen tai virheen aiheuttamaa työmäärää koko järjestelmän ylläpitämiselle. Riskialttiimmat komponentit pitäisi tutkia ja testata muita tarkemmin. Ominaisuus on kuitenkin hyvin abstrakti, ja siihen vaikuttavat niin muutosten propagoituminen, kytkeytyminen kuin kapseloinnin onnistuminen. Esimerkiksi Yacoub ym. [174] johtivat riskialttiuden 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 Martinin määrittelemiä kytkeytymis-, koheesio- tai jopa kokomittoja.

Stabiilius

Stabiilius ei tarkoita komponentin pysyväisyyttä, vaan tilan muuttamiseen vaadittavaa työmäärää. [118] Vakaalla komponentilla on vähän muutospaineita ja paljon vastuuta. Martin [116] esimerkiksi mittasi komponentin vastuita siihen saapuvien ja muutospaineita siitä lähtevien kytkösten määrällä. Stabiilien komponenttien tunnistaminen on keskeistä arkkitehtuurisuunnittelussa, sillä vakaan riippuvuuden periaatteen mukaan komponenttien tulisi olla kytkeytyneinä ainoastaan vakaisiin komponentteihin.

Suunnitteluperiaatteiden mukaan luodut stabiilit komponentit ovat myös uudelleenkäytettäviä, sillä niillä on vain vähän ulkoisia riippuvuuksia. Lisäksi vakaat komponentin ovat asiakasmäärän perusteella keskeisiä arkkitehtuurissa, miksi ne tulisi priorisoida testauksessa korkealle. Martinin [116] esittelemällä epästabiiliuden mitalla voidaan arvioida komponenttien vakautta.

Uudelleenkäytettävyys

Uudelleenkäytettävyys on yksi komponenttijaon perusteista. [102] Ominaisuutta voitaisiin käyttää esimerkiksi testauksen ja kehityksen priorisoimisessa sellaisiin komponentteihin, joiden tiedetään soveltuvan uudelleenkäytettäväksi. Kierrätettyihin komponentteihin myöhemmin tehtävien muutosten 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.

Esimerkiksi koheesio ja kytkeytyminen [144], stabiilius sekä kapselointi vaikuttavat komponentin uudelleenkäytettävyyteen. Lisäksi komponentin palveluiden on sovittava muiden järjestelmien käytettäväksi. Puhtaan uudelleenkäytettävyyden arvioiminen saattaa muodostua liian monimutkaiseksi ja epävarmaksi mitaksi, minkä vuoksi keskittyminen muihin kierrätettävyyttä tukeviin ja komponentin laatua parantaviin ominaisuuksiin on tehokkaampaa. Ominaisuutta voisi yrittää mitata esimerkiksi Martinin [120] kytkeytymis-, koheesio- ja stabiiliusmitoilla.

Yhteenveto komponenttien ominaisuuksista

Tässä alaluvussa on tarkasteltu komponenttien mitattavia piirteitä. Näitä ominaisuuksia on tunnistettu komponentin määritelmistä, toiminnallisuudesta sekä ehdotetuista komponenttimitoista. Usea arvioiduista ominaisuuksista on äärimmäisen abstrakti käsite, ja sitä 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 metriikoiden määrää tulisi myös rajata, sillä todennäköi-

sesti suuresta joukosta mittoja osa korreloi keskenään ja paljastaa samanlaisia virheitä. Vaikka metriikkaohjelmat laskevat kymmenien metriikoiden arvoja hetkessä, tällaisten tuloslistausten tulkitseminen on työlästä ja hankalaa. Käyttäjän pitäisi ymmärtää jokaisen metriikan tarkoitus, sen mahdolliset ongelmat ja menetelmät huonon arvon korjaamiseksi. Huomattavasti käyttäjäystävällisempää on kehittää pieni, tehokas ja kattava joukko komponenttimetriikoita, joita on helppo ymmärtää ja soveltaa.

Alaluvussa listatut ominaisuudet on tiivistetty taulukkoon 4.3. Tässä opinnäytteessä komponenttien keskeisimmiksi piirteiksi on tunnistettu abstraktius, kapselointi, koheesio, koko, kytkeytyminen ja stabiilius. Kirjallisuudessa esitetyistä komponenttimetriikoista erityisesti Martinin esittelemät mitat kattavat useimmat edellä mainituista ominaisuuksista. Näitä mittoja voitaisiin käyttää perustana komponenttimetriikkapaketin määrittelemiseksi. Luku 5 keskittyykin juuri Martinin mittojen kelpuuttamiseen, jotta niiden voitaisiin teoreettisesti osoittavan soveltuvan komponenttimetriikoiksi.

4.3.2 Järjestelmä

Monia ominaisuuksia voidaan mielekkäästi mitata vain järjestelmästä yhtenä kokonaisuutena. Esimerkiksi ISO 9126 [59] standardin laatuominaisuuksien, kuten luotettavuuden tai tehokkuuden, tarkasteltuun tarvitaan koko ohjelmistoa. Järjestelmätason ei kuitenkaan tarvitse tietää luokkien ja metodien rakennetta tai informaatiosisältöä, vaan tason tarkoituksena on keskittyä komponenttien välisiin suhteisiin. [2]

Abreun [9] kytkeytymiskerroin sekoittaa arkkitehtuuri- ja luokkatasojen rooleja. Mitta laskee kuinka kytkeytyneitä kaikki luokat ovat keskenään, suhteutettuna täysin kytkeytyneeseen järjestelmään. COF:n huono arvo vaatisi koko järjestelmän refaktorointia ja riippuvuussuhteiden purkamista, yksittäisten komponenttien korjaamisen sijasta. Toinen ongelma on informaation määrässä ja tarkkuuden puuttumisessa. Isoissa järjestelmissä mittojen tulokset keskiarvoistuvat, minkä vuoksi ongelmien löytäminen vaikeutuu. Sadoista luokista koostuvassa järjestelmässä on epätodennäköistä saada COF:ltä kriittisiä tuloksia,

Taulukko 4.3: Komponenteille tunnistetut ominaisuudet ja Martinin metriikkapaketin soveltuvuus ominaisuuksille. Taulukossa on myös mainittu muutamia vaihtoehtoisia mittoja ominaisuuksille, sekä millä ominaisuuksilla johdettu piirteitä voisi arvioida.

Ominaisuus	**Martin** [120]	**Vaihtoehtoja**
Abstraktius	A	
Kapselointi	—	IUR (s. 105)
Koheesio		
Sisäinen	H	LCOP (s. 106)
Asiakas	—	CPC [144], SCC [175]
Koko	N_c, N_a	LOC (s. 6)
Kompleksisuus	Voidaan korvata esim. koheesio-, koko- ja kytkeytymismetriikoilla.	
Kytkeytyminen		
Tulevat	C_a	
Lähtevät	C_e	
Modulaarisuus	Voidaan korvata koheesio-, kytkeytymis- ja kokometriikoilla.	
Muutoksen propagoituminen	Voidaan korvata kytkeytymis- ja kapselointimetriikoilla.	
Perintä	Voidaan korvata kytkeytymismetriikoilla.	
Riskialttius	Voidaan korvata kytkeytymis-, koheesio- ja kokometriikoilla.	
Stabiilius	I	
Uudelleenkäytettävyys	Voidaan korvata koheesio, kytkeytymis- ja stabiiliusmetriikoilla.	

sillä se vaatisi jokaisen järjestelmän luokan kytkeytymistä kymmeniin muihin.

Arkkitehtuurityylin noudattaminen

Arkkitehtuurityylit määräävät komponenteille rooleja, joiden pohjalta järjestelmän toiminnallisuus rakentuu. [102] Esimerkiksi MVC-arkkitehtuurissa on selvästi luokiteltavissa komponenteille kolme eri kategoriaa. Yksi järjestelmätason metriikoiden tavoitteista olisi mitata kuinka hyvin yksittäiset komponentit ja arkkitehtuuri kokonaisuudessaan noudattavat määritellyn tyylin mukaista suunnitelmaa. Mitalla pystyttäisiin havaitsemaan arkkitehtuurin rämettyminen jo kehitysvaiheen aikana, jolloin suunnittelun korjaaminen on helpompaa kuin ylläpitovaiheessa.

Arkkitehtuurityylien ja mittojen välistä suhdetta ei ole määritelty. Pitäisikö esimerkiksi tietovarastoarkkitehtuurin kokonaiskytkeytymisen olla pienempi kuin kerrosarkkitehtuuriin pohjautuva vastaava järjestelmä? Puhtaassa tietovarastossa asiakkaat ovat tietoisia ainoastaan palvelimesta, kun kerrosarkkitehtuurissa abstraktiotaso on vahvasti riippuvainen alemman kerroksen palvelurajapinnasta. Vastaavasti arkkitehtuurityyli ohjaa myös muita komponenttien ja luokkien ominaisuuksia.

Yksittäisistä arkkitehtuurityyleistä pitäisi tunnistaa keskeisimmät tyyliä tukevat ominaisuudet ja rajoitteet, joiden arvioimisella voitaisiin osoittaa toteutuneen arkkitehtuurin ongelmakohtia. Valitettavasti arkkitehtuurityylit esiintyvät harvoin puhtaina. Esimerkiksi MVC-arkkitehtuurin muunnelmissa näkymän ja kontrollerin roolit on yhdistetty suunnittelun yksinkertaistamiseksi. [102] Määrittelyjen muuttuessa tyyliä tukevien ominaisuuksien tunnistaminen vaikeutuu.

Arkkitehtuurityylit tukevat järjestelmän hahmottamista [102], miksi tyylien noudattamattomuus vaikeuttaa järjestelmän ymmärtämistä ja ylläpidettävyyttä. Samat ominaisuudet osallistuvat epäsuorasti järjestelmän ulkoiseen laadukkuuteen, sillä huonosti ymmärretyssä järjestelmässä on helpompi tehdä virheitä aikaansaavia ratkaisuja. Arkkitehtuurityylit vaikuttavat ohjelman laatuun, mutta kaikki suunnittelusta poikkeavat ratkaisut eivät

huononna järjestelmän laatutekijöitä. Esimerkiksi kerrosarkkitehtuurin ohituksia perusteellaan tehokkuuden parantumisella. [102] Arkkitehtuurityylien onnistumisen arvioiminen mittaamalla on lupaava alue järjestelmämetriikoille, mutta vaatii vielä ominaisuuksien ja rajoitteiden kartoittamista.

Koheesio

Järjestelmän kiinnevoimaisuutta voidaan tarkastella vastaavasti kuin yksittäisellä komponentilla. LCOP(SS):ssä tarkastellaan komponenttijoukon riippuvuusgraafia G_{SS}. Graafin tiheyden arvioimiseen voidaan vastaavasti käyttää suoraan Hitzin ja Montazerin [81] Co:ta. Koheesiomitan liiallinen optimointi riippuvuuksien määrillä tai tyypeillä ei ole hyödyllistä järjestelmälle, sillä jokaisen yksittäisen komponentin kytkeytyminen muihin yritetään samalla minimoida. Riippuvuusgraafin yhtenäisyyden ja tiheyden tarkastelu riittää kertomaan onko toisiaan käyttävät osat sijoitettu samaan alijärjestelmään.

Koko

Kuten yksittäisille ohjelmakomponenteille, myös järjestelmätasolle koko on yksi tärkeimmistä ominaisuuksista. Kokoa voidaan mitata tasolla komponenttien määrällä, sillä luokkien määrä kertoo enemmän yksittäisestä komponentista kuin järjestelmän osasta. Vastaavasti kuin komponenttitasolla, myös järjestelmästä on suositeltavaa laskea LOC. Isoja ohjelmistoja on vaikea hahmottaa pelkän LOC:n perusteella, mutta lukumäärää hyödynnetään monessa johdetussa mitassa. Esimerkiksi virhetiheyksiä ilmoitetaan vikojen suhteena tuhanteen lähdekoodiriviin. [63]

Kytkeytyminen

Järjestelmä voi muodostua muista osajärjestelmistä, ja näiden välistä kytkeytymistä voidaan tarkastella vastaavasti kuin yksittäisen komponentin kytkeytymistä. Asiakkaiden määrää voidaan arvioida joko joukosta riippuvien komponenttien tai alijärjestelmien mää-

rällä. Vastaavasti komponenttijoukon riippuvuuksia muista järjestelmän osista tai ulkoisista palveluista voidaan arvioida kytkösten määrällä.

Suunnittelumallien käyttö

Järjestelmän laatua voidaan yrittää mitata myös tunnistamalla käytettyjä suunnittelumalleja ja niiden määriä. [74, 140] Esimerkiksi Khaer ja kollegat [98] löysivät empiirisessä tutkimuksessaan riippuvuuksia arvioidun laadun ja suunnittelumallien määrän välillä. Lähestymistapa tuntuu kuitenkin omituiselta, sillä jokaiseen suunnittelumalliin liittyy Gamman ym. [68] mukaan sekä hyötyjä että haittoja. Malleja tulisi käyttää ainoastaan saavutettavien etujen ollessa suuremmat kuin aiheutuvat vahingot. Suunnittelumallien summittainen sijoittelu ympäri järjestelmää ei paranna arkkitehtuuria tai ohjelman laatua, ja näin väärinkäytettyinä niihin perustuvat metriikat eivät ole luotettavia laadun mittareita.

Huomattavasti tehokkaampaa olisi tunnistaa arkkitehtuurista suunnittelun antimalleja. [115] Antimallien esiintymistä järjestelmässä voidaan pitää selvänä merkkinä huonosta suunnittelusta, mutta suunnittelumallien esiintyminen ei automaattisesti tarkoita hyvää laatua. MAISA-projektin pyrkimyksenä oli tunnistaa suunnittelu- ja antimalleja arkkitehtuurista [140], mutta ohjelman tunnistamissa malleissa ei ole mainittu yhtään antisuunnittelumallia. [73] Osa antimalleista saattaa olla vaikea tunnistaa, esimerkiksi Kyhäelmällä ei ole selkeästi mitattavia tuntomerkkejä. Tosin esimerkiksi Jumalaluokan (engl. *God class*) tunnistaminen pitäisi olla taas helppoa. Tässä antimallissa yksittäinen luokka toteuttaa suurimman osan ohjelman toiminnallisuudesta, ja se pitäisi refaktoroida useammaksi luokaksi. [38] Tunnistamisessa voitaisiin käyttää esimerkiksi luokan suhteellista kokoa järjestelmässä tai komponentissa. Määrätyt rajat ylittävät luokat voitaisiin ohjata tarkempaan tarkasteluun ja mahdolliseen refaktorointiin.

Antisuunnittelumallien määrällä ei tarvitse esitellä erillistä mittaa — näiden lukumäärän ilmoittaminen riittää. Jokainen tunnistettu antimalli pitäisi poistaa refaktoroimalla. Brown ym. [38] antoivat jokaisen esittelemänsä antimallin yhteydessä myös toiminta- ja

korjausohjeet ongelman poistamiseksi. Antisuunnittelumallin määrän laskemisessa suurin ongelma on niiden tunnistamisen automatisoitavuus. Tosin niin suunnittelu- kuin antimallien koneellista tunnistamista on tutkittu viimeaikoina enenevässä määrin. Tekniikan kehittyminen saattaa mahdollistaa piankin antimallien määrän tehokkaan arvioinnin.

Viittaussyklit

Riippuvuusgraafin viittaussyklit ovat haitallisia niin komponenttien uudelleenkäytettävyydelle, testaamiselle kuin kehitykselle. [120] Martin [117] määrittelikin yhdeksi arkkitehtuuritason ohjenuoraksi *Syklittömien riippuvuuksien periaatteen* (engl. *Acyclic Dependencies Principle*, ADP) ja esitteli refaktoroinnin, jolla riippuvuuksien suunta käännetään rajapintojen avulla.

Syklien havaitseminen ja poistaminen on tärkeä osa ohjelmiston suunnittelua, ja niiden löytämiseksi on esitelty useita mittoja. Hautuksen PASTA:n [79], Lakoksen mittojen [105] sekä Meltonin ja Temperon [124] CRSS:n lähestymistavat syklien löytämiseen ovat liioitellun monimutkaisia. Mallinnettaessa komponentteja graafina, voidaan käyttää Lakshmi Narasimhan ja Hendradjayan [106] esittelemää syklien lukumäärää (engl. *Number of Cycles*, NC). Mitta yksinkertaisesti laskee viittausgraafista löytyvien riippuvuussyklien lukumäärän. Löytyneet syklit voidaan refaktoroida Martinin [120] esittelemällä menetelmällä, jossa riippuvuussuhteet käännetään rajapintojen avulla.

Yhteenveto tarkastelluista ominaisuuksista

Edellä on esitelty järjestelmän mitattavia piirteitä, joita on tunnistettu järjestelmän määritelmästä ja sille esitetyistä mitoista. Taulukko 4.4 tiivistää alaluvussa käsitellyt ominaisuudet. Osa ominaisuuksista on vastaavia, mitä alemmilla tasoilla on esitetty. Muutama ominaisuus esiintyy vain järjestelmätasolla. Esimerkiksi arkkitehtuurityylin noudattamisen arvioiminen auttaisi arkkitehtuurin onnistumisen seuraamista, mutta ominaisuudella ei ole selkeitä piirteitä, joilla sitä voisi mitata. Tällaisten mittojen luominen vaatii vielä

Taulukko 4.4: Järjestelmätason ominaisuuksia ja näiden arvioimiseen sopivia mittoja.

Ominaisuus	Ehdotetut mitat
Tyylinmukaisuus	–
Koheesio	LCOP (s. 121)
Koko	LOC (s. 6) Komponenttien lukumäärä (s. 121)
Kytkeytyminen	
Tulevat	Komponenttien tai järjestelmien määrä (s. 121)
Lähtevät	Komponenttien tai järjestelmien määrä (s. 121)
Suunnittelumallien käyttö	Suunnittelumallien määrä ei mittaa järjestelmän laatua, mutta antisuunnittelumallien määrä ilmaisee sen ongelmia. Antimalleja voidaan arvioida puhtaasti niiden määrällä. (s. 122)
Viittaussyklit	NC [106]

huomattavasti työtä.

Kytkeytyminen, koheesio ja koko ovat vastaavia ominaisuuksia, joita on alemmille tasoille määritelty. Samoin näille ehdotettavat mitat ovat samankaltaisia kuin komponentti- ja luokkatasolla. Mitat ovat hyvin suoraviivaisia ja yksinkertaisia. Kuitenkin näiden soveltuvuus tulisi arvioida vielä sekä teoreettisella että empiirisellä tarkastelulla.

Luku 5

Arkkitehtuurimetriikan kelpuuttaminen

Ohjelmistometriikan kelpuuttamisen (engl. *validation*) tarkoituksena on varmistaa mitan oikeellisuus ja käyttökelpoisuus. Metriikan hyväksymisessä on kaksi vaihetta: Teoreettisessa vahvistamisessa osoitetaan, että mitta mallintaa sitä ominaisuutta, jota sen pitäisi mitata. Empiirisessä hyväksymisessä osoitetaan mitan käyttökelpoisuus yhdistämällä sen arvot ulkoisiin laatuominaisuuksiin. [33]

Martinin metriikka on toteutettu useissa mittojen laskentaohjelmassa, mutta mittaa ei ole koskaan kelpuutettu. Luvussa 4 esitellyistä mitoista ainoastaan Ponision CPC:tä on arvioitu teoreettisesti. [144] Komponenttimetriikoiden suhdetta ulkoisiin laatuominaisuuksiin ei ole tutkittu, vain muutamien yksittäisten mittojen vaikutusta on kartoitettu.

Tässä luvussa käsitellään tunnetuimman komponenttimetriikan teoreettista ja empiiristä kelpuuttamista. Martinin osametriikat kattoivat suurimman osan edellisessä luvussa esitellyistä ominaisuuksista ja niiden soveltuvuutta yleisiksi komponenttimetriikoiksi arvioidaan seuraavaksi. Alaluvussa 5.1 tarkastellaan Martinin metriikan teoreettista taustaa. Mitalle toteutettua empiiristä mittausta esitellään alaluvussa 5.2. Luvun lopussa pohditaan vielä metriikan soveltuvuutta arkkitehtuurin arvioimiseen.

5.1 Teoreettinen testaaminen

Teoreettinen testaaminen pyrkii varmistamaan, että ehdotetut ohjelmistomitat noudattavat ominaisuuksien oletettuja piirteitä. Tarkoituksena on vahvistaa, että metriikka todella mittaa tarkoitettua ominaisuutta, eikä ohjelman satunnaisia osia. [33] Kehyksiä ominaisuuksille on ehdotettu muutamia, joista tunnetuimpien joukossa ovat Weyukeriin [170] kompleksisuusehdot ja Briandin, Morascan sekä Basilin [35] kriteerit.

Weyuker [170] määritteli vuonna 1988 yhdeksän vaatimusta kompleksisuusmitoille. Esimerkiksi McCaben syklomaattinen kompleksisuus toteuttaa näistä ehdoista vain viisi. Oliometriikoista sekä Chidamberin ja Kemererin [47] metriikkapakettia että Ponision [144] CPC:tä on yritetty kelpuuttaa Weyukerin vaatimuksilla. Tosin näistä mitoista ainoastaan WMC:tä voi luonnehtia kompleksisuusmitaksi.

Kaikkia metriikoita ei voida kohdella monimutkaisuusmittoina. [35] 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 ja kollegat [35] esittelivät koheesiolle, kytkeytymiselle, kompleksisuudelle ja koolle erilliset vaatimukset, jotka soveltuvat yleisiä kompleksisuusehtoja paremmin näiden ominaisuuksien mittojen arvioimiseen. Seuraavassa arvioidaan jokaista Martinin metriikan osamittaa erikseen Briandin ym. kriteereillä. Osamitoista muodostettu Martinin metriikkapaketti on lueteltu taulukossa 5.1.

Briand ym. [33] määrittelivät mitoille yleisiä ominaisuuksia, joita vasten Martinin metriikkapakettia verrataan alaluvussa 5.1.2. Paketin kahta kytkeytymismittaa koetellaan kytkeytymiskehyksessä, joka esiteltiin alaluvussa 2.4.3. Samoin Martinin koheesiometriikkaa tarkastellaan alaluvussa 2.3.4 esitellyssä koheesiokehyksessä.

Taulukko 5.1: Martinin metriikkapaketti.

Mitta	Kuvaus	Tyyppi	Yhtälö
C_a	Saapuvat kytkökset	Kytkeytyminen	
C_e	Lähtevät kytkökset	Kytkeytyminen	
I	Epästabiilius		$\frac{C_e}{C_a+C_e}$
N_a	Abstraktien luokkien lukumäärä	Koko	
N_c	Luokkien lukumäärä	Koko	
A	Abstraktius		$\frac{N_a}{N_c}$
D	Etäisyys pääsarjasta		$\frac{\|A+I-1\|}{\sqrt{2}}$
D'	Normalisoitu etäisyys		$\|A+I-1\|$
H	Suhteellinen koheesio	Kiinnevoima	$\frac{R+1}{N_c}$

5.1.1 Metriikoiden kriteerit

Briand ym. [35] ehdottivat kytkeytymismitoille viittä, koheesiomitoille neljää ja kokomitoille kolmea kriteeriä, jotka näiden tulisi täyttää. Määritellyistä ehdoista kolme ovat kaikille samoja. Martinin paketista kokomittoja ovat N_a ja N_c, vastaavasti C_a ja C_e mittaavat selvästi kytkeytymistä. Paketin ulkopuolinen H on Martinin esittelemä koheesiomitta, joten myös se sisällytettiin tarkasteluun. Epästabiiliuden, abstraktiuden ja etäisyyden mitat ovat ongelmallisempia, koska vastaaville ominaisuuksille ei ole määritelty kriteereitä. A:ta, I:tä ja D:tä voidaan kuitenkin tarkastella vasten Briandin ym. kahta yleisintä kriteeriä. D':a ei tarkastella, sillä se on vain skaalattu D.

Kriteereiden määrittelyissä P on mielivaltainen komponentti ja $Metric(P)$ testattava mitta. $Rel(P)$ on testattavan metriikan ulkoisten riippuvuuksien joukko. Se voi edustaa komponentin tulevia, lähteviä tai molempia riippuvuuksia. $IR(P)$ on komponentin sisäisten kytkösten joukko, jonka alkiot määräytyvät testattavan mitan mukaan. Komponentin osien, esimerkiksi luokkien ja rajapintojen, joukko on $Size(P)$.

Kokometriikoiden kriteerit

Briandin ym. [35] antoivat kokometriikoille kolme kriteeriä: *Ei-negatiivisuuden* mukaan kokometriikan arvon on oltava aina positiivinen, mikä on intuitiivisesti selkeä vaatimus. Kokomitalla on myös oltava *nolla-arvo*. Kolmannen kriteerin mukaan *kahden komponenttien yhdistämisestä* muodostuneen komponentin koon on oltava edellisten kokojen summa. Seuraavassa on tarkasteltu N_a:ta ja N_c:tä erikseen kolmen kriteerin suhteen:

1. *Ei-negatiivisuus*. (engl. *Nonnegativity*)
Komponentin P koko ei voi olla negatiivinen:

$$Metric(P) \geq 0$$

Selvästi N_a ja N_c täyttävät ehdon, sillä kumpikaan ei voi saada negatiivisia arvoja. □

2. *Nolla-arvo*. (engl. *Null value*)
Komponentin P kokomitan arvo on nolla kun $Size(P)$ on tyhjä:

$$Size(P) = \emptyset \;\Rightarrow\; Metric(P) = 0$$

$Size(P)$ on N_c:lle kaikkien luokkien joukko. Selvästi mitan arvo on nolla, kun $Size(P)$ on tyhjä joukko. Vastaavasti $Size(P)$ on N_a:lle abstraktien luokkien joukko. Myös $N_a = 0$ kun komponentissa P ei ole yhtään abstraktia luokkaa. N_c ja N_a selvästi täyttävät toisen kriteerin. □

3. *Komponenttien yhdistäminen*. (engl. *Module Additivity*)
Olkoon P' komponenttien P_1 ja P_2 yhdiste. Jos P_1:llä ja P_2:lla ei ole yhteisiä piirteitä, niin silloin:

$$Metric(P_1) + Metric(P_2) = Metric(P')$$

Selvästi N_c:lle ehto on voimassa, sillä kahdella komponentilla ei voi olla yhtenäisiä luokkia. Tällöin P':n luokkien määrä on P_1:n ja P_2:n luokkien määrän summa. Vastaavasti

kriteeri on voimassa N_a:lle, sillä komponenteilla ei voi olla yhtenäisiä abstrakteja rakenteita.

Ehto ei ole voimassa komponenteille, joilla on yhteisiä piirteitä. Esimerkiksi jos komponenttiin yhdistetään sen alikomponentti, on komponenteilla yhteisiä piirteitä. Mittojen määrittelyissä tällaisia tilanteita voidaan käsitellä useilla eri tavoilla, eikä vaihtoehtoja tarkastella tässä opinnäytteessä. □

Kytkeytymismetriikoiden kriteerit

Briandin ym. [35] antoivat kytkeytymismetriikoille viisi kriteeriä. *Ei-negatiivisuus* ja *nolla-arvo* ovat vastaavia, mitä kokometriikoille on määritelty. *Monotonisuuden* mukaan komponentin kytkösten määrän kasvaessa pitää myös metriikan arvon kasvaa. *Yhdistettäessä komponentteja* ei uuden komponentin kytkeytymismitan arvo voi olla suurempi kuin aiempien komponenttien arvojen summa. Jos *kytkeytymättömiä komponentteja yhdistetään*, uuden komponentin kytkeytymismitan arvo on aiempien komponenttien arvojen summa. Seuraavassa on verrattua C_a:ta ja C_e:tä erikseen jokaisessa viidessä kriteerissä:

1. *Ei-negatiivisuus*. (engl. *Nonnegativity*)
Komponentin P kytkeytymisen arvo on ei-negatiivinen:

$$Metric(P) \geq 0$$

Selvästi C_a ja C_e eivät voi saada negatiivisia arvoja, joten ne täyttävät ensimmäisen kriteerin. □

2. *Nolla-arvo*. (engl. *Null value*)
Komponentin P kytkeytymismitan arvo on nolla kun $Rel(P)$ on tyhjä:

$$Rel(P) = \emptyset \quad \Rightarrow \quad Metric(P) = 0$$

Selvästi sekä C_a että C_e voivat saada arvon nolla, jos komponentilla ei ole tulevia tai lähteviä riippuvuuksia. Näin myös toinen kriteeri on voimassa mitoille. □

3. *Monotonisuus*. (engl. *Monotonicity*)

Olkoon P' komponentin P kanssa täysin yhtäläinen muuten, mutta $Rel(P) \subseteq Rel(P')$. Silloin:

$$Metric(P) \leq Metric(P')$$

Jos komponentin tulevien tai lähtevien kytkösten määrä lisääntyy, kasvavat C_a ja C_e samassa suhteessa. Selvästi C_a ja C_e täyttävät myös kolmannen kriteerin. □

4. *Komponenttien yhdistäminen*. (engl. *Merging of packages*)

Olkoon P' komponenttien P_1 ja P_2 yhdiste. Silloin:

$$Metric(P_1) + Metric(P_2) \geq Metric(P')$$

Jos $P_1 \notin Rel(P_2)$ tai $P_2 \notin Rel(P_1)$ niin *kytkemättömien komponenttien yhdistämisen* vaatimus on voimassa. Muulloin sekä $C_a(P') < C_a(P_1)+C_a(P_2)$ että $C_e(P') < C_e(P_1)+C_e(P_2)$, sillä riippuvuuksien määrä vähenee vähintään yhdellä komponentteja yhdistettäessä. C_a ja C_e täyttävät neljännen ehdon, jos myös viides kriteeri on voimassa mitoille. □

5. *Kytkemättömien komponenttien yhdistäminen*. (engl. *Merging of unconnected*)

Olkoon P' komponenttien P_1 ja P_2 yhdiste. Jos P_1:llä ja P_2:llä ei ole keskinäisiä riippuvuuksia, niin silloin:

$$Metric(P_1) + Metric(P_2) = Metric(P')$$

Selvästi $C_a(P') = C_a(P_1)+C_a(P_2)$ koska uuden komponenttien riippuvuuksien määrä ei vähene tai kasva yhdistettäessä kytkeytymättömiä komponentteja. Vastaavasti $C_e(P') = C_e(P_1) + C_e(P_2)$ on voimassa kun $P_1 \notin Rel(P_2)$ ja $P_2 \notin Rel(P_1)$. C_a ja C_e täyttävät viidennen ehdon. Koska *komponenttien yhdistäminen* on riippuvainen viidennestä ehdosta, täyttävät C_a ja C_e myös neljännen ehdon. □

Kiinnevoiman metriikoiden kriteerit

Briandin ym. [35] antoivat koheesiometriikoille neljä kriteeriä, joita Briand, Daly ja Wüst [32] vielä tarkensivat. Tässä tarkastelussa käytetään uudempia kriteereitä, joihin on lisätty maksimiarvon tarkastelu. *Ei-negatiivisuus ja normalisointi* on ensimmäinen kriteeri ja se vaatii koheesiomitalta kiinteästi määriteltyjä ala- ja ylärajoja. Alaraja on erikseen määrätty nollaksi. Toinen kriteeri *Nolla- ja maksimiarvo* vaatii, että mitta on nolla, kun sisäisiä relaatioita ei ole. Toisen kriteerin mukaan mitan on myös saavutettava maksimi, kun tarkasteltavassa kohteessa on kaikki relaatiot. Kolmas kriteeri *Monotonisuus* on vastaava, mitä kytkeytymismetriikoille määriteltiin. Neljännen kriteerin, *Komponenttien yhdistämisen*, mukaan kahden yhdistetyn komponentin koheesioarvo ei voi olla suurempi kuin suuremman alkuperäisistä. Intuitiivisesti on selvää, että lisättäessä komponenttiin luokkia sen kiinnevoimaisuus heikkenee. Seuraavassa on verrattu H:ta jokaisessa neljässä kriteerissä:

1. *Ei-negatiivisuus ja normalisointi*. (engl. *Nonnegativity and Normalization*)
Koheesiomitan arvo komponentille P kuuluu suljetulle arvoalueelle:

$$Metric(P) \in [0, \max]$$

H ei voi saada arvoa 0, sillä kaavan osoittaja on aina vähintään 1. Mitalle ei myöskään ole määritelty kiinteää ylärajaa. Jos R laskisi kytkeytyneiden luokkaparien lukumäärää, vaihtelisi sen arvo välillä $[0, \frac{N_c(N_c-1)}{2}]$. Tällöin H:n suurin arvo on riippuvainen komponentin luokkien määrästä, eikä kriteerin vaatimasta kiinteästä ylärajasta. Vastaavasti mitan suurin arvo on riippuvainen luokkien määrästä, jos R määritellään laskemaan suunnattuja riippuvuuksia. H ei siis toteuta ensimmäistä ehtoa. □

2. *Nolla- ja maksimiarvo*. (engl. *Null value and maximum value*)
Komponentin P koheesiomitan arvo on nolla kun $IR(P)$ on tyhjä ja $\max$ kun $IR(P)$ on maksimaalinen:

$$IR(P) = \emptyset \quad \Rightarrow \quad Metric(P) = 0$$

$$IR(P) = maximum \quad \Rightarrow \quad Metric(P) = \max$$

Martinin mitta ei toteuta ehtoa, sillä kun $IR = \emptyset$ ja $N_c \neq 0$ on H:n arvo $\frac{1}{N_c}$. Kun $IR(P) = \max$ ei H saavuta metriikan maksimiarvoa, vaan tulos on riippuvainen komponentin luokkien määrästä. H ei täytä toista ehtoa. □

3. *Monotonisuus*. (engl. *Monotonicity*)

Olkoon P' komponentin P kanssa täysin yhtäläinen muuten, mutta $IR(P) \subseteq IR(P')$. Toisin sanottuna, kun komponentin sisäisten kytkösten määrä kasvaa, niin silloin:

$$Metric(P) \leq Metric(P')$$

Selvästi H toteuttaa ehdon. Jos R kasvaa, myös koheesiomitan arvo kasvaa. □

4. *Komponenttien yhdistäminen*. (engl. *Cohesive Modules*)

Olkoon P' keskenään kytkeytymättömien komponenttien P_1 ja P_2 yhdiste. Yhdistämällä luodun uuden komponentin kiinnevoiman arvo ei voi olla suurempi kuin P_1:n tai P_2:n arvo, eli:

$$\max\{Metric(P_1), Metric(P_2)\} \geq Metric(P')$$

Koska $R' = R_1 + R_2$ ja $N'_c = N_{c1} + N_{c2}$ niin $H(P') = \frac{R_1+R_2+1}{N_{c1}+N_{c2}}$. Oletetaan $H(P_1)$ suuremmaksi alkuperäisten komponenttien arvoista, silloin väite on:

$$\begin{aligned}
H(P_1) &\geq H(P') \\
\frac{R_1+1}{N_{c1}} &\geq \frac{R_1+R_2+1}{N_{c1}+N_{c2}} \\
\frac{(N_{c1}+N_{c2})(R_1+1)}{N_{c1}} - (R_1+1) &\geq R_2 \\
\frac{N_{c2}(R_1+1)}{N_{c1}} &\geq R_2 \\
\frac{R_1+1}{N_{c1}} &\geq \frac{R_2}{N_{c2}} \\
\frac{R_1+1}{N_{c1}} + \frac{1}{N_{c2}} &\geq \frac{R_2+1}{N_{c2}} \\
H(P_1) + \frac{1}{N_{c2}} &\geq H(P_2)
\end{aligned}$$

Olettamuksen mukaan ehto pitää paikkaansa, sillä $\frac{1}{N_{c2}} > 0$. H siis toteuttaa neljännen ehdon. □

H:n määritelmässä on ensimmäisen ja toisen ehdon kanssa kaksi ongelmaa. Ensinnäkin osoittajan yhteenlasku estää nollan saavuttamisen. Yhteenlaskun tarkoituksena oli estää huonot koheesioarvot komponenteilla, joilla on vain yksi luokka. [120] Tämä tosin voidaan toteuttaa kaksiosaisella määrittelyllä, jolloin osoittajan yhteenlaskusta voidaan luopua. Toinen ongelma on, että H:n nimittäjä voi olla osoittajaa pienempi. Tällöin H:n maksimi on riippuvainen komponentin luokkien määrästä. Ongelma voidaan poistaa määrittelemällä jakajaksi riippuvuuksien maksimi $\frac{N_c(N_c-1)}{2}$.

Määritellään mitasta uusi versio H' graafin avulla. Olkoon G_P komponentin P suuntaamaton riippuvuusgraafi. Graafin pisteinä ovat luokat ja kaaret ovat näiden välisiä riippuvuuksia. Pisteiden välillä on kaari, jos vähintään toinen piste käyttää toista eksplisiittisesti. Määritelmän mukaan kaksi luokkaa ovat kytkeytyneitä, jos niiden välillä on esimerkiksi käyttö- tai perintäsuhde. H' on tällöin:

$$H' = \begin{cases} 1 \\ \frac{R}{\frac{N_c(N_c-1)}{2}} \end{cases} = \begin{cases} 1 & \text{kun} \quad N_c = 1 \\ \frac{2R}{N_c(N_c-1)} & \quad N_c > 1 \end{cases} \tag{5.1}$$

Missä N_c on graafin pisteiden ja R kaarien määrä. Kun $N_c > 1$, H':n määrittely on identtinen Briandin, Dalyn ja Wüstin esittelemän Co':n kanssa [32]:

$$\text{Co'} = 2 \cdot \frac{|E|}{|V|(|V|-1)} \tag{5.2}$$

Missä E ja V ovat graafin kaarien ja pisteiden joukot. Briand ym. korjasivat alaluvussa 2.3.1 esitellyn Co:n teoreettisia ongelmia Co':lla. Vastaavasti H:n määritelmää korjatessa päädyttiin samanlaiseen lopputulokseen. Tosin kaksiosaisen määritelmän ansiosta H':lla on arvo, kun graafissa on vain yksi piste. Co' on tällöin määrittelemätön.

H' toteuttaa koheesiokriteerit. Mitan arvo on nolla kun $R = 0$. H':n maksimi on 1, joka saadaan kun graafissa on kaikki pisteet eli $R = \frac{N_c(N_c-1)}{2}$ tai kun $N_c = 1$. H' siis toteuttaa ensimmäisen ja toisen ehdon. Selvästi myös kolmas ehto on voimassa, sillä jos R kasvaa, myös H' kasvaa. Neljännen ehdon tarkastelu jakautuu kahteen osaan: Jos $N_{c1} = 1$ tai $N_{c2} = 1$, niin ehto on voimassa, koska $H(P')$ ei voi olla suurempi kuin 1.

Jos taas $N_{c1} \neq 1$ ja $N_{c2} \neq 1$ niin H' on sama kuin Co', jolle Briand ym. [32] todistivat kriteerien olevan voimassa. H' siis toteuttaa koheesiokriteerit.

Metriikoiden yleiset kriteerit

Stabiiliudelle tai abstraktiudelle ei ole määritelty vastaavia kriteereitä mitä esimerkiksi kytkeytymismetriikoille on annettu. Martinin metriikkapaketin I, A ja D pyrkivät kuitenkin selvästi mittaamaan komponenttien eri piirteiden määriä. Tällaisille mitoille voidaankin soveltaa kahta Briandin, Morascan ja Basilin [35] yleisintä kriteeriä: *Ei-negatiivisuus* vaatii, ettei komponentin piirteiden määrä voi olla negatiivinen. Intuitiivisesti on selvää, ettei esimerkiksi komponentin abstraktiusaste voi olla negatiivinen. Vastaavasti *nolla-arvon* kriteerin mukaan metriikan arvo on nolla, jos komponentilla ei ole yhtään mitattavaa piirrettä. Esimerkiksi jos komponentilla ei ole abstrakteja osia, sen abstraktiuden mitan pitäisi olla nolla. Seuraavassa tarkastellaan miten Martinin I, A ja D toteuttavat nämä kaksi vaatimusta:

1. *Ei-negatiivisuus*. (engl. *Nonnegativity*)
Mitan arvo on ei-negatiivinen komponentille P:

$$Metric(P) \geq 0$$

I on jakolaskun osamäärä, ja sen nimittäjä ($C_a + C_e$) sekä osoittaja (C_e) ovat aina ei-negatiivisia. I ei siis voi saada negatiivisia arvoja, ja näin toteuttaa ensimmäisen ehdon. Vastaavasti A ja D ovat jakolaskujen osamääriä, eivätkä niiden nimittäjät tai osoittajat voi saada negatiivisia aroja. Näin myös A ja D toteuttaa ensimmäisen ehdon. □

2. *Nolla-arvo*. (engl. *Null value*)
Komponentin P piirteitä laskevalla metriikka voi saada arvon nolla:

$$\exists\, P : Metric(P) = 0$$

I toteuttaa selvästi ehdon, sillä $I = 0$ kun komponentilla ei ole vastuita eli $C_e = 0$. Vastaavasti $A = 0$ kun komponentilla ei ole yhtään abstraktia luokkaa eli $N_a = 0$. D voi

saada arvon 0 kun $A + I = 1$. Koska A:n ja I:n arvot vaihtelevat välillä $[0, 1]$, tällaisia pisteitä on useita. Esimerkiksi $D = 0$ kun $A = 1$ ja $I = 0$. Selvästi siis I, A ja D toteuttavat toisen kriteerin. □

I ja A ovat määrittelemättömiä kun jakolaskun osoittaja on nolla. Esimerkiksi I on määrittelemätön kun $C_e = 0$ ja $C_a = 0$. Briand ym. [35] eivät kuitenkaan puutu epäjatkuvuuteen, vaan he ainoastaan velvoittavat metriikalta alarajaa. I:n ja A:n määrittelemättömät arvot osoittavat kuitenkin epäonnistunutta komponenttisuunnittelua. Esimerkiksi A on määrittelemätön kun komponentilla ei ole yhtään luokkaa. Selvästi tällaisia komponentteja ei saisi jäädä järjestelmään. Vastaavasti I:n määrittelemättömyyskohdassa kukaan ei käytä komponentin palveluita, eikä komponentti käytä kenenkään muun palveluita. Tällainen komponentti on selvästi täysin ylimääräinen osa järjestelmää, ja se voitaisiin poistaa.

Teoreettisten kriteerien yhteenveto

Briand, Morasca ja Basili [35] antoivat kokometriikoille kolme ja kytkeytymismetriikoille viisi kriteeriä. Martinin metriikkapaketin kokomitat N_c ja N_a toteuttavat kolme vaatimusta, ja vastaavasti kytkeytymismitat C_a ja C_e toteuttavat viisi ominaisuudelle asetettu vaatimusta. Kriteerien puolesta niitä voidaan pitää teoreettisesti kelvollisina metriikkoina.

Briand ym. [35] antoivat kiinnevoiman metriikoille neljä vaatimusta. Martinin H suoriutui kahdesta kriteeristä (*komponenttien yhdistäminen* ja *monotonisuus*), mutta ei täyttänyt kahta vaatimusta (*ei-negatiivisuus ja normalisointi* sekä *nolla- ja maksimiarvo*). Normalisoinnin epäonnistuminen tarkoittaa, ettei erikokoisten komponenttien arvoja voida vertailla keskenään. Ongelma on keskeinen. Esimerkiksi kahden luokan kiinnevoimaiselle komponentille H on 1. Jos viiden luokan komponentilla on vain viisi — puolet kaikista mahdollisista – kytköstä, sen $H = \frac{5+1}{5} = \frac{6}{5}$. Tulos on selkeästi harhaanjohtava, sillä jälkimmäisen kiinnevoima on huomattavasti ensimmäistä huonompi. Toinen keskeinen ongelma koheesiomitalle on relaatioiden, R:n, määritelmän puuttuminen. Vertailtaessa

eri komponenttien arvoja on tiedettävä myös käytetty laskentapa. Teoreettisessa tarkastelussa H on kelvoton koheesiometriikaksi. Samoin intuitiivinen käsityksen mukaan se on huonosti määritelty.

Luvussa esiteltiin myös H', joka yrittää korjata alkuperäisen H:n ongelmia koheesiokriteerien kanssa. Mitasta on poistettu osoittajasta yhteenlasku ja vaihdettu nimittäjän luokkien määrä suuntaamattomien relaatioiden maksimiin. Yllättävästi näin määritelty H' on lähes identtinen Co':n kanssa, jonka Briand ym. [32] esittelivät korjaamaan Co:n ongelmia koheesiokriteereiden kanssa. Erona on ainoastaan käyttäytyminen yhden pisteen graafilla. Määritelty H' toteuttaa kaikki neljä koheesiokriteeriä.

Epästabiilius, abstraktius ja etäisyys pääsarjasta puolestaan toteuttavat yleiset ehdot ei-negatiivisuudesta ja nolla-arvosta. Näiden kahden kriteerin perusteella ei kuitenkaan voida päätellä, ovatko mitat teoreettisesti kelvollisia ominaisuuksiensa mittareita. A:lle, I:lle ja D:lle on kuitenkin määritelty nolla-arvo, ala- sekä ylärajat, joten ne noudattavat johdettujen mittojen yleisiä piirteitä.

5.1.2 Kytkeytymis- ja koheesiokehykset sekä yleiset ominaisuudet

Briandin, Dalyn ja Wüstin kytkeytymis- [33] ja koheesiokehyksiä [32] voidaan käyttää mittojen teoreettisen taustan tarkastelemiseen. Niillä pyritään selvittämään mittojen määrittelyihin liittyviä ongelmia. Kehyksien avulla voidaan esimerkiksi arvioida, ottavatko metriikat huomioon mahdollisia erikoistapauksia. Eräs tällainen poikkeustapaus on kytkeytymiskehyksessä tarkasteltava perinnän huomioiminen.

Briand ym. [32, 33] uskovat vakaasti mittojen olevan hyödyllisiä vain, jos ne pystytään linkittämään ulkoisiin laatuominaisuuksiin. Tämän vuoksi he pyytävät kehittäjiä antamaan *mittaushypoteesin* eli nimeämään ulkoiset ominaisuudet, joihin metriikan mittaamat ominaisuudet vaikuttavat. Mittaushypoteesia on arvioitu Briandin ym. yleisillä ominaisuuksilla, jotka esitellään alaluvussa 5.1.2. Yleisillä ominaisuuksilla voidaan myös arvioida mitan käyttövaihetta ohjelmistokehitysprosessissa.

Seuraavaksi tarkastellaan Martinin metriikkapaketin osia kytkeytymis- ja koheesiokehyksissä sekä yleisillä ominaisuuksilla. Samalla myös selvitetään mitä tarkastelu paljastaa metriikoiden määrittelystä, ja miten mittoja voisi kehittää. Tarkasteluun sisällytetään myös esitelty H', jotta alkuperäistä koheesiomittaa voidaan verrata sitä vasten.

Kytkeytymiskehys

Briandin, Dalyn ja Wüstin [33] kytkeytymiskehys koostuu kuudesta kriteeristä, joiden avulla kytkeytymismittoja voidaan luokitella ja vertailla. Kytkeytymiskehystä on tarkasteltu tarkemmin alaluvussa 5.1.2. Briand ja kollegat mainitsivat Martinin kytkeytymismetriikat esitellessään kehystään, mutta niiden puutteellisen määrittelyn vuoksi he eivät käsitelleet tai luokitelleet niitä. Seuraavassa on tarkasteltu C_a:n ja C_e:n soveltuvuutta kytkeytymiskehykseen ja taulukossa 5.2 on tiivistelmä Martinin kytkeytymismetriikoiden piirteistä kehyksessä.

Kytkeytymisehto. Mikä kytkee tarkasteltavat osat toisiinsa? Martinille [116] luokka on kytkeytynyt komponenttiin, jos niiden välillä on tarkemmin määrittelemätön relaatio. Briand ym. [33] kritisoivat juuri kytkeytymisehdon puuttumista, ja ohittivat sen vuoksi mitan tarkemman tarkastelun. Martin [120] tosin ehdottaa `#include` ja `import`-lauseiden hyödyntämisestä mittojen laskennassa. Tästä voidaan arvella hänen tarkoittaneen eksplisiittisesti ilmaistuja käyttö- ja perintäkytköksiä komponenttien välillä.

Kytkösten suunta. Tarkasteleeko mitta lähteviä, saapuvia vai molempia kytköksiä? C_a laskee komponenttiin saapuvia kytköksiä ja C_e lähteviä. [116]

Hienojakoisuus. Kriteeri jakautuu kahteen alakohtaan: abstraktiotasoon ja laskentatapaan. Martinin metriikoille abstraktiotaso on selvästi alaluvussa 4.1 esitelty komponenttitaso. Kytkeytymisen laskentatapana käytetään suurpiirteistä menetelmää,

Taulukko 5.2: Martinin kytkeytymismetriikat kehyksessä.

	Kytkeytymismitat	
Nimi	C_a	C_e
Kytkeytymisehto	Luokka on *relaatiossa* komponentin kanssa. Riippuvuuden muotoa ei ole määritelty.	
Suunta	Tulevat	Lähtevät
Hienojakoisuus		
Abstraktiotaso	Komponentti †	
Laskentatapa	Luokkien määrä, johon tarkasteltava komponentti on kytkeytynyt.	
Palvelimen pysyvyys	Ei huomioida	
Epäsuorat kytkökset	Epäsuoria kytköksiä ei huomioida.	
Perintä		
Rajaus	Kaikki ‡	
Polymorfismi	Ei huomioida ‡	
Perityt piirteet	Ei huomioida ‡	

† Määritelty alaluvussa 4.1.
‡ Martin [116, 118, 119, 120] ei ohjeistanut käytössä.

jossa huomioidaan muiden osien lukumäärät suhteessa tarkasteltavaan komponenttiin.

Palvelimen muuttumattomuus. Tarkastellaanko kytkeytymistä erikseen epästabiileihin ja vakaisiin osiin? C_e ei huomioi käytettyjen palveluiden pysyvyyttä ja pitää kaikkia komponentteja samanarvoisina.

Epäsuorat kytkökset. Huomioiko mitta suorien kytkösten lisäksi epäsuorat? C_e ja C_a tarkastelevat vain luokkien välisiä riippuvuuksia, eivätkä huomioi epäsuoria kytköksiä.

Perintä. Kriteeri jakautuu kolmeen alakohtaan: tarkastelun rajaukseen, polymorfismin

huomioimiseen ja perittyjen piirteiden kohteluun. Martin ei määrittelyissään antanut ali- tai yliluokkille erilaista asemaa, joten myös sukulaisluokat sisällytetään tarkasteltuun. Polymorfismia ei Briandin ym. [33] mukaan ei tarvitse huomioida komponenttitasolla, joten sen tarkastelu voidaan ohittaa.

Yksittäistä luokkaa voidaan tarkastella joko itsenäisenä kokonaisuutena tai sisällyttämällä siihen perityt piirteet. Martin [120] ei ohjeistanut yksittäisen luokan riippuvuuksien kartoittamisessa. Jos laskenta perustuu `#include` ja `import`-lauseisiin, ei luokkien perimiä piirteitä sisällytetä tarkasteluun.

Kytkeytymiskehys osoitti Martinin jättäneen huomioimatta useita ominaisuuden laskentaan vaikuttavia tekijöitä. Hän ei käsitellyt perinnän erityispiirteitä, palvelimen pysyvyyttä tai epäsuoria kytköksiä. Tosin näiden kriteerin kohtelu voidaan päätellä hänen antamistaan ehdotuksista. Suurimpana ongelmana on kuitenkin kytkeytymisehto, sillä riippuvuuden muotoa ei ole määritelty. Tämäkin tosin voidaan ohittaa tulkitsemalla Martinin ehdotusta `#include` ja `import`-lauseiden hyödyntämisestä. Ehdotuksesta voidaan päätellä hänen tarkoittaneen eksplisiittisesti ilmaistuja riippuvuuksia kahden komponentin välillä.

Koheesiokehys

Briandin, Dalyn ja Wüstin [32] esittelivät myös koheesiomitoille kehyksen. Koheesiokehys koostuu viidestä kriteeristä, joita on käsitelty tarkemmin alaluvussa 2.4.3. Seuraavassa tarkastellaan sekä alkuperäistä H:ta että H':a kehyksen kriteereissä. Taulukossa 5.3 on tiivistelmä Martinin koheesiometriikoiden piirteistä kehyksessä.

Koheesion kriteeri. Mikä kytkee komponentin osat toisiinsa? Martinin [120] mukaan H tarkastelee vain *"komponentin sisäisiä relaatioita"*, eikä hän antanut erityistä koheesioehtoa. Martin tosin käytti kytkeytymismitoillaan vastaavaa määritelmää ja suositteli näiden relaatioiden automaattista laskemista `#include` ja `import`-lauseilla. Tästä voidaan olettaa hänen tarkoittaneen myös koheesiomitan relaatioilla

Taulukko 5.3: Martinin alkuperäinen ja uudistettu koheesiomitta kehyksessä.

	Koheesiomitat	
Nimi	H	H'
Koheesion kriteeri	Luokkien välillä on *relaatio*. Kriteeriä ei määritelty tarkemmin.	Eksplisiittisesti nimetyt riippuvuudet, esimerkiksi käyttö ja perintä.
Määrittelyalue	Komponentti †	
Epäsuorat kytkökset	Ei huomioida	
Periytyvät piirteet	Ei huomioida	
Erikoismetodien kohtelu	Ei huomioida	

† Määritelty alaluvussa 4.1.

luokkien välisiä suoria riippuvuuksia. Tällaisia ovat esimerkiksi käyttösuhde- ja perintärelaatiot. H' määriteltiin myös vastaavien relaatioiden varaan.

Määrittelyalue. Millä abstraktiotasolla koheesiota tarkastellaan? Selvästi H:n ja H':n tarkasteltutasona on alaluvussa 4.1 määritelty komponenttitaso.

Epäsuorat kytkennät. Huomioidaanko laskennassa osien epäsuorat vai vain suorat kytkökset? Martin [120] ei puuttunut epäsuorien kytkösten käsittelemiseen, mutta puhtaasti `#include` ja `import` tai vastaaviin lauseisiin perustuvassa laskennassa epäsuoria kytköksiä ei huomioida. H' määriteltiin laskemaan vain suoria kytköksiä.

Periytyvät piirteet. Huomioidaanko laskennassa perintä? Arkkitehtuuritasolla perimän tarkastelu ei ole mielekästä, sillä perintä on luokkatason työkalu.

Erikoismetodien kohtelu. Miten laskennassa huomioidaan erikoismetodit? Luokkajoukon koheesion tarkastelussa erikoismetodien huomioiminen ei ole mielekästä.

Kehys osoitti lähinnä puutteita Martinin koheesiokriteerin määrittelyissä. Tässä opinnäytteessä käsitettiin Martinin *relaatio* luokkien väliseksi eksplisiittiseksi kutsuksi. Tul-

Taulukko 5.4: Martinin metriikoiden yleiset ominaisuudet.

	Määrittely	**Objekt.**	**Asteikko**	**Käytettävissä**	**Vakaa**	**Kieli**	**Validoitu**
C_a	Ei	On	Järjestys	HLD	LLD	-	Ei
C_e	Ei	On	Järjestys	HLD	LLD	-	Ei
I	On	On	Suhde	HLD	LLD	-	Emp. [44]
N_a	On	On	Järjestys	HLD	LLD	-	Ei
N_c	On	On	Järjestys	HLD	LLD	-	Ei
A	On	On	Suhde	HLD	LLD	-	Ei
D	On	On	Suhde	HLD	LLD	-	Ei
D'	On	On	Suhde	HLD	LLD	-	Ei
H	Ei	On	Intervalli	HLD	LLD	-	Ei
H'	On	On	Suhde	HLD	LLD	-	Ei

kinta samaistaa käyttö- ja perintäsuhteet. Tosin luokkien välinen kytkeytyminen komponenttitasolla on kiinnostavaa vain määrällisesti, ei kytkösten laadun suhteen.

Toinen koheesiokehyksestä tehtävä huomio on epäsuorien kytkösten unohtaminen. Epäsuorat kytkökset tarkoittavat komponenteille sitä, että luokka c_1 käyttää c_2:ta luokan c_3 kautta. Tällainen tilanne saattaa esiintyä komponenteissa useastikin. Esimerkiksi luokka voisi käyttää toista rajapinnan kautta. Tämän vuoksi epäsuorat kytkennät olisi hyvä huomioida jollain tavalla koheesiota laskettaessa. Yksinkertaisinta on tarkistaa, muodostaako komponentti yhtenäisen graafin. Tällöin kaikki komponentin osat ovat edes epäsuorasti riippuvaisia toisistaan.

Yleiset ominaisuudet

Briand, Daly ja Wüst [33] käyttivät yleisiä ominaisuuksia luomaan nopean kuvan tarkasteltavasta metriikasta. Yleiset ominaisuudet koostuvat kahdeksasta kohdasta, joita voidaan tarkastella riippumatta siitä, mitä ominaisuutta ohjelmistometriikka laskee. Kohdat käsittelevät esimerkiksi mitan objektiivisuutta, käyttövaiheita kehitysprosessissa, validointia ja mittaushypoteesia. Seuraavassa on tarkasteltu Martinin mittojen ja H':n yleisiä ominaisuuksia. Tulokset on esitetty taulukossa 5.4.

Toimintakuntoinen määrittely. Onko mitta alun perin määritelty niin, että sitä voi suoraan soveltaa ohjelmiston arvioimiseen? Martinin [120] metriikkapaketin C_a:n ja C_e:n määrittelyjen ongelmia on tarkasteltu alakohdissa 4.2.1 ja 5.1.2. H:n ongelmia on käsitelty alakohdissa 4.2.1 ja 5.1.1. Muiden metriikoiden kuvaukset soveltuvat suoraan käytettäviksi.

Objektiivisuus. Onko mitan arvo objektiivinen vai subjektiivinen? Kaikki tarkasteltavat mitat ovat objektiivisia, mutta esimerkiksi alaluvussa 2.1.1 esitetty FPA on subjektiivinen.

Mitta-asteikko. Käyttääkö metriikka laatuero-, järjestys-, intervalli- vai suhdeasteikkoa? I, A ja D ovat selvästi suhdeasteikolla, sillä näille on määritelty yksikäsitteiset ylä- ja alarajat. Koko- ja kytkeytymismitat ovat järjestysasteikoilla. Niillä ei ole kiinteitä ylärajoja, mutta komponentit voidaan järjestää ominaisuuden mukaan mittojen avulla. H:lle ei ole määritelty yksikäsitteistä alarajaa, mutta komponentit voidaan mitan arvon perusteella järjestää oikean etäisyyden päähän toisistaan jatkuvalla suoralle. H on siis intervalliasteikolla. H':lle on taas määritelty yksikäsitteiset ylä- ja alarajat, joten se on suhdeasteikoilla.

Käytettävissä. Missä vesiputousmallin vaiheessa mitta on ensimmäisen kerran käytettävissä? Neljä vaihetta ovat analyysi (engl. *Analysis*), korkean tason suunnittelu (engl. *High-level design*, HLD), matalan tason suunnittelu (engl. *Low-level design*, LLD) ja implementointi (engl. *Implementation*). Analyysivaiheessa määritellään komponenttijako ja alustavat suhteet. Korkean tason suunnittelun aikana komponentit jaetaan luokiksi. LLD:n aikana voidaan vielä tunnistaa ja soveltaa uusia luokkia tai kirjastoja. Implementaatiovaiheen jälkeen lähdekoodi on käytettävissä. [33]

Analyysivaiheessa määritellään jo komponenttien välisiä suhteita, jolloin olisi mahdollista laskea ensimmäisiä kytkeytymismittojen arvoja. C_a ja C_e arvioivat kuitenkin luokkien määrää, joten niiden arvoja voidaan laskea vasta korkean tason suun-

nittelussa. Samoin vasta tällöin voidaan arvioida N_c:tä, N_a:ta, H:ta ja H':a. Johdetut mitat ovat käytössä vasta kun kaikkia niiden osamittoja voidaan arvioida, eli Martinin I:tä, A:ta, D:tä voidaan arvioida vasta HLD:n aikana.

Vakaa. Missä vesiputousmallin vaiheessa mitan arvo on vakaa? Luokkien määrä varmentuu vasta matalan tason suunnittelussa, jolloin N_c ja N_a vakautuvat. Samoin C_a, C_e, H, H' ja näistä johdetut mitat stabiloituvat vasta kun luokkien määrät ovat selvillä.

Kielikohtaisuus. Onko metriikka suunniteltu erityisesti yksittäiselle ohjelmointikielelle? Tarkasteltavat metriikat rakentuvat yleisten olio-ohjelmoinnin käsitteiden varaan.

Kelpuuttaminen. Onko mittaa kelpuutettu teoreettisesti tai empiirisesti? Martinin metriikoita ei ole kokonaisuudessaan kelpuutettu aiemmin. Ainoastaan Champaign [44, 45] on tutkinut Martinin epästabiiliutta empiirisesti. Champaignen tuloksia käsitellään alaluvussa 5.2.1. Tässä opinnäytteessä osoitettiin Martinin metriikkapaketin ilman H:ta olevan teoreettisesti kelvollinen. Uudistettu koheesiomitta H' todettiin myös teoreettisessa tarkastelussa kelvolliseksi.

Mittaushypoteesi. Hypoteesi selvittää mitan ja ulkoisten ominaisuuksien välistä oletettua suhdetta. Martin [120] ei määritellyt mittojen tarkoitusta, mutta metriikkapaketti luotiin SDP ja SAP -periaatteiden noudattamisen seuraamiseksi. Tästä voidaan päätellä, että Martin metriikalla pyritään mittaamaan ohjenuorien korostamia laatuominaisuuksia: *uudelleenkäytettävyyttä*, *ylläpidettävyyttä* ja *siirrettävyyttä*.

SDP ja SAP tekevät komponenteista abstrakteja ja stabiileja, jolloin niillä on paljon vastuuta, mutta vain vähän syitä muuttua. Tällaiset komponentit soveltuvat vakautensa puolesta hyvin uudelleenkäytettäväksi. [120] Abstraktit komponentit ovat myös siirrettävissä, sillä alustariippuvainen toteutus on konkreettisissa osissa. Jos komponentit SDP:n mukaisesti riippuvat vain itseään vakaammista, niiden ylläpidettävyys paranee. Tällöin abstrakteihin komponentteihin tehdyt laajennukset eivät aiheuta muutoksia näistä riippuvissa komponenteissa.

Yleiset ominaisuudet osoittavat jo huomioituja ongelmia C_a:n, C_e:n ja H:n määrittelyissä. H:sta erityisesti korostuu mitta-asteikon ongelma, sillä suhteellisille mitoille voidaan käyttää tehokkaita tilastollisia menetelmiä. Suurin ongelma, jonka yleiset ominaisuudet osoittavat, on teoreettisen ja empiirisen kelpuuttamisen puuttuminen. Ainoastaan yhtä mittaa esitellyistä on tarkasteltu empiirisesti. Tässä opinnäytteessä ongelma pyrittiin korjaamaan tarkastelemalla Martinin mittojen teoreettista taustaa. Metriikkapaketti ilman H:ta todettiin teoreettisesti kelvollisiksi. Koheesiomitasta esiteltiin kriteerit täyttävä H'.

Yleisistä ominaisuuksista huomataan myös mittojen myöhäinen käyttövaihe. Komponenttien väliset suhteet saadaan jo analyysin aikana, mutta ensimmäiset mitat ovat laskettavissa vasta seuraavassa vaiheessa. Kuitenkin *vakaan riippuvuuden* periaatteen mukaista suunnittelua voitaisiin arvioida jo analyysivaiheessa. Tämä vaatisi I:n arvioimista aikaisemmin, mikä voisi onnistua, jos analyysivaiheessa C_a ja C_e laskisivat luokkien sijasta komponentteja. Ongelmana tosin voisi olla mitan arvon raju muuttuminen vaihdettaessa komponenteista takaisin luokkiin. Esimerkiksi jokin vakaa komponentti muuttuisi epästabiiliksi, mikä voisi taas rikkoa SDP-periaatteen mukaisen suunnittelun. Tässä opinnäytteessä ei aihetta käsitellä tarkemmin tilarajoituksen vuoksi.

5.2 Empiirinen testaaminen

Empiirisen kelpuuttamisessa pyritään todistamaan mittaushypoteesi [33] tai tunnistamaan huonoja suunnitteluratkaisuja [115, 136]. Tarkoituksena on osoittaa mitan käyttökelpoisuus. Käytännön testaamista tarvitaan, sillä teoreettisesti hyväksytyt mitat ovat osoittautuneet riippuvaisiksi ohjelmakoosta [60] tai suoriutuneet yksinkertaisempia mittoja huonommin [150]. Empiirinen testaaminen on myös ainoa keino osoittaa mitan hyödyllisyys todellisessa käytössä [32], ja siirtää kehitetyt teoriat hyödynnettäväksi ohjelmistoteollisuudessa.

Yllättävästi Martinin mitta on sisällytetty useisiin mittaohjelmiin ja käytetty monis-

sa tutkimuksissa, mutta metriikkapakettia ei silti ole kattavasti testattu tai kelpuutettu. Teoreettinen tausta sekä intuitiivinen käsitys tukevat Martinin mitan tarpeellisuutta, mutta mitan todellinen hyödyllisyys on vielä osoittamatta. Mitan empiirinen kelpuuttaminen on kuitenkin hyvin vaikeaa, sillä mittaushypoteesin todistaminen vaatisi järjestelmän uudelleenkäytettävyyden, ylläpidettävyyden ja siirrettävyyden arvioimista. Valitettavasti laatuominaisuuksien arvioiminen ei ole yksinkertaista. Esimerkiksi ylläpidon vaatimaa työmäärää on vaikea mitata pelkästään LOC:llä, sillä muutaman rivin korjaus on saattanut viedä useita tunteja.

Uudelleenkäytettävyyttä ja siirrettävyyttä voisi tietysti arvioida komponentin uudelleenkäyttökerroilla. Tällaisen tiedon kerääminen vaatii tarkasti dokumentoitua ympäristöä, jossa on suuri joukko komponentteja ja jokaisen käyttökohteet on vielä kirjattu ylös. Ehtoihin soveltuvan ohjelman löytäminen on hyvin hankalaa. Uudelleenkäytettävyyttä voisi yrittää arvioida myös empiirisellä kokeella, jossa ohjelmistokehityksen ammattilaiset antavat arvosanan komponentin kierrätettävyydelle. Tällaisen kokeen järjestäminen on kuitenkin huomattavan työlästä, sillä riittävään tilastolliseen analyysiin tarvitaan useita arvioijia ja huomattava määrä komponentteja. Ongelmana olisi myös arvion subjektiivisuus.

Seuraavassa on esitelty empiirinen mittaus, jonka tarkoituksena ei ole todistaa Martinin metriikan mittaushypoteesia. Mittauksessa tarkastellaan, kuinka hyvin hyväksi oletetut ohjelmat noudattavat mitan taustalla olevia suunnitteluperiaatteita. Jos hyväksi oletettavat ohjelmat eivät noudata Martinin ideoita, voidaan mitta hylätä tarpeettomana. Vastaavasti jos ohjelmat noudattavat mitan arvoja, metriikoiden taustalla olevat periaatteet vaikuttavat arvioitavissa ohjelmissa. Tällöin voidaan olettaa mitan hyvien arvojen liittyvän hyvän ohjelman ominaisuuksiin. Tämä ei kuitenkaan todistaisi mittaushypoteesia tai mitan taustalla olevien periaatteiden vaikutusta ulkoiseen laatuun.

5.2.1 Metriikan arviointeja kirjallisuudessa

Martinin metriikasta on raportoitu muutamia käyttötapauksia. Esimerkiksi Itkonen [86] arvioi Martinin metriikalla kuuden komponentin kytkeytymistä. Gorton ja Zhu [71] valitsivat mittojen avulla refaktoroitavia komponentteja. Binkley ja Schah [30] sovelsivat epästabiiliutta luokkatasolla, mutta arvioivat huonon menestyksen syyksi yksityiskohtien liian suurta määrää. Ahmad ja Laplante [12] vertasivat normalisoidun etäisyyden arvoa asiantuntijoiden mielipiteisiin. Heidän tutkimuksessaan D oli kolmen tärkeimmän kompleksisuutta arvioivan mitan joukossa.

Redin ym. [147] ja Oliveira ym. [139] käyttivät Martinin metriikoita osana sulautettujen järjestelmien arviointia. Mitattujen pakettien normalisoidut etäisyydet olivat huomattavan korkeita kaikissa komponenteissa. Yksi selitys on se, että abstraktius oli suurimmalla osalla komponenteista 0. Oliveira ja kollegat kuitenkin huomasivat D':n korreloivan fyysisten mittareiden — kuten muistinkäytön tai energiankulutuksen — kanssa. Tosin löydetty riippuvuus ei ollut tilastollisesti yhtä merkitsevä kuin muut havainnot.

Capiluppi ja Boldyreff [40, 41] löysivät epästabiiliuden ja komponenttien uudelleenkäytön väliltä korrelaation. He tosin käyttivät C_e:stä ja C_a:sta painotettuja versioita, jossa kertoimina olivat komponenttien välisten kutsujen määrä. He arvioivat neljän avoimen lähdekoodin ohjelman komponenttien uudelleenkäytettävyyttä. Lupaaviksi todettuja komponentteja he etsivät muista avoimen lähdekoodin projekteista. Neljästätoista komponentista kuutta ei ollut uudelleenkäytetty, tai vastaavuutta ei löydetty hakukoneilla.

Champaign [44] sekä Champaign, Malton ja Dong [45] ovat yrittäneet kelpuuttaa Martinin epästabiiliuden mittaa. He vertasivat epästabiiliuden mittaa Eclipsen ja Linuxin ytimen muutoshistorioihin. Muutosherkkyyttä he arvioivat suhteuttamalla muuttuneiden julkaisujen määrän kaikkiin julkaisuihin, joissa komponentti oli mukana. Komponentti luokiteltiin muuttuneeksi, jos sen koko oli muuttunut edellisestä julkaisuversiosta. Valitulle tekniikalle sekä suuri refaktorointi että yksittäisen kirjoitusvirheen oikaiseminen ovat samanarvoisia muutoksia. Champaign ei suorittanut tilastollista analyysiä riippuvuuksis-

ta, mutta esitettyjen kuvaajien perusteella I:n ja muutosherkkyyden välillä ei ollut yhteyttä.

Martin [119] määrittelee stabiiliuden muutoksen tekemisen vaikeudeksi. Hän painottaa, ettei komponentin vakaus liity muutosten määriin tai toistuvuuteen. Selvästi Champaign [44] ei mitannut muutoksen vaativuutta tai työmäärää, jos yksittäinen muutos on mikä tahansa komponentin koon muutos. Tutkimuksessa arvioitiin komponentteja Eclipsen kuudessa julkaisussa, mutta stabiiliuden arvo laskettiin vain toiseksi viimeisestä julkaisusta. Lähestymistapa on erikoinen, sillä komponentti on voinut muuttua rajusti ensimmäisten julkaisujen aikana ja viimein stabilisoitua. Tutkimuksessa tällainen näkyisi epävakaana komponenttina, joka stabiiliusmitan mielestä on vakaa. Intuitiivisempi lähestymistapa olisi ollut laskea stabiilius ensimmäisestä versiosta.

Champaignin, Maltonin ja Dongin [45] mittauskohde oli Linuxin ydin. Martin [119] määritteli metriikkansa olioparadigman mukaisille komponenteille, mitä C:llä kirjoitetun Linuxin kansiojärjestelmä ei selvästi ole. Champaign ym. [45] tosin puolustivat oliometriikan soveltuvan proseduraaliselle kielelle, sillä Martinin stabiiliusmitan määrittely ei vaadi olioparadigman ominaisuuksia.

Wermelinger ja kollegat [168, 169] tutkivat suunnitteluperiaatteiden noudattamista sekä Martinin epästabiiliuden ja Van Bellen mittojen arvoja Eclipsen kehityksen aikana. He arvioivat komponentin muuttuneeksi, jos komponentin riippuvuuksissa oli tapahtunut muutos. Wermelinger ym. [169] löysivät vähäistä riippuvuutta Martinin stabiiliuden ja Van Bellen muutostiheyden väliltä. Tutkimuksen havaittiin myös Eclipsen noudattavan SDP:n ja ADP:n mukaisia periaatteita. Wermelinger ym. [169] tosin huomauttivat stabiiliuden mittaavan vain järjestelmän sisäisiä muutospaineita, ja mitattuihin muutoksiin vaikuttivat myös ulkoiset syyt.

Martinin mittaa ovat käyttäneet useat tutkijat, mutta ainoastaan Champaign [44, 45] on yrittänyt kelpuuttaa osaa metriikkapaketista. Wermelinger ym. [169] kumosivat Champaignen mittauksen tulokset. Kumpikaan tutkimus ei tosin etsinyt yhteyttä epästabiiliuden

ja ulkoisten laatuominaisuuksien väliltä. Lisäksi useimpia Martinin paketin mittoja ei ole testattua lainkaan empiirisesti. Tästä huolimatta mitat ovat lisätty useisiin metriikkaohjelmiin.

5.2.2 Mittausjärjestelyt ja työkalut

Eric Raymond esittämän kuuluisan *Linuksen lain* mukaan *"tarpeeksi suurella määrällä tarkastelijoita, kaikki virheet häviävät"*. [153, s. 30] Hän vertasi periaatteella avoimen lähdekoodin kehitystä perinteisiin menetelmiin. Suljetussa kehityksessä pieni joukko ammattilaisia testaa ja poistaa virheitä ohjelmasta kuukausia. Raymondin [153] mukaan avoimen lähdekoodin kehityksessä suuri määrä avustavia kehittäjiä tulee etsimään ja tuhoamaan virheitä aina uusista julkaisuista, minkä vuoksi mahdolliset viat poistuvat nopeasti ja tehokkaasti. Hän uskoo tällaisen avoimen lähdekoodin kehityksen tekevän ohjelmasta laadukkaan.

Vuze (v. 4.2.0.0) [165], *Apache Tomcat* (v. 6.0.18) [158], *ArgouML* (v. 0.28) [17], *Apache Xerces2 J* (v. 2.9.1) [173] ja *jEdit* (v. 4.2) [90] ovat hyvin erilaisia, tunnettuja ja käytettyjä vapaita Java-ohjelmia. Avoimen lähdekoodin projekteja luokittelevia ja listaava verkkopalvelu *ohloh* [138] luonnehtii kaikkia viittä valittua ohjelmaa koodikannaltaan kypsiksi ja vakaiksi. *Vuze*:n, *jEdit*:in, *Tomcat*:in ja *ArgoUML*:n vahvuuksina pidetään vielä suurta ja aktiivista kehittäjätyöryhmää. Tässä opinnäytteessä oletetaan näiden viiden edustavan laadukasta sekä hyvien tapojen mukaista ohjelmistosuunnittelua, ja arvioidaan Martinin metriikkaa näiden ohjelmien mittaustulosten perusteella.

Martinin [120] metriikka pyrkii mittaamaan *stabiilin riippuvuuden* ja *vakaan abstraktiuden* periaatteita. Jos kypsän ja hyvin suunnitellun ohjelman taustalla oletetaan periaatteiden vaikuttavan, pitäisi ohjelmien IA-kuvaajien pisteiden painottua pääsarjan lähelle. Toisaalta jos riippuvuus suoran ja pisteiden välillä ei ole merkittävä, saattaa tulos kertoa mitan tai ohjelman huonosta laadusta. Kuitenkin jos kaikkien viiden hyvänä pidetyn ohjelman kuvaajat eivät noudata ihanteellista tapausta, voidaan kyseenalaistaa Martinin

Taulukko 5.5: Mitattavien ohjelmien koko.

	Mitattujen pakettien määrä	kLOC†
Vuze 4.2.0.0	208	450
Apache Tomcat 6.0.18	90	349
ArgoUML 0.28	80	226
Apache Xerces2 J 2.9.1	36	136
jEdit 4.2	16	195
Kaikki	430	1356

† Laskettu koko ohjelmasta. Ilman kommentteja tai tyhjiä rivejä.

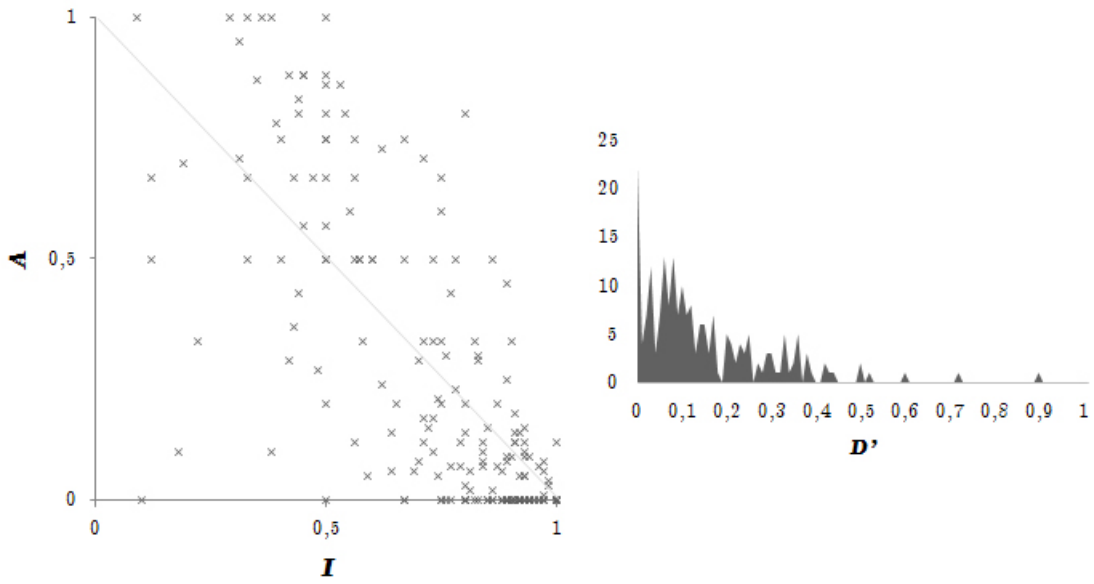
metriikan tarkoituksenmukaisuus.

Mittaukset suoritettiin Eclipsen ohjelmointiympäristössä, sillä tulosten lisäksi haluttiin myös tarkastella ohjelmakoodia ja sen ongelmakohtia. Eclipsen käyttämisen etuna on myös laaja kirjasto lisäosia, joista löytyy useita ohjelmistomittoja laskevia liitännäisiä. Esimerkiksi *JHawk* [164], *Metrics 1.3.6* [127], *DSM* [53] ja *JDepend4Eclipse* [88] ovat Eclipsen lisäosia, jotka laskevat Martinin metriikan arvoja.

Mittauksissa käytettäväksi työkaluiksi näistä valittiin *JDepend4Eclipse* (v. 1.2.1), sillä lisäosa perustuu pitkään kehitettyyn *JDepend*-analyysiohjelmaan [89]. Liitännäinen myös tarjosi edellä mainituista joustavimman käyttöliittymän, jolla metriikoita voitiin tarkastella millä tahansa tasolla. Lisäosa mahdollistaa mittaustulosten tallentamisen XML-formaattiin, millä tulokset saatiin siirrettyä tilasto-ohjelmiin tarkasteltaviksi. *JDepend4Eclipse* laskee abstrakteiksi osiksi abstraktit luokat ja rajapinnat. Perintä- ja käyttösuhteet lasketaan kytköksiksi.

5.2.3 Mittaustulokset

Mitattavista ohjelmista *Vuze* ja *jEdit* käyttivät huomattavia määriä muiden osapuolien kirjoittamia komponentteja. Tarkasteltavien osien joukko rajattiin kuitenkin vain projektien itse tuottamiin komponentteihin, sillä nämä edustavat varmasti samanlaisia suunnittelu-



Kuva 5.1: *Vuze*:n `com.aelitis.*`:n IA-kuvaaja ja D':n histogrammi.

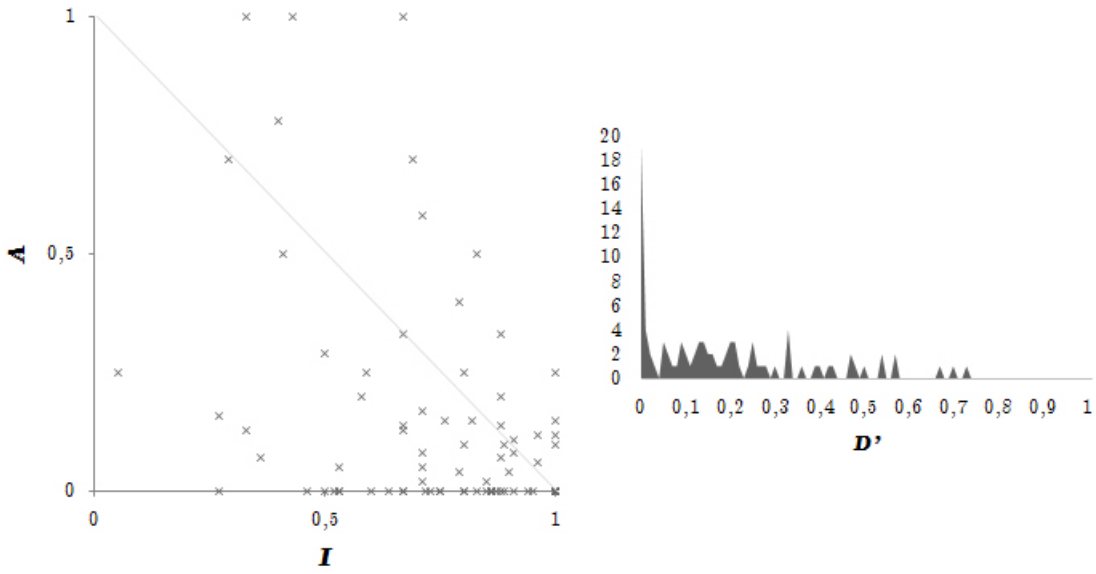
periaatteita ja käytäntöjä. Taulukossa 5.5 on listattu tarkasteltavien komponenttien määrä sekä ohjelmien koko. Mittauksissa yksittäisenä komponenttina pidettiin Javan pakettia.

Kirjastokomponentin stabiiliuden 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.

Tulosten ohessa on tarkasteltu muutamia ohjelmien komponentteja, joiden D':n arvot ovat suuria. Tarkoituksena on pohtia, voiko näiden komponenttien laatua parantaa Martinin [120] refaktorointien mukaisesti.

Vuze

Vuze, aiemmin *Azureus*, on BitTorrent-protokollan asiakasohjelma. [165] Kuvassa 5.1 on esitetty `com.aelitis.*` pakettien sijainti Martinin kuvaajalla ja normalisoidun etäisyyden histogrammi. *Vuze*:n histogrammista on selvästi huomattavissa pakettien painottuminen lähelle nollaa. Tarkasteltavista paketeista 156 (75 %) on alle 0.21 päässä



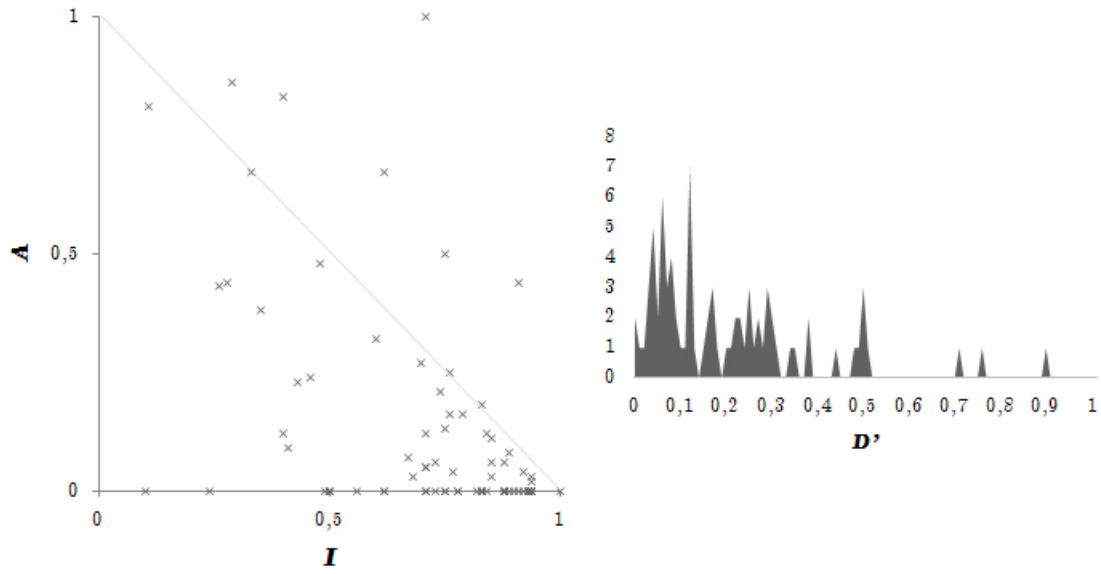
Kuva 5.2: *Tomcat*:in `org.apache.*`:n *IA*-kuvaaja ja D':n histogrammi.

pääsarjasta, ja itse suoralla on 22 pistettä. Vain kuuden komponentin D':n arvo on suurempi kuin 0.5. Selvästi suurin osa pisteistä on pääsarjan lähettyvillä, joten järjestelmä noudattaa joko tarkoituksella tai sattumalta Martinin periaatteiden mukaista suunnittelua.

Apache Tomcat

Apache Tomcat on web-palvelin, joka toteuttaa Sun Microsystemsin Java Servlet ja JSP -määrittelyt. [158] Kuvassa 5.2 on esitetty `org.apache.*` pakettien sijainti Martinin kuvaajalla ja normalisoidun etäisyyden histogrammi. D':n histogrammissa on huomattava piikki kohdassa 0. Noin 20 % järjestelmän komponenteista sijaitsee pääsarjalla; loput pisteet ovat hajautuneet tasaisesti eri arvoille. *Tomcat*:in kuvaaja noudattaa Martinin ihannekuvaa huomattavan tarkasti ja selkeästi ohjelman taustalla vaikuttavat hänen suunnitteluperiaatteensa.

Suurimman D':n arvon ohjelmassa saa `org.apache.tomcat.util.res`-paketti, jota käytetään ohjelman lokalisoinnissa eri kieliversioille. Paketissa on vain `StringManager`-luokka ja tästä luokasta ovat suoraan riippuvaisia kahdeksan eri pakettia. Tällaisenkin komponentin palvelut olisi voinut piilottaa rajapinnan taakse, siitä huolimatta,



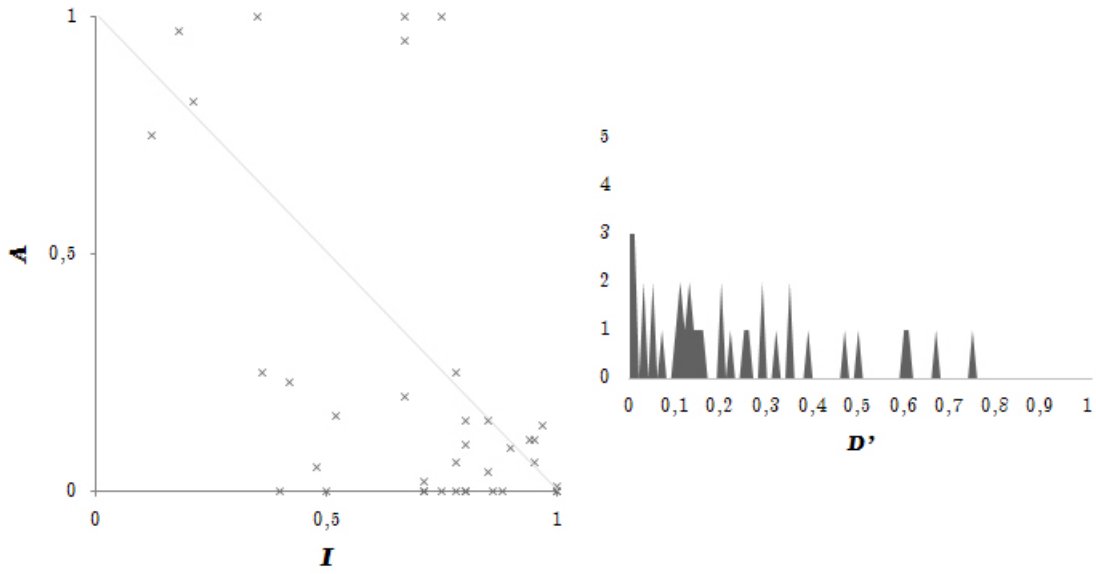
Kuva 5.3: *ArgoUML*:in `org.argouml.*`:n *IA*-kuvaaja ja D':n histogrammi.

että rajapinnalla olisi vain yksi toteuttaja. Nyt yhden luokan muuttaminen vaikuttaa kahdeksaan pakettiin. Kaikki luokan käyttämät palvelut ovat osa Javan luokkakirjastoa. Jos ne luokiteltaisiin stabiileiksi palveluiksi ja jätettäisiin tarkastelun ulkopuolelle, paketin etäisyys pääsarjasta olisi 1.

ArgoUML

ArgoUML on UML:n mallinnustyökalu, joka on käännetty kymmenelle kielelle. [17] Kuvassa 5.3 on esitetty `org.argouml.*` pakettien sijainti Martinin kuvaajalla ja normalisoidun etäisyyden histogrammi. *ArgoUML*:lä on tarkasteltavista ohjelmista ainoa, jossa histogrammin suurin piikki ei ole nollassa: D':n arvolla 0.12 on seitsemän pakettia. Tästä huolimatta yli 80 % pisteistä on korkeintaan 0.3 etäisyyden päästä pääsarjasta. Suurin osa ohjelman komponenteista noudattaa Martinin suunnitteluperiaatteita siitä huolimatta, että pääsarjalla on vain muutama piste.

Paketin `org.argouml.i18n` D':n arvo on 0.9, joka on ohjelman suurin. Komponentissa on vain `Translator`-luokka, joka vastaa järjestelmän lokalisoinnista. Tästä luokasta ovat riippuvia 495 luokkaa 43 paketista. Vastaavasti kuten *Tomcat*:in lokalisoin-



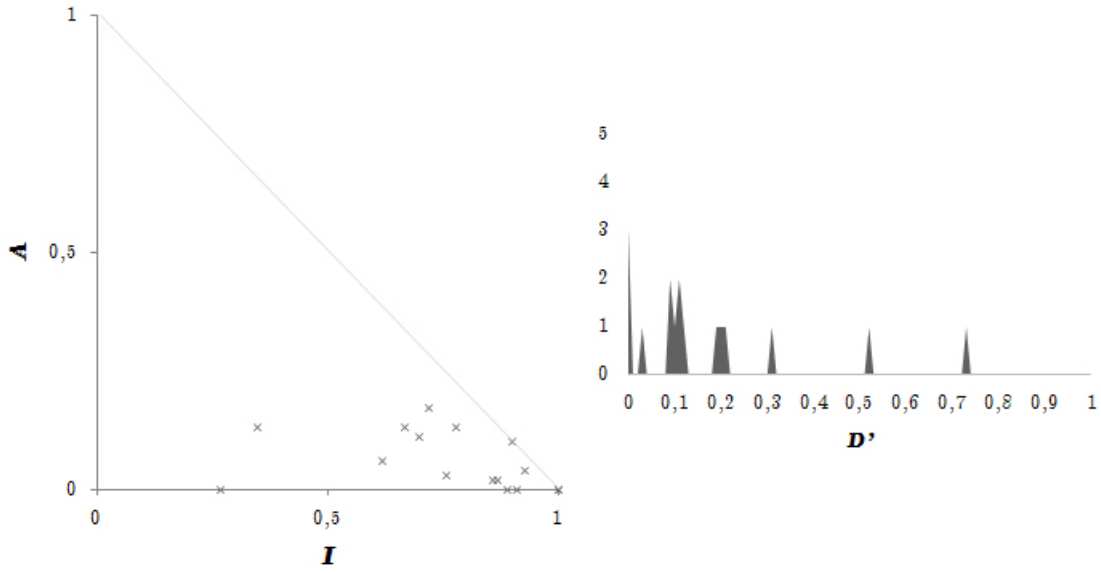
Kuva 5.4: *Xerces*:in `org.apache.*`:n *IA*-kuvaaja ja D':n histogrammi.

nista vastannut paketti, myös `org.argouml.i18n` palvelut pitäisi piilottaa rajapinnan taakse. `Translator`-luokan muutokset vaikuttaisivat puolessa järjestelmän komponenteista.

Paketti `org.argouml.application.helpers`:in etäisyys pääsarjasta on 0.76. Se muodostuu kolmesta konkreettisesta luokasta, jotka tarjoavat järjestelmälle sekalaisia palveluja. Tarkastelevat 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.

Apache Xerces2 J

Apache Xerces2 J on XML-parseri Javalle. [173] Parserista arvioitavat komponentit koostuvat 36 paketista, joten *Xerces* on huomattavasti *Vuze*:a pienempi. Kuvassa 5.4 on esitetty etäisyyksien histogrammi ja komponenttien kuvaaja, jonka perusteella *Xerces*:issä



Kuva 5.5: *jEdit*:in `org.gtj.sp.*`:n *IA*-kuvaaja ja D':n histogrammi.

käytetään suhteessa selvästi enemmän abstrakteja luokkia kuin muissa. Histogrammista nähdään myös D':n hajonnan olevan suurempi kuin muilla mitatuista ohjelmista.

Xerces:in paketeista 28:lla on D':n arvo pienempi kuin 0.32. Yli 0.5 etäisyyden päästä pääsarjasta on viisi komponenttia. Suurin osa järjestelmän paketeista on kuitenkin lähellä pääsarjaa, miksi *Xerces*:kin näyttäisi noudattavan Martinin mitan periaatteita joko suunnitellusti tai tarkoituksettomasti.

Paketti `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.

jEdit

jEdit on ohjelmointiympäristöksi tarkoitettu tekstieditori, joka tukee lisäosien asennuksen jälkeistä kytkemistä. [90] *jEdit* on huomattavasti *Vuze*:a pienempi järjestelmä, ja tarkasteltavia komponentteja on vain 16. Kuvassa 5.5 on piirretty komponenttien kuvaaja ja etäisyyksien histogrammi.

Martinin kuvaajasta on selvästi nähtävissä *jEdit*:in käyttäneen varsin säästeliäästi abs-

traktiota. Tosin suurin osa paketeista on keskittynyt pisteen $(0, 1)$ ja pääsarjan lähettyville. Komponentin `org.gtj.sp`:n alipaketeista kaksi on yli 0.5 päässä pääsarjasta, ja 75 % pisteistä on alle 0.20 päässä.

Paketti `org.gtj.sp.jedit.msg`:n normalisoitu etäisyys on 0.72, ja se koostuu komponenttien keskinäisessä keskustelussa käytettävissä 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* avulla. Vaikka paketti 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 D' 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 pahimmilaan johtaa näistä paketista riippuvien luokkien refakatoroimiseen.

5.3 Teoreettisen ja empiirisen kelpuuttamisen lopputulos

Martinin metriikka on suosittu ja käytetty, mutta mittaa ei ole koskaan kelpuutettu teoreettisesti tai empiirisesti. Luvun alun tarkastelussa huomattiin koko- ja kytkeytymismittojen toteuttavan ominaisuuksien matemaattisen taustan vaatimukset. Huolimatta teoreettisten vaatimusten läpäisemisestä, mitan määrittelyissä olevia puutteita esiteltiin alaluvussa 5.1.2. Keskeisimmät ongelmat liittyivät riippuvuuden määrittelemiseen ja perinnän käsittelyyn. Martin [120] esimerkeistä päätellen huomioidaan kaikki eksplisiittiset ilmaisut toisiin komponentteihin. Hän ei myöskään käsitellyt perittyjä piirteitä, josta voidaan olettaa Martinin jättäneen perinnän kokonaan huomioimatta. Mitan määrittelyissä voidaan

Taulukko 5.6: Normalisoidun etäisyyden tilastollisia tunnuslukuja tarkasteltavista ohjelmista.

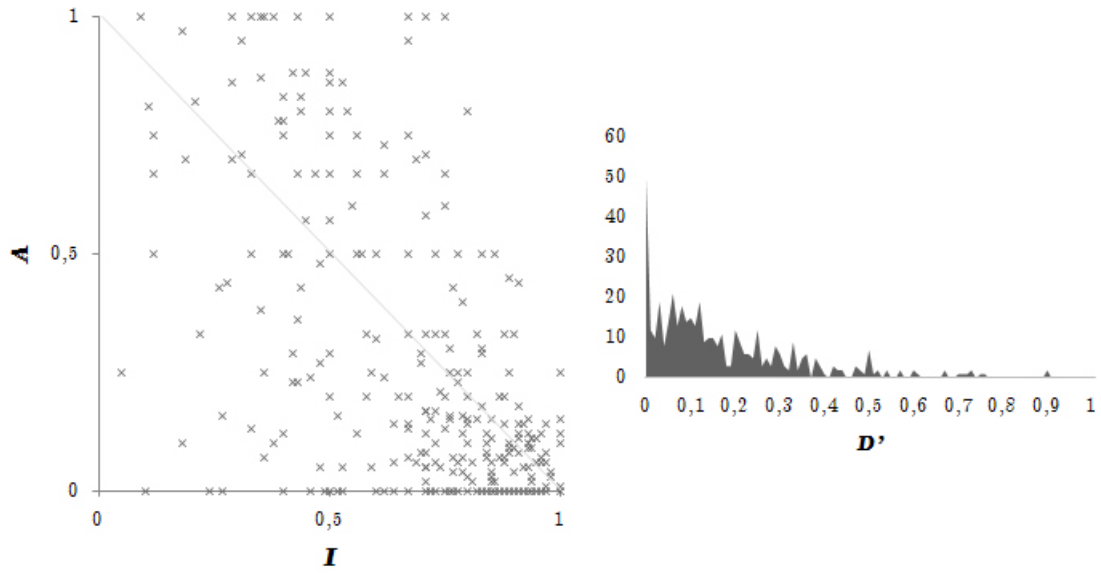
	Keskiarvo	**Mediaani**	**min** (kpl)	**Q1**	**Q3**	**max** (kpl)
Vuze	0.15	0.10	0 (22)	0.05	0.21	0.90 (1)
Tomcat	0.18	0.14	0 (19)	0.01	0.27	0.73 (1)
ArgoUML	0.20	0.16	0 (2)	0.07	0.28	0.90 (1)
Xerces	0.22	0.15	0 (3)	0.05	0.32	0.75 (1)
jEdit	0.18	0.11	0 (3)	0.08	0.20	0.73 (1)
Kaikki	0.17	0.12	0 (49)	0.05	0.25	0.90 (2)

perintää käsitellä vain kytkeytymisen yhtenä muotona.

Opinnäytteessä toteutettiin mittaus viidelle tunnetulle avoimen lähdekoodin ohjelmalle. Toiminnaltaan ja kooltaan hyvinkin erilaiset ohjelmat noudattivat Martinin teoriaa, sillä mitatuista komponenteista suurin osa on hyvin lähellä pääsarjaa. Testin tarkoituksena ei ollut kelpuuttaa mittaa ulkoisten laatuominaisuuksien perusteella, vaan tarkastella miten hyvin hyvänä pidetyt ohjelmat noudattavat metriikan periaatteita. Samalla pystyttiin tunnistamaan muutamien mitan ääriarvoisten komponenttien refaktorointitarpeita.

Taulukossa 5.6 on mitattujen komponenttien tilastollisia tunnuslukuja. Kaikkien mitattujen ohjelmien D':n keskiarvot vaihtelevat välillä 0.15–0.22 ja näiden mediaani on myös lähellä pääsarjaa. Komponenttien arvojen jakaumasta nähdään 75 % pisteistä sijaitsevan korkeintaan 0.25 etäisyyden pääsuorasta. Tilastojen perusteella ohjelmat noudattavat Martinin suunnitteluperiaatteita.

ArgoUML:n ja *Xerces*:in arvot ovat tilastojen ja kuvaajien perusteella huonoimpia. Tosin ohjelmien D':n keskiarvo tai kolmas kvartaali ei eroa merkitsevästi muista. *ArgoUML*:n histogrammi poikkeaa muista ohjelmista, sillä vain 2.5 % pisteistä sijaitsee pääsarjalla. Suurin osa ohjelman komponenteista sijaitsee silti lähellä suoraa, sillä 75 % on korkeintaan 0.28 etäisyyden päässä.



Kuva 5.6: Kaikkien 430 arvioidun komponentin IA-kuvaaja ja D':n histogrammi.

Kuvassa 5.6 on esitetty kaikkien mitattujen komponenttien histogrammi ja IA-kuvaaja. Kuvaajasta nähdään suurimman pisteistä olevan lähellä pääsarjaa ja kaikista mitatuista 430 komponentista 75 % on korkeintaan 0.25 etäisyyden päässä suorasta. Sama on nähtävissä histogrammista, jossa on huomattava piikki arvolla 0. Joka kymmenes tarkastelluista komponenteista sijaitsee pääsarjalla. Histogrammin perusteella suurin osa komponenteista noudattaa Martinin suunnitteluperiaatteita.

Kuvan 5.6 kuvaajasta on havaittavissa huomattavaa painotusta pisteen $(1, 0)$ lähelle. Tämä tarkoittaa abstraktion vähäistä käyttöä tarkastelluissa ohjelmissa. Taulukossa 5.7 on tarkasteltujen ohjelmien abstraktien ja konkreettien luokkien määrät. Kaikkiaan komponenteissa oli 8 105 luokkaa, joista abstrakteja on vain 1 150. Vähiten abstrakteja rakenteita on käytetty *jEdit*:issä, jossa luokista vain 6 %:a on abstrakteja.

Abstraktiuden vähyys voi tarkoittaa, etteivät kehittäjät täysin ymmärrä olioparadigmaa. Erityisesti ohjelmien tarkastelussa esitellyissä komponenteissa konkreetit luokat tarjoavat palveluita suoraan asiakkaille. Näihin luokkien kohdistuvat muutokset vaikuttaisivat laajasti järjestelmässä. Esimerkiksi *ArgoUML*:n `Translator`-luokan muuttaminen vaikuttaa tästä riippuviin 495 luokkaan. Abstraktien luokkien minimimäärälle ei tosin ole

Taulukko 5.7: Abstraktien luokkien osuus ohjelmissa.

	Abstrakteja	**Osuus**	**Konkreetteja**	**Yhteensä**
Vuze	491	15 %	2 699	3 190
Tomcat	164	12 %	1 217	1 381
ArgoUML	278	14 %	1 713	1 991
Xerces	180	20 %	708	888
jEdit	37	6 %	618	655
Kaikki	1 150	14 %	6 955	8 105

olemassa vaatimuksia, eikä ohjelman laatua voida arvioida yleisesti sen perusteella.

Mittausten luotettavuuteen vaikuttaa monta tekijää. Ensinnäkin tutkittujen ohjelmien suunnitteluperiaatteita ei tiedetä. Jos ohjelmien suunnittelun lähtökohtana on ollut noudattaa Martinin periaatteita, mittaukset heijastaisivat vain tämän suunnittelupäätöksen tulosta. Tosin osa tarkastelluista komponenteista rikkoi suunnitteluperiaatteita, joten tarkoituksenmukaisuus on epätodennäköistä.

JDepend4Eclipse liitännäinen laskee C_e:n ja C_a:n riippuvuudet komponenteilla, Martinin [120] määrittelemien luokkien sijasta. Erilainen tulkinta on saattanut vaikuttaa mittauksen lopputulokseen. Tosin suurempi vaikutus tulokseen on stabiilien palveluiden erottamisella mitasta. Jos tällaisia riippuvuuksia ei tarkasteltaisi, mitan arvot voisivat muuttua rajusti. Tarkastelussa havaittiin muutama luokkakirjastoista riippuvainen komponentti, joiden arvot olisivat nousseet stabiilien riippuvuuksien poistamisella. Vastaavasti myös joidenkin komponenttien D':n arvot pienenisivät tällöin.

Mittauksen tulokseen on voinut vaikuttaa myös komponenttien koko. Pienempien komponenttien voidaan olettaa soveltuvan periaatteisiin helpommin ja tämän näkyvän myös tuloksessa. *Vuze*:n komponenteista 152 on alle 0.2:n päässä pääsarjasta. Näistä 67 paketilla on enemmän kuin 10 luokkaa ja kolmella yli sata luokkaa. Isot komponentit

voivat myös saavuttaa matalia arvoja, eikä tarkastelussa huomattu selvää riippuvuutta kokoon.

Mittaustulosten perusteella voidaan arvioida hyvien ohjelmien noudattavan Martinin periaatteita. Empiirisessä mittauksessa ei liitetty mitan arvoja ulkoisiin ominaisuuksiin, mutta pystyttiin osoittamaan refaktorointia tarvitsevia komponentteja. Teoreettisessa tarkastelussa mitat tai niiden korjatut versiot täyttivät asetetut kriteerit. Mittoja ei ole kokonaisuudessaan kelpuutettu, sillä riippuvuus ulkoisiin laatuominaisuuksiin on vielä todistamatta. Martinin metriikkaa voidaan kuitenkin pitää sopivana mittana arvioimaan komponenttien ominaisuuksia, sillä mitalla on mahdollista havaita ongelmallisia komponentteja.

Luku 6

Yhteenveto

Tässä opinnäytteessä on tarkasteltu lukuisia ohjelmointimittoja niin proseduraaliselle kuin olioparadigmalle. Esitetyistä metriikoista on myös osoitettu huomattavia puutteita sekä mittojen määrittelyissä että tarkoituksessa. Kirjallisuuskatsauksessa löydettiin ristiriitaisia tuloksia metriikoiden ja ulkoisten laatuominaisuuksien riippuvuuksissa. Esimerkiksi El Emam ym. [60] kyseenalaistivat suurimman osan empiirisistä tutkimuksista, sillä niissä ei ollut huomioitu ohjelmakoon vaikutusta.

Ohjelmistoarkkitehtuurien dokumentoimista ja kuvaamista esiteltiin lyhyesti. Erityisesti tarkasteltiin keskeisimpiä arkkitehtuurityylejä sekä suunnittelu- ja antisuunnittelumalleja. Samalla pohdittiin mahdollisuuksia arvioida ohjelman laatua näiden automaattisella tunnistamisella. Arkkitehtuuritasolle määriteltyjä mittoja esiteltiin laajasti ja näistä tarkemmin tarkasteltiin Martinin [120], Ducassen ym. [57] ja Ponision [144] metriikoita.

Opinnäytteessä arvioitiin ominaisuuksia, joita arkkitehtuuritasolta voitaisiin mitata. Suurin osa tarkastelluista piirteistä vain heijastuu muista ominaisuuksista, minkä vuoksi on hyödyllisempää keskittyä vain tärkeimpien ominaisuuksien mittaamiseen. Mitattavaksi piirteiksi komponenttitasolle ehdotetaan *abstraktiutta*, *kapselointia*, *koheesiota*, *kokoa*, *kytkeytymistä* ja *stabiiliutta*. Esitellyistä komponenttimetriikoista Martinin [120] mitat kattoivat suurimman osan tarpeellisiksi arvioiduista ominaisuuksista, minkä vuoksi luku 5 keskittyikin Martinin mittojen teoreettiseen ja empiiriseen kelpuuttamiseen.

Lukuun ottamatta Martinin koheesiomittaa, olivat hänen metriikkansa teoreettisesti kelvollisia. Tässä opinnäytteessä määriteltiin uudistettu versio komponenttien koheesiomitasta. H' tukee intuitiivisista koheesiokäsitystä alkuperäistä mittaa paremmin ja esitelty mitta myös toteuttaa Briand ym. [35] koheesioehdot. Tarkasteltaessa kaikkia Martinin mittoja sekä H':a erilaisissa luokittelukehyksissä, havaittiin muutamia puutteita alkuperäisten määrittelyissä. Esimerkiksi Martin ei ollut antanut yksiselitteistä kytkeytymisehtoa mitoilleen.

Empiirisessä arvioinnissa mitattiin viiden avoimen lähdekoodin ohjelman komponenttien D':n arvoja. Laadukkaiksi oletetut ohjelmat noudattivat metriikan suunnitteluperiaatteita, joten Martinin mitta näyttäisi arvioivan järjestelmän hyviä ominaisuuksia. Suoritettu koe ei kuitenkaan liitä mitan arvoja ulkoisiin laatuominaisuuksiin, mitä Briand ym. [32, 33] vaativat mittojen hyväksymiseksi. Tosin Martinin metriikalla pystyttiin osoittamaan muutamia refaktorointia vaativia komponentteja.

Arkkitehtuuritason ohjelmistometriikat pyrkivät arvioimaan järjestelmän tai sen osien laatua. Huolimatta siitä, ettei Martinin metriikan ole osoitettu liittyvän ulkoisiin laatutekijöihin, sillä voidaan havaita huonosti suunniteltuja komponentteja. Teoreettisen taustan ja empiiristen löytöjen perusteella Martinin metriikkaa voidaan siis pitää hyödyllisenä komponenttimittana.

Tämä opinnäyte käsitteli vain pientä osaa arkkitehtuuritason mahdollisuuksista. Metriikoihin liittyviä, vielä aukinaisia kysymyksiä, on useita. Muutamia jatkotutkimuskohteita ovat esimerkiksi:

Metriikoiden vaikutus kehitysprosessissa. Mittojen vaikutusta kehitysprosessiin ja lopullisen tuotteen laatuun ei ole tutkittu. Suurin osa nykyisistä mittauksista on tehty ohjelmistokehityksen päätyttyä, ilman mittareiden ennustaminen arvojen hyödyntämistä. Parantaako aktiivinen reagoiminen mittojen arvoihin ohjelmiston laatua, mikä kuitenkin on laatumetriikoiden tarkoitus?

Metriikoiden käyttäminen. Ainoastaan muutaman mitan käyttämistä ja arvojen tulkit-

semista on ohjeistettu, mutta miten muiden mittojen tuloksia pitäisi arvioida ja miten niihin pitäisi reagoida? Esimerkiksi perintämittojen huonot arvot ovat tietoisten suunnittelupäätösten tuloksia, eikä perintähierarkioita muuteta yksittäisten mittojen vuoksi. Vastaavasti, miten tulisi toimia luokan kanssa, jonka kytkeytymisen arvot ovat suuria?

Metriikoiden käyttämisessä pitäisi huomioida myös niiden eettiset ongelmat ja taloudelliset seuraamukset, joita ei ole kattavasti kartoitettu. Mittoja voidaan käyttää helposti arvioimaan tuotteen sijasta sen toteuttanutta ihmistä. Samoin ohjelmaa saatetaan rakentaa mittojen optimoimiseksi laadun ja toiminnallisuuden kustannuksella, jos käytettävät arviointimenetelmät ovat julkisia ja vaikuttavat esimerkiksi palkkaukseen tai työtehtäviin.

Arkkitehtuuritason metriikat. Opinnäytteessä esitelty lista komponentin ominaisuuksista keskittyi selkeisiin ja yksinkertaisiin piirteisiin. Ominaisuuksien laajemmalla kartoittamisella voitaisiin kehittää uusia mittoja komponentin laadun tai ongelmien arviointiin. Uusien mittojen ohella tulisi myös vanhoja arvioida. Laajaa empiiristä tutkimusta komponenttimetriikoiden ja ulkoisten laatuominaisuuksien välillä ei ole tehty. Suosituinta Martinin metriikkaa on koeteltu ainoastaan muutamassa tutkimuksessa ja näissäkin ainoastaan osittain.

Arkkitehtuurimetriikat ovat tehokas väline komponenttien ja järjestelmien arvioimiseen. Valitettavasti niitä ei ole vielä tutkittu tai arvioitu laajasti, eikä niiden todellista hyödyllisyyttä ole vielä pystytty selvittämään. Ohjelmistokehityksen työvälineiden ja menetelmien abstraktiotason kohotessa on todennäköistä, että tulevaisuudessa arkkitehtuurimetriikoita tutkitaan ja käytetään nykyistä tehokkaammin ohjelmistojen laadun parantamiseen.

Viitteet

[1] Hani Abdeen, Ilham Alloui, Stéphane Ducasse, Damien Pollet ja Mathiu Suen. Package reference fingerprint: a rich and compact visualization to understand package relationships. Kirjassa *CSMR '08: Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, ss. 213–222, huhtikuu 2008.

[2] W. Abdelmoez, D. M. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan, H. H. Ammar, Bo Yu ja A. Mili. Error propagation in software architectures. Kirjassa *METRICS '04: Proceedings of the Software Metrics, 10th International Symposium*, ss. 384–393, Washington, DC, USA, 2004. IEEE Computer Society.

[3] Fernando Brito e Abreu. Design metrics for object-oriented software systems. Kirjassa *ECOOP'95 Workshop "Quantitative Methods for Object-Oriented Systems Development"*, Aarhus, Tanska, elokuu 1995.

[4] Fernando Brito e Abreu. The MOOD2 metrics set (in Portuguese). Technical Report R7/98, INESC, Portugali, huhtikuu 1998.

[5] Fernando Brito e Abreu. Using OCL to formalize object oriented metrics definitions. Technical Report ES007/2001, INESC, Portugali, kesäkuu 2001.

[6] Fernando Brito e Abreu ja Rogério Carapuça. Object-oriented software engineering: Measuring and controlling the development process. Kirjassa *Proceedings of the 4th International Conference on Software Quality*, ss. 1–8, McLean, VA, USA, lokakuu 1994. Revised version.

[7] Fernando Brito e Abreu ja Jean Sebastien Cuche. Collecting and analyzing the MOOD2 metrics. Kirjassa *Proceedings of the ECOOP'98 Workshop - Object-Oriented Product Metrics for Software Quality Assessment*, ss. 258–260, heinäkuu 1998.

[8] Fernando Brito e Abreu, Rita Esteves ja Miguel Goulão. The design of Eiffel programs: Quantitative evaluation using the MOOD metrics. Kirjassa *TOOLS '96: Proceedings of the Technology of Object Oriented Languages and Systems*, heinäkuu 1996.

[9] Fernando Brito e Abreu, Miguel Goulão ja Rita Esteves. Toward the design quality evaluation of object-oriented software systems. Kirjassa *Proceedings of the 5th International Conference on Software Quality*, Austin, TX, USA, lokakuu 1995. Revised version.

[10] Fernando Brito e Abreu ja Walcélio Melo. Evaluating the impact of object-oriented design on software quality. Kirjassa *METRICS '96: Proceedings of the 3rd International Symposium on Software Metrics*, ss. 90–99, Washington, DC, USA, maaliskuu 1996. IEEE Computer Society.

[11] Adam Abubakar, Jarallah AlGhamdi ja Moatz Ahmed. Can cohesion predict fault density? Kirjassa *AICCSA '06: Proceedings of the IEEE International Conference on Computer Systems and Applications*, ss. 890–893, Washington, DC, USA, maaliskuu 2006. IEEE Computer Society.

[12] Norita Ahmad ja Phillip 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*, ss. 119–124, Washington, DC, USA, 2007. IEEE Computer Society.

[13] Aivosto Oy, Helsinki, Finland. *Project Analyzer v8.1 Help*, helmikuu 2007. Viitattu 21.10.2008.

[14] Edward B. Allen, Taghi M. Khoshgoftaar ja Ye Chen. Measuring coupling and cohesion of software modules: An information-theory approach. Kirjassa *METRICS '01: Proceedings of the 7th International Symposium on Software Metrics*, s. 124, Washington, DC, USA, 2001. IEEE Computer Society.

[15] Scott W. Ambler. *The Elements of UML style*. Cambridge University Press, Cambridge, Englanti, 2003.

[16] Juan José Amor, Gregorio Robles ja Jesús M. González-Barahona. Measuring woody: The size of debian 3.0. Reports on Systems and Communications RoSac-2005-10, Univeridad Rey Juan Carlos, Madrid, Espanja, kesäkuu 2005.

[17] ArgoUML. `http://argouml.tigris.org/`. Viitattu 28.5.2009.

[18] Erik Arisholm, Lionel C. Briand ja Audun Føyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, elokuu 2004.

[19] Jack Barnard ja Art Price. Managing code inspection information. *IEEE Software*, 11(2):59–69, maaliskuu 1994.

[20] Victor R. Basili, Lionel C. Briand ja Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, lokakuu 1996.

[21] Victor R. Basili, Gianluigi Caldiera ja H. Dieter Rombach. Goal question metric paradigm. Kirjassa John J. Marciniak, toim., *Encyclopedia of Software Engineering*, ss. 528–532. John Wiley & Sons — Interscience, New York, NY, USA, 1994.

[22] Victor R. Basili ja David H. Hutchens. An empirical study of a syntactic complexity family. *IEEE Transactions on Software Engineering*, 9(6):664–672, marraskuu 1983.

[23] Victor R. Basili ja David M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, 10(6):728–738, marraskuu 1984.

[24] Gareth Baxter, Marcus Frean, James Noble, Mark Rickerbyand Hayden Smith, Matt Visser, Hayden Melton ja Ewan Tempero. Understanding the shape of java software. *ACM SIGPLAN Notices*, 41(10):397–412, 2006.

[25] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert Cecil Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland ja Dave Thomas. Manifesto for agile software development. `http://www.agilemanifesto.org/`, 2001.

[26] Theodore B. Van Belle. *Modularity and the Evolution of Software Evolvability*. Väitöskirja, The University of New Mexico, Albuquerque, New Mexico, USA, joulukuu 2004.

[27] Per Olof Bengtsson. Towards maintainability metrics on software architecture: An adaptation of object-oriented metrics. Kirjassa Jan Bosch, Görel Hedin, Kai Koskimies ja Bent Bruun Kristensen, toim., *NOSA'98: Proceedings of the First Nordic Workshop on Software Architecture*, ss. 87–91, Ronneby, Ruotsi, 1998. University of Karlskrona/Ronneby.

[28] Edward V. Berard. Metrics for object-oriented software engineering. `http://www.ipipan.gda.pl/~marek/objects/TOA/moose.html`, 1995. The Object Agency, Inc., Viitattu 11.4.2009.

[29] James M. Bieman ja Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. Kirjassa *SSR '95: Proceedings of the 1995 Symposium on Software reusability*, ss. 259–262, New York, NY, USA, 1995. ACM.

[30] Aaron B. Binkley ja Stephen R. Schach. A comparison of sixteen quality metrics for object-oriented design. *Information Processing Letters*, 58(6):271–275, kesäkuu 1996.

[31] Jan Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education, New York, NY, USA, 2000.

[32] Lionel C. Briand, John W. Daly ja Jürgen Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[33] Lionel C. Briand, John W. Daly ja Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, tammikuu/helmikuu 1999.

[34] Lionel C. Briand, Prem Devanbu ja Walcélio L. Melo. An investigation into coupling measures for C++. Kirjassa *ICSE '97: Proceedings of the 19th international conference on Software engineering*, ss. 412–421, New York, NY, USA, 1997. ACM.

[35] Lionel C. Briand, Sandro Morasca ja Victor R. Basili. Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, 22(1):68–86, tammikuu 1996.

[36] Lionel C. Briand, Jürgen Wüst, John Daly ja Victor Porter. A comprehensive empirical validation of design measures for object-oriented systems. Kirjassa *METRICS '98: Proceedings of the 5th International Symposium on Software Metrics*, ss. 246–257, Washington, DC, USA, marraskuu 1998. IEEE Computer Society.

[37] Lionel C. Briand, Jürgen Wüst ja Hakim Lounis. Replicated case studies for investigating quality factors in object-oriented designs. *Empirical Software Engineering*, 6(1):11–58, 2001.

[38] William J. Brown, Raphael C. Malveau, III Hays W. McCormick ja Thomas J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*, sarjassa *Object Technology*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[39] Bernd Bruegge ja Allen H. Dutoit. *Object-Oriented Software Engineering: Using UML, Patterns, and Java*. Pearson Prentice Hall, Upper Saddle River, NJ, USA, toinen laitos, 2004.

[40] Andrea Capiluppi ja Cornelia 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*, s. 9, Washington, DC, USA, 2007. IEEE Computer Society.

[41] Andrea Capiluppi ja Cornelia Boldyreff. Identifying and improving reusability based on coupling patterns. Kirjassa *ICSR '08: Proceedings of the 10th international conference on Software Reuse*, ss. 282–293, Berlin, Heidelberg, 2008. Springer-Verlag.

[42] David N. Card ja Robert L. Glass. *Measuring software design quality*. Prentice-Hall, Inc., Englewood Cliffs, NJ, USA, 1990.

[43] Heung Seok Chae, Yong Rae Kwon ja Doo Hwan Bae. A cohesion measure for object-oriented classes. *Software: Practice and Experience*, 30(12):1405–1431, lokakuu 2000.

[44] John C. Champaign. An empirical study of software packaging stability. Pro gradu, University of Waterloo, Waterloo, Ontario, Kanada, 2003.

[45] John C. Champaign, Andrew Malton ja Xinyi Dong. Stability and volatility in the linux kernel. Kirjassa *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, ss. 95–102, Washington, DC, USA, syyskuu 2003. IEEE Computer Society.

[46] Shyam R. Chidamber ja Chris F. Kemerer. Towards a metrics suite for object oriented design. Kirjassa *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, ss. 197–211, New York, NY, USA, 1991. ACM.

[47] Shyam R. Chidamber ja Chris. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, kesäkuu 1994.

[48] Shyam R. Chidamber ja Chris. F. Kemerer. Author's reply. *IEEE Transactions on Software Engineering*, 21(3):265, maaliskuu 1995.

[49] Neville I. Churcher ja Martin J. Shepperd. Comments on 'a metrics suite for object oriented design'. *IEEE Transactions on Software Engineering*, 21(3):263–265, maaliskuu 1995.

[50] Peter Coad ja Edward Yourdon. *Object-Oriented Analysis*, sarjassa *Yourdon Press Computing Series*. Prentice Hall, Englewood Cliffs, NJ, USA, toinen laitos, 1990.

[51] Peter Coad ja Edward Yourdon. *Object-Oriented Design*, sarjassa *Yourdon Press Computing Series*. Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

[52] Samuel D. Conte, Hubert E. Dunsmore ja Vincent Y. Shen. *Software Engineering Metrics and Models*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, USA, 1986.

[53] Dsm. `https://gamma.mini.pw.edu.pl/mediawiki/index.php?title=DSM`. Viitattu 16.4.2009.

[54] Desmond F. D'Souza ja Alan Cameron Wills. *Objects, components, and frameworks with UML: the Catalysis approach*, sarjassa *Addison-Wesley Object Technology Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[55] Stéphane Ducasse, Tudor Girba ja Adrian Kuhn. Distribution map. Kirjassa *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, ss. 203–212, Washington, DC, USA, 2006. IEEE Computer Society.

[56] Stéphane Ducasse, Michele Lanza ja Laura Ponisio. A top-down program comprehension strategy for packages. Technische Berichte IAM-04-007, Software Composition Group, University of Bern, Sveitsi, syyskuu 2004.

[57] Stéphane Ducasse, Michele Lanza ja Laura Ponisio. Butterflies: A visual approach to characterize packages. Kirjassa *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium*, s. 7, Washington, DC, USA, 2005. IEEE Computer Society.

[58] Stéphane Ducasse, Damien Pollet, Mathiu Suen, Hani Abdeen ja Ilham Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. Kirjassa *ICSM '07: In Proceedings of the IEEE Intenational Conference on Software Maintenance*, ss. 94–103, lokakuu 2007.

[59] ISO/IEC 9126-1 First edition 2001-06-15. Software engineering – Product quality – part 1: Quality model. International Standard ISO/IEC 9126-1:2001(E), ISO/IEC, Geneva, Sveitsi, kesäkuu 2001.

[60] Kalhed El Emam, Saïda Benlarbi, Nishith Goel ja Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, heinäkuu 2001.

[61] Letha H. Etzkorn, Sampson E. Gholston, Julie L. Fortune, Cara E. Stein, Dawn Utley, Phillip A. Farrington ja Glenn W. Cox. A comparison of cohesion metrics for object-oriented systems. *Information and Software Technology*, 46(10):677–687, elokuu 2004.

[62] Norman E. Fenton ja Martin Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2):149–157, heinäkuu 1999.

[63] Norman E. Fenton ja Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company, Boston, MA, USA, toinen laitos, 1998.

[64] Martin Fowler, Kent Beck, John Brant ja William Opdyke. *Refactoring: Improving the Design of Existing Code*, sarjassa *Object Technology Series*. Addison-Wesley, Upper Saddle River, NJ, USA, 1999.

[65] Xavier Franch ja Juan Pablo Carvallo. Using quality models in software package selection. *IEEE Software*, 20(1):34–41, tammikuu 2003.

[66] FrontEndART Ltd., Szeged, Hungary. *User's Guide for FrontEndART Monitor Version 1.0 beta*, helmikuu 2008. Viitattu 20.10.2008.

[67] Jayababu G., David Peter S. ja Sumam Mary Idicula. An empirical validation of the cohesion measure based on member connectivity for object-oriented classes. Kirjassa Hamid R. Arabnia ja Hassan Reza, toim., *Proceedings of the International Conference on Software Engineering Research and Practice, SERP '04*, osa 2, ss. 705–711. CSREA Press, kesäkuu 2004.

[68] Erich Gamma, Richard Helm, Ralph Johnson ja John Vlissides. *Olio-ohjelmointi - Suunnittelumallit*, sarjassa *IT Press Professional*. IT Press, Oy Edita Ab, Helsinki, Finland, 2001. Alkuperäisteos "*Design patterns: Elements of reusable object-oriented software 20th p.*", kääntänyt Anita Toivonen.

[69] David Garlan ja Mary Shaw. An introduction to software architecture. CMU Software Engineering Institute Technical Report CMU-CS-94-166, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[70] Jesus M. Gonzalez-Barahona, Gregorio Robles, Martin Michltoukokuur, Juan José Amor, ja Daniel M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, marraskuu 2009.

[71] Ian Gorton ja Liming Zhu. Tool support for just-in-time architecture reconstruction and evaluation: an experience report. Kirjassa *ICSE '05: Proceedings of the 27th International Conference on Software engineering*, ss. 514–523, New York, NY, USA, 2005. ACM.

[72] Miguel Goulão ja Fernando Brito e Abreu. Software components evaluation: an overview. Kirjassa *CAPSI'2004: Proceedings of the 5th Conference of the Portuguese Association of Information Systems*, marraskuu 2004.

[73] Juha Gustafsson ja Lilli Nenone. *MAISA Metrics Tool: User Manual*. University of Helsinki, Helsinki, Suomi, version 1.3 laitos, joulukuu 2001.

[74] Juha Gustafsson, Jukka Paakki, Lilli Nenonen ja A. Inkeri Verkamo. Architecture-centric software evolution by software metrics and design patterns. Kirjassa *CSMR '02: Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, ss. 108–115, Washington, DC, USA, 03 2002. IEEE Computer Society.

[75] Harri Hakonen. Designing object oriented software (design patterns), huhtikuu 2008. Lecture notes of University of Turku, Suomi.

[76] Maurice H. Halstead. *Elements of Software Science*, sarjassa *Operating and programming systems series*. Elsevier Science Inc., New York, NY, USA, 1977.

[77] Hannu Harju ja Mika Koskela. Kustannustehokas ohjelmiston luotettavuuden suunnittelu ja arviointi. osa 2. VTT Tiedotteita - Research Notes 2193, Valtion Teknillinen Tutkimuskeskus: Tuotteet ja tuotanto, Espoo, Suomi, 2003.

[78] Rachel Harrison, Steve J. Counsell ja Reuben V. Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, 24(6):491–496, 1998.

[79] Edwin Hautus. Improving Java software through package structure analysis. Kirjassa *The 6th IASTED International Conference Software Engineering and Applications*. ActaPress, marraskuu 2002.

[80] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*, sarjassa *The object-oriented series*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[81] Martin Hitz ja Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems. Kirjassa *Proceedings of International Symposium on Applied Corporate Computing*, ss. 75–76, 197, 78–84, lokakuu 1995.

[82] Martin Hitz ja Behzad Montazeri. Measuring product attributes of object-oriented systems. Kirjassa *ESEC'95: Proceedings of the 5th European Software Engineering Conference*, sarjan *Lecture Notes in Computer Science* osa Volume 989/1995, ss. 124–136, London, UK, 1995. Springer-Verlag.

[83] Martin Hitz ja Behzad Montazeri. Chidamber and Kemerer's metrics suite: A measurement theory perspective. *IEEE Transactions on Software Engineering*, 22(4):267–271, huhtikuu 1996.

[84] International Function Point Users Group (IFPUG). Organisaation kotisivut. `http://www.ifpug.org/`, marraskuu 2005. Viitattu 17.9.2008.

[85] ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15. *Systems and software engineering - Recommended practice for architectural description of software-intensive systems*. IEEE, heinäkuu 2007.

[86] Jonne Itkonen. Comparing the results of relation analysis and coupling metrics — initial case study. Kirjassa *QAOOSE '05: The 9th European Conference on Object-Oriented Programming Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, heinäkuu 2005.

[87] Panjak Jalote. Use of metrics in high maturity organizations. *Software Quality Professional*, maaliskuu 2002.

[88] JDepend4Eclipse. `http://andrei.gmxhome.de/jdepend4eclipse/`. Viitattu 16.4.2009.

[89] JDepent. `http://clarkware.com/software/JDepend.html`. Viitattu 16.4.2009.

[90] jEdit. `http://www.jedit.org/`. Viitattu 16.4.2009.

[91] Capers Jones. *Software Assessments, Benchmarks, and Best Practices*, sarjassa *Addison-Wesley Information Technology Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[92] Padmaja Joshi ja Rushikesh K. Joshi. Microscopic coupling metrics for refactoring. Kirjassa *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, ss. 145–152, Washington, DC, USA, maaliskuu 2006. IEEE Computer Society.

[93] Ravi Kalakota, Sukumar Rathnam ja Andrew B. Whinston. The role of complexity in object-oriented systems development. Kirjassa *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences*, osa 4, ss. 759–768. IEEE Computer Society Press, tammikuu 1993.

[94] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, toinen laitos, 2002.

[95] Ville Karp. Correlation between software metrics and defect density. Pro gradu, Department of Information Technology, University of Turku, Suomi, kesäkuu 2007.

[96] Hla Myat Kaung, Nan Si Kham ja Ni Lar Thein. To visualize the coupling among modules. Kirjassa *APSITT 2005: Proceedings of the 6th Asia-Pacific Symposium on Information and Telecommunication Technologies*, ss. 111–116, marraskuu 2005.

[97] Chris F. Kemerer. Reliability of function points measurement: a field experiment. *Communications of the ACM*, 36(2):85–97, helmikuu 1993.

[98] Md. Abdul Khaer, M.M.A. Hashem ja Md. Raihan Masud. An empirical analysis of software systems for measurement of design quality level based on design patterns. Kirjassa *ICCIT'07: Proceedings of the 10th International Conference on Computer and Information Technology, 2008.*, ss. 1–6, joulukuu 2007.

[99] Latika Kharb ja Rajender Singh. Complexity metrics for component-oriented software systems. *SIGSOFT Software Engineering Notes*, 33(2):1–3, maaliskuu 2008.

[100] Barbara Kitchenham, Shari Lawrence Pfleeger ja Norman Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, joulukuu 1995.

[101] Mikko Kontio. Architectural manifesto: Designing software architectures, part 5. `http://www.ibm.com/developerworks/wireless/library/wi-arch11/`, helmikuu 2005. IBM developerWorks.

[102] Kai Koskimies ja Tommi Mikkonen. *Ohjelmistoarkkitehtuurit*, sarjassa *Valikko-sarja*. Talentum Media Oy, Jyväskylä, Suomi, 2005.

[103] Philippe Kruchten. Architectural blueprints — the "4+1"view model of software architecture. *IEEE Software*, 12(6):42–50, marraskuu 1995.

[104] Al Lake ja Curtis Cook. Use of factor analysis to develop OOP software complexity metrics. Kirjassa *Proceedings of the 6th Annual Oregon Workshop on Software Metrics*, huhtikuu 1994.

[105] John Lakos. *Large-scale C++ software design*, sarjassa *Addison Wesley Professional Computing Series*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[106] 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.

[107] Michele Lanza ja Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer-Verlag Berlin Heidelberg New York, Berlin, Saksa, 2006.

[108] Wei Li ja Sallie Henry. Maintenance metrics for the object oriented paradigm. Kirjassa *Proceedings of the first International Software Metrics Symposium*, ss. 52–60. IEEE Computer Society Press, toukokuu 1993.

[109] Wei Li ja Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[110] MOT Tietotekniikan liiton ATK-sanakirja 5.0. Viitattu 14.10.2008.

[111] Mikael Lindvall, Roseanne Tesoriero ja Patricia Costa. Avoiding architectural degeneration: An evaluation process for software architecture. Kirjassa *METRICS '02: Proceedings of the 8th International Symposium on Software Metrics*, s. 77, Washington, DC, USA, 2002. IEEE Computer Society.

[112] Mark Lorenz ja Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*, sarjassa *The Object-Oriented Series*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[113] Mark W. Maier, David Emery ja Rich Hilliard. Software Architecture: Introducing IEEE standard 1471. *Computer*, 34(4):107–109, 2001.

[114] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. Kirjassa *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, ss. 350–359, Washington, DC, USA, syyskuu 2004. IEEE Computer Society.

[115] Radu Marinescu. Measurement and quality in object-oriented design. Kirjassa *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, ss. 701–704, Washington, DC, USA, syyskuu 2005. IEEE Computer Society.

[116] Robert Cecil Martin. Object oriented design quality metrics: An analysis of dependencies. `http://www.objectmentor.com/resources/articles/oodmetrc.pdf`, lokakuu 1995.

[117] Robert Cecil Martin. Granularity article. `http://www.objectmentor.com/resources/articles/granularity.pdf`, joulukuu 1996. Alun perin julkaistu *Engineering Notebook* kolumnina *C++ Report*-lehdessä, joulukuu 1996.

[118] Robert Cecil Martin. Stability article. `http://www.objectmentor.com/resources/articles/stability.pdf`, helmikuu 1997. Alun perin julkaistu *Engineering Notebook* kolumnina *C++ Report*-lehdessä, helmikuu 1997.

[119] Robert Cecil Martin. Design Principles and Design Patterns. `http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf`, tammikuu 2000.

[120] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*, sarjassa *Alant Apt Series*. Prentice Hall, Upper Saddle River, NJ, USA, 2002.

[121] Tobias Mayer ja Tracy Hall. A critical analysis of current OO design metrics. *Software Quality Control*, 8(2):97–110, 1999.

[122] Tobias Mayer ja Tracy Hall. Measuring oo systems: A critical analysis of the MOOD metrics. Kirjassa *TOOLS '99: Proceedings of Technology of Object-Oriented Languages and Systems*, ss. 108–117, Washington, DC, USA, heinäkuu 1999. IEEE Computer Society.

[123] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, joulukuu 1976.

[124] Hayden Melton ja Ewan Tempero. A simple metric for package design quality. Research Report Series UoA-SE-2005-7, University of Auckland, Software Engineering, Auckland, Uusi-Seelanti, syyskuu 2005.

[125] Hayden Melton ja Ewan Tempero. The CRSS metric for package design quality. Kirjassa *ACSC '07: Proceedings of the thirtieth Australasian conference on Computer science*, ss. 201–210, Darlinghurst, Australia, tammikuu 2007. Australian Computer Society, Inc.

[126] Tom Mens ja Michele Lanza. A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, 72(2):57–68, 2002. GraBaTs 2002, Graph-Based Tools (First International Conference on Graph Transformation).

[127] Metrics 1.3.6. `http://metrics.sourceforge.net/`. Viitattu 16.4.2009.

[128] Bertnard Meyer. *Object-Oriented Software Construction*, sarjassa *Object-Oriented Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, toinen laitos, 1997.

[129] Tommi Mikkonen. *Programming Mobile Devices: An Introduction for Practitioners*. John Wiley & Sons, Rivers Street, NJ, USA, 2007.

[130] Subhas C. Misra ja Virendra C. Bhavsar. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. Kirjassa Gerhard Goos, Juris Hartmanis ja Jan van Leeuwen, toim., *Computational Science and Its Applications – ICCSA 2003*, sarjan *Lecture Notes in Computer Science* osa 2667/2003, ss. 724–732. Springer Berlin, Heidelberg, Saksa, 2003.

[131] Vojislav B. Mišić. Cohesion is structural, coherence is functional: Different views, different measures. Kirjassa *METRICS '01: Proceedings of the 7th International Symposium on Software Metrics*, ss. 135–144, Washington, DC, USA, 04 2001. IEEE Computer Society.

[132] Sami Mäkelä ja Ville Leppänen. Observations on Lack of Cohesion Metrics. Kirjassa *CompSysTech '06: Proceedings of the 2006 international conference on Computer systems and technologies*, ss. 1–6, 2006.

[133] Sami Mäkelä ja Ville Leppänen. Client based object-oriented cohesion metrics. Kirjassa *COMPSAC '07: Proceedings of the 31st Annual International Computer*

Software and Applications Conference - Vol. 2- (COMPSAC 2007), ss. 743–748, Washington, DC, USA, 2007. IEEE Computer Society.

[134] Sami Mäkelä ja Ville Leppänen. External views on class cohesion. Kirjassa *CompSysTech '07: Proceedings of the 2007 international conference on Computer systems and technologies*, ss. 1–6, New York, NY, USA, 2007. ACM.

[135] Sami Mäkelä ja Ville Leppänen. Experimental evaluation of cohesion metrics. Julkaisematon, 2009.

[136] Sami Mäkelä ja Ville Leppänen. Client-based cohesion metrics for Java programs. *Science of Computer Programming*, 74(5–6):355–378, maaliskuu 2009.

[137] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti ja Giancarlo Succi. A case study on the impact of refactoring on quality and productivity in an agile team. Kirjassa *Balancing Agility and Formalism in Software Engineering*, osa 5082/2008, ss. 252–266. Springer-Verlag Berlin, Heidelberg, Saksa 2008.

[138] Ohloh, the open source network. `http://www.ohloh.net/`. Viitattu 16.4.2009.

[139] Marcio F. S. Oliveira, Ricardo Miotto Redin, Luigi Carro, Luís da Cunha Lamb ja Flávio Rech 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*, ss. 68–77, Washington, DC, USA, 2008. IEEE Computer Society.

[140] Jukka Paakki, Anssi Karhinen, Juha Gustafsson, Lilli Nenonen ja A. Inkeri Verkamo. Software metrics by architectural pattern mining. Kirjassa *ICS '00: Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*, ss. 325–332, elokuu 2000.

[141] Mark C. Paulk. The practices of high maturity organizations. Kirjassa *SEPG Conference*, ss. 8–11, 1999.

[142] Laura Ponisio ja Oscar Nierstrasz. Using context information to re-architect a system. Kirjassa *Proceedings of the 3rd Software Measurement European Forum 2006 (SMEF'06)*, ss. 91–103, Rome, Italia, toukokuu 2006.

[143] Laura Ponisio ja Oscar Nierstrasz. Using contextual information to assess package cohesion. Technische Berichte IAM-06-002, Software Composition Group, University of Bern, Sveitsi, toukokuu 2006.

[144] Maria Laura Ponisio. *Exploiting Client Usage to Manage Program Modularity*. Väitöskirja, University of Bern, Sveitsi, kesäkuu 2006.

[145] Jeffery S. Poulin. Measurement and metrics for software components. Kirjassa George T. Heineman ja William T. Councill, toim., *Component-based software engineering: putting the pieces together*, ss. 435–452. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[146] Sandeep Purao ja Vijay Vaishnavi. Product metrics for object-oriented systems. *ACM Computing Surveys*, 35(2):191–221, 2003.

[147] Ricardo M. Redin, Marcio F.S. Oliviera, Lisane B. Brisolara, Julio C.B. Mattos, Luis C. Lamb, Flávio R. Wagner ja Luigi Carro. On the use of software quality metrics to improve physical properties of embedded systems. Kirjassa Bernd Kleinjohann, Lisa Kleinjohann ja Wayne Wolf, toim., *Distributed Embedded Systems: Design, Middleware and Resources*, sarjan *IFIP International Federation for Information Processing* osa 271/2008, ss. 101–110. Springer Boston, 2008.

[148] Marco Ronchetti, Giancarlo Succi, Witold Pedrycz ja Barbara Russo. Early estimation of software size in object-oriented environments a case study in a CMM level 3 software firm. *Information Sciences*, 176(5):475–489, maaliskuu 2006.

[149] Santonu Sarkar, Avinash C. Kak ja Girish Maskeri Rama. Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE Transaction On Software Engineering*, 34(5):700–720, syyskuu/lokakuu 2008.

[150] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, maaliskuu 1988.

[151] Jouni Smed ja Harri Hakonen. *Algorithms and Networking for Computer Games*. John Wiley & Sons Ltd., Southern Gate, Chichester, UK, huhtikuu 2006.

[152] Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA. How do you define software architecture? `http://www.sei.cmu.edu/architecture/definitions.html`, 2009. Viitattu 25.1.2009.

[153] Eric S.Raymond. *The Cathedral and the Bazaar: Musings on Linux and open source by an accidental revolutionary*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, revised laitos, 2001.

[154] Charles R. Symons. Function point analysis: difficulties and improvements. *IEEE Transactions on Software Engineering*, 14(1):2–11, tammikuu 1988.

[155] Charles R. Symons. *Software sizing and estimating: Mk II FPA (Function Point Analysis)*. John Wiley & Sons, Inc., New York, NY, USA, 1991.

[156] David P. Tegarden, Steven D. Sheetz ja David E. Monarchi. A software complexity model of object-oriented systems. *Decision Support System*, 13(3-4):241–262, maaliskuu 1995.

[157] Testwell Oy, Tampere, Suomi. *Measurement of Halstead Metrics with Testwell CMT++ and CMTJava*, kesäkuu 2007. Viitattu 15.10.2008.

[158] Apache Tomcat. `http://tomcat.apache.org/`. Viitattu 28.5.2009.

[159] Jos J. Trienekens, Rob J. Kusters, Michiel J. Van Genuchten ja Hans Aerts. Targets, drivers and metrics in software process improvement: Results of a survey in a multinational organization. *Software Quality Control*, 15(2):135–153, 2007.

[160] Joel Troster. Assessing design-quality metrics on legacy software. Kirjassa *CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, ss. 113–131. IBM Press, 1992.

[161] Edmond VanDoren. Function point analysis. `http://www.sei.cmu.edu/str/descriptions/fpa_body.html`, Jan 1997. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, Viitattu 18.9.2008.

[162] Edmond VanDoren. Cyclomatic complexity. `http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html`, heinäkuu 2000. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, Viitattu 19.9.2008.

[163] Bill Venners. Composition versus inheritance: Which one should you choose? *JavaWorld*, lokakuu 1998.

[164] Virtual Machinery, Dublin, Ireland. *JHawk Java Code Metrics - Metrics Guide*. Viitattu 27.10.2008.

[165] Vuze. `http://azureus.sourceforge.net/`. Viitattu 16.4.2009.

[166] W3C Recommendation REC-DOM-Level-1-19981001. *Document Object Model (DOM) Level 1 Specification, Version 1.0*. The World Wide Web Consortium (W3C), lokakuu 1998.

[167] Hironori Washizaki, Hirokazu Yamamoto ja Yoshiaki Fukazawa. A metrics suite for measuring reusability of software components. Kirjassa *METRICS '03: Proceedings of the 9th International Symposium on Software Metrics*, s. 211, Washington, DC, USA, syyskuu 2003. IEEE Computer Society.

[168] Michel Wermelinger ja Yijun Yu. Analyzing the evolution of Eclipse plugins. Kirjassa *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, ss. 133–136, New York, NY, USA, 2008. ACM.

[169] Michel Wermelinger, Yijun Yu ja Angela Lozano. Design principles in architectural evolution: A case study. Kirjassa *ICSM '08: IEEE International Conference on Software Maintenance*, ss. 396–405, Washington, DC, USA, syyskuu 2008. IEEE Computer Society.

[170] Elaine J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, syyskuu 1988.

[171] Michael Wilhelm ja Stephen Diehl. Dependency viewer - a tool for visualizing package design quality metrics. Kirjassa *VISSOFT '05: Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, ss. 125–126, Washington, DC, USA, 2005. IEEE Computer Society.

[172] Gyun Woo, Heung Seok Chae, Jian Feng Cui ja Jeong-Hoon Ji. Revising cohesion measures by considering the impact of write interactions between class members. *Information and Software Technology*, 51(2):405–417, 2009.

[173] Apache Xerces2 Java Parser. `http://xerces.apache.org/xerces2-j/`. Viitattu 16.4.2009.

[174] Sherif M. Yacoub, Hany H. Ammar ja Tom Robinson. A methodology for architectural-level risk assessment using dynamic metrics. Kirjassa *ISSRE '00: Proceedings of the 11th International Symposium on Software Reliability Engineering*, ss. 210–221, Washington, DC, USA, 2000. IEEE Computer Society.

[175] Tianlin Zhou, Baowen Xu, Liang Shi, Yuming Zhou ja Lin Chen. Measuring package cohesion based on context. Kirjassa *WSCS '08: Proceedings of the IEEE*

International Workshop on Semantic Computing and Systems, ss. 127–132, Washington, DC, USA, heinäkuu 2008. IEEE Computer Society.