Tampereen yliopisto

Antti Eskelinen

OHJELMAKOODIN KOMPLEKSISUUDEN MITTAAMINEN METODITASOLLA

Informaatioteknologian ja viestinnän tiedekunta
Kandidaatintyö
Huhtikuu 2019

TIIVISTELMÄ

Antti Eskelinen: Ohjelmakoodin kompleksisuuden mittaaminen metoditasolla
Kandidaatintyö
Tampereen yliopisto
Tietotekniikka
Huhtikuu 2019

Ohjelmakoodin kompleksisuus tekee koodista vaikeammin ylläpidettävää. Heikko ylläpidettävyys näkyy suurempina kustannuksina. Kompleksisuuden seuraaminen voi auttaa ohjaamaan kaikkia ohjelmistoprojektien vaiheita määrittelystä ylläpitämiseen. Tässä työssä tutkitaan kolmea laajasti tunnettua kompleksisuusmittaria, joita voidaan käyttää metoditason kompleksisuuden seuraamiseen. Tutkitut mittarit ovat syklomaattinen kompleksisuus, Halsteadin ohjelmistometriikat ja ylläpidettävyysindeksi.

Työssä tutkittiin mittareiden käyttäytymistä analysoimalla ohjelmallisesti 6448 metodia Java-kielisestä avoimen lähdekoodin projektista. Tutkimus koostui kahdesta osasta. Ensimmäisessä vaiheessa mittareiden antamista tuloksista laskettiin korrelaatiomatriisi ja piirrettiin hajontakaaviot. Toisessa vaiheessa tutkittiin mittareiden käyttäytymistä tekemällä havaintoja kompleksisimmista metodeista. Lisäksi vertailtiin keskenään rivimääriltään yhtä pitkiä, mutta kompleksisuudeltaan erilaisia metodeja.

Tutkimuksessa selvisi, että kaikkien mittarien antamat tulokset olivat melko vahvasti riippuvaisia metodien rivimääristä. Suurin korrelaatio, joka oli arvoltaan 0,942, löytyi syklomaattisen kompleksisuuden ja ohjelmakoodin rivimäärän väliltä.

Mittareiden käyttäytymisestä löydettiin eroja. Syklomaattiselta kompleksisuudeltaan kompleksisemmissa metodeissa käytettiin paljon switch-lauseita sekä lyhyitä ehtolauseita ja silmukkarakenteita. Halsteadin työmäärän ja vaikeuden mukaan kompleksisemmat metodit sisälsivät keskimäärin pidempiä rivejä ja enemmän erilaisia ohjelmointikielen rakenteita. Ylläpidettävyysindeksin mukaan vaikeammin ylläpidettävissä metodeissa havaittiin sekä paljon silmukkarakenteita, ehtoja switch-lauseita että pitkiä rivejä ja useiden erilaisten ohjelmointikielen rakenteiden käyttöä.

Avainsanat: Kompleksisuus, Ylläpidettävyys, Staattinen analyysi

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

D	Halsteadin vaikeus
E	Halsteadin työmäärä
MI	Ylläpidettävyysindeksi
v	Syklomaattinen kompleksisuus

1 JOHDANTO

Ohjelmistokehittäjän työssä valmiin koodin lukeminen ja ymmärtäminen vie suuren osan työpäivästä. Ohjelmakoodia ei kirjoiteta pelkästään tulkille tai kääntäjälle, joka muuttaa sen tietokoneen ymmärtämään muotoon, vaan myös toisille koodin kanssa työskenteleville ihmisille. Tämä tuo lisävaatimuksia hyvälle koodille: hyvä ohjelmakoodi ei ole pelkästään esimerkiksi virheetöntä tai tehokasta, vaan se on myös helposti ymmärrettävää. Yksi merkittävä tekijä, joka haittaa koodin luettavuutta, on sen kompleksisuus.

Koodin luettavuuden ja kompleksisuuden kvantitatiivinen analysoiminen on haastavaa niiden subjektiivisen luonteen vuoksi. Kuitenkin varsinkin kompleksisuuden mittaamiseen on kehitetty lukuisia menetelmiä, joita kutsutaan myös kompleksisuusmittareiksi (complexity metrics).

Tässä työssä tutkitaan kolmea laajasti tunnettua kompleksisuuden mittaamiseen käytettyä menetelmää: syklomaattista kompleksisuutta, Halsteadin mittareita ja ylläpidettävyysindeksiä. Mittareiden yhteiseksi vertailukohdaksi valittiin koodirivien määrä, joka on yksi perinteisimmistä ja käytetyimmistä ohjelmistometriikoista. Vaikka kaikki näistä mittareista on luotu ennen olio-ohjelmoinnin yleistymistä, käytetään niitä yhä kompleksisuuden mittaamiseen metoditasolla.

Työssä perehdytään valittujen mittareiden teoreettisiin taustoihin, jotta niiden antamia tuloksia voitaisiin ymmärtää paremmin. Valittujen mittareiden käyttäytymistä tutkitaan myös käytännössä. Tavoitteena on löytää mittareiden ominaispiirteitä ja eroavaisuuksia mittareiden toiminnasta. Työssä pyritään myös selvittämään, antavatko valitut kompleksisuusmittarit lisäarvoa ohjelmistokehittäjälle verrattuna koodirivien määrään.

Työn alkupuolella kappaleessa 2 esitellään yleisesti ohjelmakoodin kompleksisuutta ja siihen liittyviä ominaisuuksia. Lisäksi siinä perehdytään kompleksisuuden mittaamisen merkitykseen ohjelmistojen tuotannossa. Taustoituksen jälkeen kappaleessa 3 syvennytään työhön valittujen mittareiden taustalla olevaan teoriaan. Varsinaista työhön liittyvää tutkimusta ja tutkimustuloksia kuvaillaan kappaleessa 4. Lopuksi työssä tehdyt tärkeimmät havainnot esitetään yhteenvetona kappaleessa 5.

2 KOMPLEKSISUUS OSANA OHJELMISTON LAATUA

2.1 Kompleksisuuden osa-alueet

Ohjelmointikielten perusrakenteiden yksikäsitteinen nimeäminen on tärkeää, jotta myös kompleksisuutta voidaan käsitellä yksikäsitteisesti. Tässä työssä valintarakenteilla tarkoitetaan yleisesti sekä ohjelmointikielten if- että switch-lauseita. Ehtolauseilla viitataan puolestaan ainoastaan if-lauseisiin. While- ja for-rakenteita kutsutaan silmukkarakenteiksi.

Ohjelmakoodin kompleksisuudelle on annettu monia määritelmiä. Institute of Electrical and Electronics Engineers -järjestön sanaston määritelmän mukaan ohjelmakoodi on kompleksista, kun se sisältää paljon komponentteja tai komponenttien välisiä suhteita [11]. Ajamin et al. artikkelissa koodin kompleksisuuteen viitataan ominaisuutena, joka tekee siitä vaikealukuista [1]. Ohjelmiston kompleksisuutta on tutkittu laajalti jo vuodesta 1976 [25].

Koodin kompleksisuudella on monta eri tekijää. Antinyan et al. määrittelevät artikkelissaan kompleksisuuden lähteeksi koodin osan elementtien ja näiden välisten suhteiden suuren määrän, esitystavan epäselkeyden ja koodin osaan tehdyt muutokset ohjelman kehittämisen aikana. Elementeillä tarkoitetaan esimerkiksi muuttujia ja funktiokutsuja. Elementtien välisillä suhteilla tarkoitetaan puolestaan matemaattisia operaattoreita, ehtolauseita, koodin sisäkkäisyyttä ja niin edelleen. Koodiin tehdyt muutokset lisäävät kompleksisuutta, koska tällöin ohjelmoijan aikaisempi tieto koodin toiminnallisuudesta ei ole enää ajan tasalla. [2]

Eri koodirakenteet ja niiden keskinäiset suhteet lisäävät kompleksisuutta eri tavoin. Esimerkiksi silmukkarakenteen ymmärtämisen on huomattu vievän yli kaksinkertaisen ajan verrattuna ehtolauseiden ymmärtämiseen. Silmukkarakenteet johtavat myös useammin virheisiin. [1] Rakenteiden sisäkkäisyyden ajatellaan kasvattavan kompleksisuutta [2].

Koodin säännönmukaisuus ja yleistyneiden koodauskonventioiden noudattaminen vaikuttavat tutkimusten perusteella vähentävän koodin kompleksisuutta. Silmukat, joiden silmukkamuuttujan arvoa kasvatetaan, ovat helpompia ymmärtää kuin silmukat, joissa silmukkamuuttujan arvoa vähennetään. Silmukassa silmukkamuuttujan alustamisen arvolla yksi arvon nolla sijaan, <= -operaattorin käyttäminen lopetusehdossa < -operaattorin sijaan ja silmukkamuuttujan vertaaminen lopetusehdossa laskutoimitukseen $n - 1$ pelkän

muuttujan n sijaan johtaa keskimäärin noin 20 % useammin virheisiin. [1] Suunnittelumallien käytöllä on tutkittu olevan kompleksisuutta vähentävä vaikutus [20]. Myös muut koodin rakenteen toisteisuudet, symmetriat ja samankaltaisuudet vähentävät ohjelmoijien kokemaa kompleksisuutta [2][12].

Olio-ohjelmoinnin yleistyminen on laajentanut kompleksisuuden käsitettä entisestään. Monia ennen olio-ohjelmoinnin yleistymistä kehitettyjä mittareita voidaan käyttää edelleen metoditasolla, mutta korkeamman abstraktiotason mittaamiseen ne eivät sovellu. Olio-ohjelmia varten onkin kehitetty omia mittareita. Olio-ohjelmista voidaan mitata esimerkiksi peritymiseen liittyviä ominaisuuksia kuten periytymishierarkian syvyyttä, aliluokkien määrää ja jälkeläisluokkien määrää. Lisäksi voidaan mitata esimerkiksi luokkien välisiä suhteita. [9]

2.2 Kompleksisuuden mittaaminen osana koodin laadun seuraamista

Koodin kompleksisuus on ohjelmistotuotannossa erityisen tärkeä mittari. Kompleksisuuden mittaaminen voi auttaa ohjaamaan vaatimusmäärittelyä, ohjelmiston suunnittelua, ohjelmointia, testaamista ja ylläpitoa. Ohjelmiston laadun mittareiden kasvava käyttö on synnyttänyt yhä uusia mittareita koodin kompleksisuudelle. Myös kasvavat vaatimukset ohjelmiston laadulle ovat lisänneet tarvetta laadun mittaamiselle. [25]

Ohjelmistojen laatutekijät voidaan jakaa kahteen osaan: ulkoisiin ja sisäisiin. Ulkoisia laatutekijöitä ovat esimerkiksi ohjelmiston oikeellisuus, käytettävyys ja tehokkuus. Loppukäyttäjä kokee ohjelmasta ainoastaan ulkoiset laatutekijät. Sisäiset laatutekijät ovat sellaisia, joiden kanssa vain ohjelmistokehittäjä joutuu suoraan tekemisiin. Ohjelmiston sisäisiä laatutekijöitä ovat ylläpidettävyys, joustavuus, siirrettävyys, uudelleenkäytettävyys, luettavuus, testattavuus ja ymmärrettävyys. [17, s. 463–465] Ohjelmiston sisäisillä laatutekijöillä on vahva yhteys ohjelmakoodin laatuun, koska ohjelmisto koostuu pohjimmiltaan ohjelmakoodista.

Kompleksisuus heikentää koodin ylläpidettävyyttä, mikä johtaa ylläpitokustannuksien kasvamiseen. Bankerin et al. tutkimuksen mukaan hyvin kompleksinen koodi voi aiheuttaa noin 25 % ohjelmistoprojektin ylläpitokustannuksista ja vähintään 17 % ohjelmiston elinkaarikustannuksista [3].

Kompleksisuutta mittaamalla on mahdollista arvioida koodimuutoksien vaikutusta ylläpidettävyyteen. Kompleksisuus osoittaa työmäärän, joka tarvitaan muutoksen tekemiseksi koodiin [25]. Kompleksisuuden mittaamisen avulla voidaan myös löytää ylläpidettävyyden kannalta ongelmallisimmat kohdat. Ongelmakohtien löytäminen on tärkeää, sillä niillä on tapana muuttua pahemmiksi ajan myötä. Mitä vaikeammin ylläpidettävää koodi on, sitä työläämpää on muutosten tekeminen koodiin ilman sen ylläpidettävyyden heikentämistä edelleen. Pahimmillaan ongelmat kumuloituvat ja johtavat siihen, että ohjelmistoa ei enää

kannata ylläpitää. [14]

Koska kompleksista koodia on vaikea ymmärtää, on myös sen oikeellisen toiminnan varmistaminen haastavaa. Näin ollen kompleksisen koodin voisi ajatella olevan altis myös ohjelmointivirheille. Tutkimusten valossa kompleksisuuden ja virhealttiuden välinen suhde ei kuitenkaan vaikuta aivan näin suoraviivaiselta. Muutamissa ohjelmistoprojekteissa on havaittu virheiden määrän vähentyneen, kun ongelmakohtia on tunnistettu ja korjattu kompleksisuusmittareiden avulla [17, s. 457]. Eräässä tutkimuksessa kuitenkin huomattiin vastoin odotuksia, etteivät kompleksisuusmittareiden antamat lukemat korreloineet virhetiheyden kanssa [27].

3 KOMPLEKSISUUDEN MITTAREITA

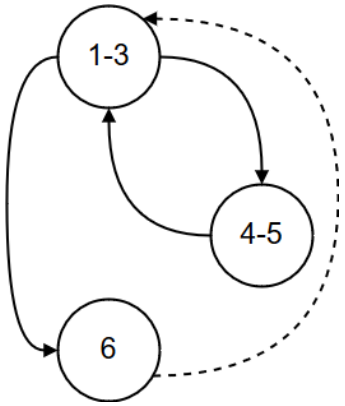
3.1 Syklomaattinen kompleksisuus

Yksi tunnetuimmista ja käytetyimmistä kompleksisuuden mittareista on McCaben syklomaattinen kompleksisuus [6][25]. Se esiteltiin jo vuonna 1976 [16]. Vaikka syklomaattinen kompleksisuus suunniteltiin alun perin proseduraaliseen ohjelmointiin, on sen suosio säilynyt myös olio-ohjelmoinnin yleistymisen jälkeen [6].

Syklomaattisen kompleksisuuden taustalla oleva matematiikka pohjautuu verkkoteoriaan [16]. Verkkoteoriassa *graafi* tai *verkko* koostuu rajallisesta määrästä solmuja (node) ja kaaria (edge). Kaaret yhdistävät solmuja toisiinsa, ja ne voivat olla suunnattuja tai suuntaamattomia. Suunnattua kaarta pitkin voi liikkua vain yhteen suuntaan solmusta toiseen, kun taas suuntaamatonta kaarta voi liikkua molempiin suuntiin. [22] Graafi on yhtenäinen, jos jokaisesta solmusta päästään kaaria ja solmuja pitkin jokaiseen graafin solmuun [16].

Verkkoteoriaa sovelletaan ohjelmistoon siten, että ohjelmakoodin ajatellaan muodostavan suunnattu graafi, kuten kuvassa 3.1. Graafissa on yksi sisääntulo- ja poistumissolmu. Suunnattujen kaarien ajatellaan kuvaavan valittuja suoritushaaroja ja solmut puolestaan eri haaroissa aina peräkkäisessä järjestyksessä suoritettavaa koodia. Koodissa kutsuttavia funktioita sanotaan komponenteiksi. Jos poistumissolmu liitetään suunnatulla kaarella sisääntulosolmuun, tulee graafista vahvasti yhtenäinen. [16] Kuvassa 3.1 poistumissolmun ja sisääntulosolmun välistä takaisinkytkentää on merkitty katkoviivalla.

```
1  def kertoma(a):
2      tulos = 1
3      while a != 0:
4          tulos = a * tulos
5          a = a-1
6      return tulos
```



Kuva 3.1. *Ohjelmakoodi ja sitä vastaava graafi. Graafiin solmuihin on lisätty kutakin solmua vastaavat ohjelmakoodin rivit.*

Vahvasti yhtenäisessä graafissa syklomaattinen luku kertoo ylärajan graafin lineaarisesti riippumattomien silmukoiden lukumäärälle. Graafin lineaarisesti riippumattomien silmukoiden määrä vastaa pienintä määrää erilaisia polkuja, joiden kombinaationa voidaan esittää kaikki mahdolliset silmukat graafissa. Näin ollen graafin lineaarisesti riippumattomat silmukat vastaavat ohjelman suorituspolkuja, joiden kombinaationa voidaan esittää kaikki mahdolliset suorituspolut ohjelmakoodissa. Vahvasti yhtenäisen graafin G syklomaattinen luku $v(G)$ saadaan laskettua kaavalla

$$v(G) = e - n + p \tag{3.1}$$

jossa e on kaarien lukumäärä, n solmujen lukumäärä ja p yhdistettyjen komponenttien lukumäärä graafissa. [16]

Syklomaattinen kompleksisuus kertoo ylärajan lineaarisesti riippumattomien silmukoiden määrän sijaan lineaarisesti riippumattomien polkujen määrälle. Syklomaattinen kompleksisuus v saadaan laskettua kaavalla

$$v = e - n + 2p \tag{3.2}$$

Syklomaattista kompleksisuutta laskettaessa ohjelmakoodia kuvaavan graafin poistumissolmun ja sisääntulosolmun välillä ei ajatella olevan suunnattua kaarta. [16]

Koska syklomaattisen kompleksisuuden laskeminen kaavalla (3.2) voi olla liian työlästä käytännön ohjelmistokehityksen näkökulmasta, on syklomaattiselle kompleksisuudelle esitelty yksinkertaisempia laskutapoja. Näissä laskutavoissa otetaan huomioon vain yksi komponentti, joten funktioiden sisäinen kompleksisuus jää pois. [16]

Millsin esittelemässä mallissa ohjelmakoodia kuvaavan graafin solmut voidaan jakaa ehtosolmuihin (predicate node), funktiosolmuihin (function node) ja keräyssolmuihin (collection node). Siinä ohjelmakoodia esittävässä graafissa on yhden sisääntulo- ja poistumissolmun sijasta pelkästään yksi sisääntulo- ja poistumiskaari. [19]

Ehtosolmut kuvaavat ohjelmakoodin ehtoja, joita esiintyy valinta- ja silmukkarakenteissa. Monihaaraiset valintarakenteet esitetään useamman ehtosolmun avulla siten, että kustakin ehtosolmusta lähtee kaksi haaraa. Ehtosolmuun tulee aina yksi kaari ja siitä lähtee kaksi kaarta. Funktiosolmut vastaavat aina peräkkäisessä järjestyksessä suoritettavia osia ohjelmassa, joten niillä on yksi tulo- ja lähtökaari. Keräyssolmut vastaavat ohjelmakoodin kohtia, joista kaksi eri suorituspolkua kohtaavat. Tällainen kohta on esimerkiksi ehtolauseen jälkeen. Keräyssolmuista suoritus jatkuu vain yhtä kaarta pitkin eteenpäin, joten keräyssolmuun tulee kaksi kaarta ja siitä lähtee yksi kaari. [19]

On osoitettu, että ohjelmakoodia kuvaavan graafin kaarien lukumäärä saadaan laskettua

kaavalla

$$e = 1 + \theta + 3\pi \tag{3.3}$$

kun θ on funktiosolmujen lukumäärä ja π ehtoja kuvaavien solmujen lukumäärä. [19] Koska graafilla McCaben mukaan on olemassa vain yksi sisääntulo- ja poistumissolmu, on jokaista ehtosolmua kohden oltava vähintään yksi keräyssolmu. Näin ollen solmujen määrä voidaan laskea kaavalla

$$n = \theta + 2\pi + 2 \tag{3.4}$$

Jos tutkitaan ohjelmakoodia yhtenä komponenttina, eli pätee $p = 1$, saadaan sijoittamalla kaavaan (3.2) graafin syklomaattiseksi kompleksisuudeksi v tällöin

$$v = (1 + \theta + 3\pi) - (\theta + 2\pi + 2) + 2 = \pi + 1 \tag{3.5}$$

Toisin sanoen siis ohjelmakoodin syklomaattinen kompleksisuus voidaan laskea koodista laskemalla pelkästään ehtojen lukumäärä. [16]

On huomattava, että esimerkiksi konjunktion sisältämä ehtolause ”if (a and b)” voidaan kirjoittaa myös kahtena sisäkkäisenä ehtolauseena, joten niiden kohdalla kompleksisuutta pitää kasvattaa kahdella. [16] Kompleksisuuden laskemisessa voi noudattaa seuraavaa tekniikkaa: aloitetaan laskeminen arvosta yksi, lisätään kompleksisuuteen yksi aina jokaisen avainsanan ”if”, ”while”, ”for”, ”and” ja ”or” kohdalla ja lisätään yksi myös jokaisen case-ehdon kohdalla [17, s. 458].

Ohjelman syklomaattinen kompleksisuus on aina vähintään yksi. McCabe ehdottaa suuntaa antavaksi syklomaattisen kompleksisuuden ylärajaksi arvon 10. Tämän rajan ylittävä koodi on liian kompleksista. [16]

Monet ohjelmalliset laatutyökalut mittaavat syklomaattista kompleksisuutta [6]. Esimerkiksi staattisen analyysin työkalu SonarQube laskee syklomaattisen kompleksisuuden [18].

Syklomaattinen kompleksisuus ei mittaa kaikkia kompleksisuuden osa-alueita. Syklomaattista kompleksisuutta laskettaessa ei esimerkiksi tehdä eroa silmukoiden ja ehtolauseen välillä, sillä molempien kohdalla syklomaattinen kompleksisuus kasvaa vain yhdellä. Sisäkkäisten valinta- ja silmukkarakenteiden aiheuttama syklomaattinen kompleksisuus on sama kuin peräkkäisten valinta- ja silmukkarakenteiden.

Syklomaattinen kompleksisuutta onkin kritisoitu liian karkeaksi mittariksi. Syklomaattinen kompleksisuus ei ota huomioon lausekkeiden ja muuttujien määrää. [26] Näin esimerkiksi suuri määrä laskentaa ei kasvata syklomaattista kompleksisuutta lainkaan, jos se ei sisällä valintarakenteita. Myöskään lausekkeiden ja muuttujien keskinäisillä suhteilla ei ole vaikutusta syklomaattiseen kompleksisuuteen [26].

Syklomaattinen kompleksisuus ei myöskään ainakaan suoraan mittaa oliokielissä ylemmän abstraktiotason kompleksisuutta, kuten luokkien keskinäisiä suhteita ja periytymistä. Näillä ei ole kuitenkaan merkitystä, jos kompleksisuutta tarkastellaan metoditasolta.

3.2 Halsteadin mittarit

Halstead esitteli mittarinsa ohjelmiston kvantitatiiviseen analysointiin vuonna 1977. Mittarit kuvaavat ohjelmistoa laajemmin kuin pelkästään kompleksisuuden kautta. [8] Mittareita on kuitenkin tutkittu paljon juuri kompleksisuuden mittaamisen näkökulmasta [25].

Halsteadin mukaan kaikki algoritmit koostuvat pohjimmiltaan vain ja ainoastaan operandeista ja operaattoreista. Niinpä Halsteadin mittarit pohjautuvat erilaisten operaattoreiden ja operandien laskemiseen ohjelmakoodista. Mittarit on johdettu erilaisten operaattoreiden lukumäärästä η_1, erilaisten operandien lukumäärästä η_2, kaikkien operaattorien kokonaislukumäärästä N_1 ja kaikkien operandien kokonaislukumäärästä N_2. [8, s. 6]

Operandit ovat ohjelmakoodin muuttujia ja vakioita. Operaattoreiksi lasketaan esimerkiksi ohjelmointikielten matemaattiset operaattorit, sijoitusoperaattori, sulkeet, puolipilkut ja valinta- ja silmukkarakenteet. Koska ohjelmakoodi koostuu ainoastaan operaattoreista ja operandeista, voidaan operaattoreiksi tulkita kaikki ohjelman osat, joita ei voida pitää operandeina. Kommenttien ei ajatella olevan osa ohjelmaa, joten niitä ei oteta lainkaan mukaan. [8, s. 7–8]

Operandien ja operaattorien määristä voidaan johtaa perusyhtälöitä koodin kvantitatiiviseen analysointiin. Ohjelmakoodin sanasto (vocabulary) η määritellään kaavalla

$$\eta = \eta_1 + \eta_2 \tag{3.6}$$

ja pituus (length) kaavalla

$$N = N_1 + N_2 \tag{3.7}$$

Kaavojen (3.6) ja (3.7) avulla saadaan laskettua algoritmin koko (volume) V, joka saadaan laskettua kaavalla

$$V = N \log_2 \eta \tag{3.8}$$

ja se kuvaa algoritmin kokoa bitteinä. [8, s. 6-19] Oletetaan, että algoritmin kirjoittaminen vaatii N valintaa η:sta operandista ja operaattorista. Koska binäärihaku on tehokkaimpia hakualgoritmeja, ja siinä järjestettyä listaa puolitetaan $\log_2 n$ kertaa, liittyy Halsteadin mukaan myös operaattorin tai operandin valintaan ainakin keskimäärin $\log_2 \eta$ mielessä tehtävää mentaalista vertailua. Tällöin algoritmin luomiseksi tarvitaan kokonaisuudessaan

$N \log_2 n$ mentaalista vertailua. Havaitaan, että johdettu kaava on sama kuin kaava (3.8). Algoritmin koon mittari mittaa siis myös algoritmin luomiseen tarvittavien mentaalisten vertailujen määrää. [8, s.46–47]

Ohjelmakoodin abstraktiotaso (program level) kuvaa sitä, kuinka paljon koodia tarvitaan saman algoritmin esittämiseen. Esimerkiksi korkeamman tason ohjelmointikielessä saman algoritmin saa esitettyä tiiviimmin verrattuna alemman tason kieliin. Abstraktiotaso riippuu käytetyn kielen lisäksi myös toteutustavasta: pelkkä funktiokutsu on korkeammalla abstraktiotasolla verrattuna varsinaiseen toteutukseen. Ohjelmakoodin abstraktiotaso L saadaan laskettua kaavalla

$$L = \frac{2}{\eta_1} \frac{\eta_2}{N_2} \tag{3.9}$$

Abstraktiotaso arvona voi olla enintään yksi. Pienempi abstraktiotason arvo tarkoittaa alempaa abstraktiotasoa. [8, s. 25–24]

Abstraktiotason ja algoritmin koon avulla voidaan laskea ohjelman tekemisen vaadittava työmäärä. Halsteadin mukaan jokainen mielessä tehtävä vertailu koostuu tietystä määrästä alkeellisia mentaalisia erotteluita. Mentaalista erottelujen määrää kutsutaan ohjelmakoodin vaikeudeksi (difficulty). Ohjelmakoodin vaikeus D saadaan laskettua abstraktiotason L käänteislukuna, jolloin saadaan kaava

$$D = \frac{1}{L} \tag{3.10}$$

Algoritmin kokonaistyömäärä E saadaan laskemalla tarvittavien vertailujen määrän ja vaikeuden tulo $E = VD$. Sijoittamalla tähän kaava (3.10) saadaan työmäärän kaavaksi

$$E = \frac{V}{L} \tag{3.11}$$

Kaavasta (3.11) nähdään, että työmäärä kuvaa algoritmin toteuttamiseen vaadittavien alkeellisten mentaalisten erottelujen kokonaismäärää. [8, s. 47]

Halstead johti työmäärästä edelleen kaavat ohjelmointivirheiden ja ohjelmointiin kuluvan ajan mittaamiseen [8, s. 82–87]. Niiden tarkastelu on kuitenkin rajattu tämän työn ulkopuolelle.

Myös Halsteadin mittarit ovat kohdanneet kritiikkiä. Operaattoreiden ja operandien laskemiselle ei ole kehitetty selkeää ja yksikäsitteistä laskutapaa, mikä on johtanut siihen, että tutkijoiden käyttämät laskutavat ja siten myös mittarien antamat tulokset vaihtelevat. Mittareita on myös käytetty eri tarkoitukseen kuin ne on alunperin kehitetty: esimerkiksi ohjelman pituuden N on tulkittu suoraan mittaavan kompleksisuutta. [23] Halsteadin mittareita on kritisoitu myös siitä, että ne eivät ota huomioon ohjelmakoodien lausekkeiden järjestystä [26].

Osassa Halsteadin mittareiden matemaattisissa määrittelyistä on havaittu olevan epäkohtia. On todistettu esimerkiksi, että työmäärän kaavalla (3.11) voidaan saada suurempi arvo saman ohjelmakoodin osalle kuin koko ohjelmakoodille. Tämä on ristiriidassa luontaisen käsityksen kanssa työmäärästä. [26] Monien Halsteadin mittareiden määrittelyiden alkuperää ja tarkoitusta pidetään epäselvänä. Halsted ei esimerkiksi kuvaillut, mitä ohjelmakoodin vaikeuden kaavan (3.10) oikea puoli tarkalleen ottaen tarkoittaa. Lisäksi osa mittareiden yksiköistä ei täsmää keskenään. [23]

3.3 Ylläpidettävyysindeksi

Oman et al. esittelivät ylläpidettävyysindeksin käsitteen artikkelissaan vuonna 1992. Heidän mukaan ohjelmakoodin ylläpidettävyys koostuu kolmesta hierarkkisesta osasta, jotka ovat seuraavat:

- valintarakenteet
- tietorakenteet
- koodin asettelu, nimeäminen ja kommentointi.

Osa-alueita nimetään myös dimensioiksi. Kukin dimensio koostuu ominaisuuksista, joita voidaan mitata. Kun yhdistetään yksittäisiä ominaisuuksia mittaavat mittarit, saadaan ylläpidettävyyttä kokonaisuutena kuvaava ylläpidettävyysindeksi. [21]

Ylläpidettävyysindeksi voidaan muodostaa dimensioiden ominaisuuksia mittaavista mittareista kaavalla

$$\prod_{i=1}^{m} W_{D_i} \left(\frac{\sum_{j=1}^{n} W_{A_j} M_{A_j}}{n}\right)_i \tag{3.12}$$

jossa W_{D_i} on ylläpidettävyyden dimension painoarvo, W_{A_j} on dimension ominaisuuden painoarvo ja M_{A_j} on ominaisuutta mittaavan mittarin antama tulos. Ylläpidettävyysindeksi saadaan siis laskettua painotettujen dimensioiden tulona, jossa kunkin dimension arvo on keskipoikkeama sen ominaisuuksia mittaavista attribuuteista. [21]

Ominaisuuksia mittaavista mittareista laskettavat keskipoikkeamat eli dimensioiden arvot tulee muodostaa siten, että ne ovat yhden ja nollan välillä. Dimensioiden arvojen tulo eli ylläpidettävyysindeksi voidaan siis myös esittää prosenttilukuna. Mitä lähempänä dimensioiden ja ylläpidettävyysindeksin arvot ovat yhtä, sitä parempi se on ylläpidettävyyden kannalta. Omanin et al. mukaan kaikkia dimensioiden ominaisuuksia ei tarvitse mitata, vaan riittää, että dimensioita parhaiten kuvaavat arvot otetaan mukaan. [21]

Coleman et al. tutkivat dimensioita parhaiten kuvaavia mittareita. He tutkivat noin 40 kompleksisuusmittaria ja vertasivat niiden antamia tuloksia Hewlett-Packardin insinöörien subjektiivista arviota vasten. Tutkimusta varten muodostettiin noin 50 regressiomallia. [5]

Tutkimuksen havaittiin, että Halsteadin mittareiden puutteista huolimatta Halsteadin algoritmin koko (3.8) ja työmäärä (3.11) ennustivat parhaiten ohjelmakoodin ylläpidettävyyttä. Parhaiten insinöörien arviota vastaava malli koostui Halsteadin työmäärästä, laajennetusta syklomaattisesta kompleksisuudesta, koodirivien määrästä ja kommenttien määrästä. Mallin mukaan ylläpidettävyysindeksi MI saadaan laskettua kaavalla

$$MI = 171 - 3,42 \times \ln aveE - 0,23 \times aveV(g') - 16,2 \times \ln aveLOC + aveCM \quad (3.13)$$

jossa $aveE$, $aveV(g')$, $aveLOC$, $aveCM$ kuvaavat funktioiden keskimääräistä Halsteadin työmäärää, laajennettua syklomaattista kompleksisuutta, koodirivien määrää ja kommenttien määrää. Laajennetulla syklomaattisella kompleksisuudella viitataan siihen, että ehtojen lisäksi lasketaan ohjelmakoodista Boolen operaattoreita. Malli saa arvoja väliltä 0 – 100. [5]

Ensimmäinen versio ylläpidettävyysindeksistä oli kuitenkin liian herkkä kommenteille. Varsinkin pitkät kommenttilohkot vaikuttivat liikaa ylläpidettävyysindeksiin. Sen vuoksi keskimääräisten kommenttien määrän tilalle otettiin kommenttien suhteellinen osuus funktiossa $perCM$. Halsteadin työmäärän tilalle otettiin keskimääräinen algoritmin koko $aveVol$, sillä työmäärää oli kritisoitu sen epämonotonisesta käyttäytymisestä ohjelmakoodin osia yhdistettäessä. [5]

Ylläpidettävyysindeksi, johon päädyttiin, saadaan laskettua kaavalla

$$\begin{aligned} MI =& 171 - 5,2 \times \ln aveVol - 0,23 \times aveV(g') \\ & - 16,2 \times \ln(aveLOC) + (50 \times \sin \sqrt{2,46 \times perCM}) \end{aligned} \quad (3.14)$$

Kaavan (3.14) antamat lukemat vaihtelivat analyysissa välillä -91 – 183 [5]. Matemaattista alarajaa ei ole, mutta ylläpidettävyyden kannalta ei ole havaittu eroa nollaa lähestyvällä ja negatiivisella ylläpidettävyysindeksillä [15]. Alle 65 oleva ylläpidettävyysindeksi tarkoittaa vaikeaa ylläpidettävyyttä. Ylläpidettävyysindeksi välillä 65 – 85 tarkoittaa kohtalaista ylläpidettävyyttä. Komponentit, joiden ylläpidettävyysindeksi on yli 85, ovat helposti ylläpidettäviä. [5]

Microsoftin Visual Studio -ohjelmointiympäristöä varten Colemanin et al. ylläpidettävyysindeksiä on kehitetty edelleen. Koska negatiivisten arvojen ei todettu tuovan mitään lisätietoa ylläpidettävyydestä, muutettiin kaavaa siten, että ylläpidettävyysindeksin arvo on vähintään nolla. Kaava antaa suurimmillaan ylläpidettävyysindeksiksi arvon 100. Visual Studion laskema ylläpidettävyysindeksi saadaan laskettua kaavalla [5]

$$MI = \max\{0, (171 - 5,2 \times \ln aveVol - 0,23 \times aveV(g) - \ln aveLOC) \times \frac{100}{171}\} \quad (3.15)$$

Visual Studion ylläpidettävyysindeksissä arvot välillä 0 – 9 tarkoittavat alhaista ylläpidettävyyttä, välillä 10 – 19 kohtalaista ylläpidettävyyttä ja välillä 20 – 100 hyvää ylläpidettävyyttä [4].

3.4 Koodirivien lukumäärä

Koodirivien laskeminen on yksi vanhimmista ohjelmistometriikoista. Sitä käytetty hyvin paljon kompleksisuusmittarina. Yksi tunnetuimmista laskutavoista koodirivien lukumäärälle on se, että lasketaan koodista kaikki rivit, jotka eivät ole kommentteja tai tyhjiä rivejä. [10, s. 87–88]

Koodirivien lukumäärän käyttämisestä kompleksisuusmittarina on paljon ristiriitaisia kokemuksia. Sillä on todettu olevan monia vahvuuksia. Koodirivien määrä on intuitiivinen mittari, joka on helppo laskea ja ymmärtää. Koodirivien määrän laskemisen on todettu useammassa tutkimuksessa korreloivan todellisen työmäärän kanssa paremmin kuin Halsteadin työmäärän. Lisäksi se on ollut vähintään yhtä hyvä kuin McCaben syklomaattinen kompleksisuus. [10, s. 88]

Koodirivien laskemisessa ei oteta huomioon rivien sisäistä kompleksisuutta, mikä on yksi sen suurimmista heikkouksista. Näin ollen esimerkiksi lukuisia sisäkkäisiä rakenteita, muuttujien välisiä suhteita ja silmukkarakenteita ei huomioida tuloksissa erityisemmin. Lisäksi koodirivien määrälle esitetyt lukuisat laskutavat aiheuttavat vaihteluja tuloksissa. [10, s.87–88]

4 MITTAREIDEN KÄYTTÄYTYMISEN TUTKIMINEN

4.1 Tutkimuksen toteutus

Tutkimusta varten rajattiin pois osa kappaleessa 3 esitellyistä mittareista, jotta mittareiden tuloksiin pystyttiin perehtymään tarkemmin. Ylläpidettävyysindekseistä otettiin mukaan Microsoft Visual Studion ylläpidettävyysindeksi (3.15), koska sen antamat tulokset ovat helpoimmin ymmärrettävissä selkeiden ylä- ja alarajojen ansiosta. Valintaan vaikutti myös se, että kyseinen versio ylläpidettävyysindeksistä on integroitu osaksi tunnettua kehitysympäristöä. Halsteadin mittareista valittiin puolestaan Halsteadin ohjelmakoodin vaikeus (3.10) ja työmäärä (3.11), koska kumpaakaan niistä ei lasketa suoraan osana mukaan valittua ylläpidettävyysindeksiä.

McCaben syklomaattinen kompleksisuus (3.2) otettiin mukaan sellaisenaan, sillä McCaben alkuperäinen versio on edelleen tunnetuin syklomaattisen kompleksisuuden mittari. Vertailukohteeksi valitusta koodirivien lukumäärästä otettiin mukaan alaluvussa 3.4 esitelty versio.

Valittujen mittareiden ominaispiirteet ja niiden väliset eroavaisuudet pyrittiin saamaan esille analysoimalla ohjelmallisesti JSON-tiedostojen käsittelyyn luodun Jackson-projektin koodia. Jackson on Java-kielellä kirjoitettu avoimen lähdekoodin projekti, jota käytetään laajalti Java-ohjelmoinnissa [7]. Projektin suuren koon vuoksi analysoitava koodi rajattiin projektin databind-pakettiin.

Analyysiin otettiin mukaan pelkästään metodit, joilla oli myös toteutus. Näin esimerkiksi interface-rajapinnat jätettiin analyysissa kokonaan huomioimatta. Lopulta mukaan tuli 6448 metodia.

Analysointia varten vertailtiin kahta eri analysointiohjelmistoa: SourceMeteriä ja Jhawkia. Jhawk on Java-kielen analysoimiseen tehty ohjelma, joka laskee tulokset kaikille tässä työssä esitellyille mittareille [13]. SourceMeter tukee Java-kielen lisäksi monia muita tunnettuja ohjelmointikieliä. SourceMeterillä voidaan laskea kompleksisuuksia metoditason lisäksi mm. luokka- ja pakkaustasolta. Myös se tukee kaikkia tässä työssä esiteltyjä kompleksisuusmittareita. [24]

Analyysiin valittiin SourceMeter sen ilmaisversion laajemman toiminnallisuuden ansiosta. Jhawkin ilmaisella kokeiluversiolla pystyi analysoimaan vain muutamia Java-tiedostoja kerrallaan, kun taas SourceMeterin ilmaisversio mahdollisti huomattavasti suurempien

projektien analysoinnin kerralla. JHawk osoittautui toisaalta kokemattomalle käyttäjälle helppokäyttöisemmäksi sen graafisen käyttöliittymän ansiosta.

SourceMeter laskee myös rivimäärät usealla eri tavalla. Ylläpidettävyysindeksin laskemisessa käytetään ohjelmassa rivimäärille laskutapaa, jossa lasketaan kaikki rivit, jotka eivät ole kommentteja, tyhjiä rivejä tai anonyymejä luokkia [24]. Tutkimuksen korrelaatiomatriisiin ja hajontakaavioihin on valittu puolestaan laskutapa, joissa myös anonyymit luokat lasketaan mukaan koodiriveihin. Anonyymejä luokkia käytetään kuitenkin valitussa tutkimusaineistossa suhteellisen harvoin, joten niillä ei pitäisi olla suurta vaikutusta tuloksiin.

Varsinainen tutkimus koostui kahdesta vaiheesta. Ensimmäisessä vaiheessa tutkittiin mittareiden käyttäytymistä suhteessa toisiinsa käyttäen tilastollisia menetelmiä. Aineiston pohjalta laskettiin korrelaatiomatriisi mittareille.

Korrelaatio ei kerro kaikkea mittareiden välisistä suhteista, sillä esimerkiksi eksponentiaaliset suhteet näkyvät pienempänä korrelaatioina verrattuna lineaarisiin. Korrelaatioiden tueksi mittareiden käyttäytymistä kuvaamaan piirrettiin niistä hajontakaaviot rivimäärien kanssa. Hajontakaaviot havainnollistavat mittarien käyttäytymistä rivimäärien kasvaessa sekä lisäksi niissä on näkyvissä mittarien tulosten jakaumaa.

Vertailukelpoisuuden parantamiseksi käytettiin korrelaatiomatriisin laskennassa vastalukuja ylläpidettävyysindeksin antamista tuloksista. Tämä tehtiin, koska ylläpidettävyysindeksi käyttäytyy päinvastoin kuin muut kompleksisuusmittarit: se saa sitä pienempiä arvoja, mitä heikommin ylläpidettävää koodi on. Hajontakaaviossa sen sijaan käytettiin ylläpidettävyyden antamia arvoja sellaisenaan, jotta kaavion asteikko vastaa mittarin antamia lukemia.

Toisessa vaiheessa tutkimusta etsittiin metodeista koodirakenteita, jotka näkyivät suurempana kompleksisuutena mittareiden antamissa tuloksissa. Tavoitteena oli löytää eroja mittareiden väliltä. Aluksi tutkittiin kunkin mittarin osoittamaa kolmea kompleksisinta metodia. Ongelmaksi kuitenkin muodostui se, että melkein kaikilla mittareilla löydettiin samat kolme metodia. Näin ollen mittareiden erot eivät tulleet esille.

Erojen löytämiseksi päätettiin tehdä lisätutkimusta. Koodirivien määrän vaikutus pyrittiin eliminoimaan sillä, että vertailtiin keskenään yhtä pitkiä metodeja. Metodien pituuksien keskiarvo oli noin 20 riviä, joten mukaan otettiin metodit, joiden pituus oli 10, 20 tai 30 koodiriviä. Samanpituisista metodeista etsittiin erikseen kullakin rivimäärällä ja kunkin mittarin kohdalla mittarin mukaan eniten ja vähiten kompleksisimmat metodit. Pelkkiä luokkamuuttujien alustuksia sisältäneet rakentajametodit jätettiin pois tarkastelusta, sillä ne olivat kaikkien mittareiden mukaan kaikista vähiten kompleksisia metodeja. Lopulta mittareiden välisiä eroja saatiin paremmin esille, kun rivimäärä ei aiheuttanut eroa keskenään vertailtavien metodien kompleksisuuteen.

Taulukko 4.1. *Mittareiden antamista tuloksista laskettu korrelaatiomatriisi*

	D	E	$-MI$	$v(g)$	Rivilkm
D	1,000				
E	0,790	1,000			
$-MI$	0,817	0,506	1,000		
$v(g)$	0,850	0,797	0,721	1,000	
Rivilkm	0,912	0,831	0,823	0,942	1,000

4.2 Tutkimuksen tulokset

4.2.1 Mittareiden keskinäinen korrelaatio

Kullekin metodille lasketuista Halsteadin vaikeudesta D, Halsteadin työmäärästä E, Visual Studion ylläpidettävyysindeksin vastaluvusta $-MI$, syklomaattisesta kompleksisuudesta $v(g)$ ja koodirivien lukumäärästä laskettiin edelleen taulukossa 4.1 esitetty korrelaatiomatriisi. Kullekin mittarien väliselle korrelaatiolle otoskoolla 6448 laskettu 2-suuntainen p-arvo oli alle 0,0001, joten kaikkia korrelaatioita voidaan pitää tilastollisesti merkittävänä.

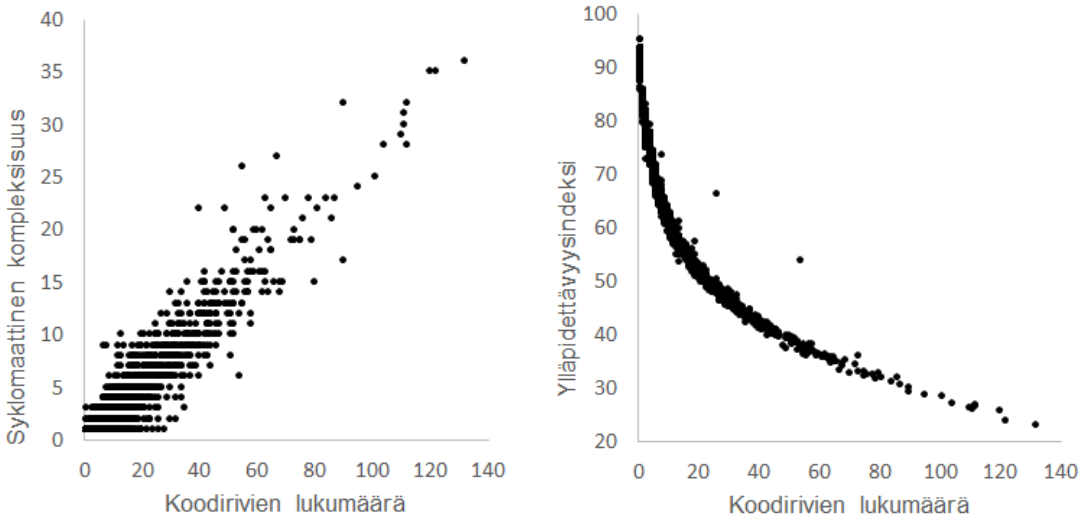
Tuloksista havaitaan, että kaikkien mittareiden välillä on positiivista korrelaatiota. Sekä syklomaattisessa kompleksisuudessa, Halsteadin mittareissa, rivimäärien laskemisessa että näistä johdetussa ylläpidettävyysindeksissä lasketaan koodista erilaisia osia. Suuri mittareiden välinen korrelaatio johtunee siitä, että koodin määrän kasvaessa myös näiden osien, kuten ehtojen tai operandien, määrä todennäköisesti kasvaa.

Suurin mittareiden välinen korrelaatio oli arvoltaan 0,942 ja löytyi hieman yllättäen syklomaattisen kompleksisuuden ja koodirivien lukumäärän väliltä. Korrelaatio oli pienin Halsteadin työmäärän ja ylläpidettävyysindeksin vastaluvun välillä ja oli arvoltaan 0,506.

Korrelaatiomatriisin tueksi piirrettiin valittujen mittareiden ja koodirivien lukumäärän välille hajontakaaviot. Syklomaattisen kompleksisuuden ja ylläpidettävyysindeksin hajontakaaviot rivimäärien kanssa on esitetty kuvassa 4.1.

Syklomaattisen kompleksisuuden ja rivimäärän hajontakaaviosta on nähtävissä lineaarinen riippuvuus mittareiden välillä. Kaaviossa näkyvät vaakasuuntaiset raidat havainnollistavat syklomaattisen kompleksisuuden karkeutta. Hyvin monet metodit saavat saman syklomaattisen kompleksisuuden arvon varsinkin vähemmän kompleksisissa metodeissa.

Ylläpidettävyysindeksin ja rivimäärien suhde vaikuttaa hajontakaavion perusteella hyvin vahvalta, vaikka se ei erottunutkaan korrelaatiomatriisissa. Hajontakaavion kuvio muistuttaa negatiivista logaritmifunktiota. Ylläpidettävyysindeksin kaava (3.15) vahvistaa havaintoa: siinä on mukana negatiiviset logaritmit sekä rivimäärästä että Halsteadin algoritmin koosta.



Kuva 4.1. *Syklomaattisen kompleksisuuden ja ylläpidettävyysindeksin hajontakaaviot rivimäärien kanssa.*

Tutkimalla ylläpidettävyysindeksin ja rivimäärän hajontakaaviossa kahta muista pisteistä erossa olevaa pistettä vastaavaa metodia havaittiin, että kummassakin metodissa käytettiin anonyymiä luokkaa. Poikkeamat johtuivat siis laskuteknisestä epäyhdenmukaisuudesta. Hajontakaavion rivimäärään laskettiin mukaan anonyymit luokat, mutta ylläpidettävyysindeksin rivimäärää laskettaessa ne jätettiin huomioimatta.

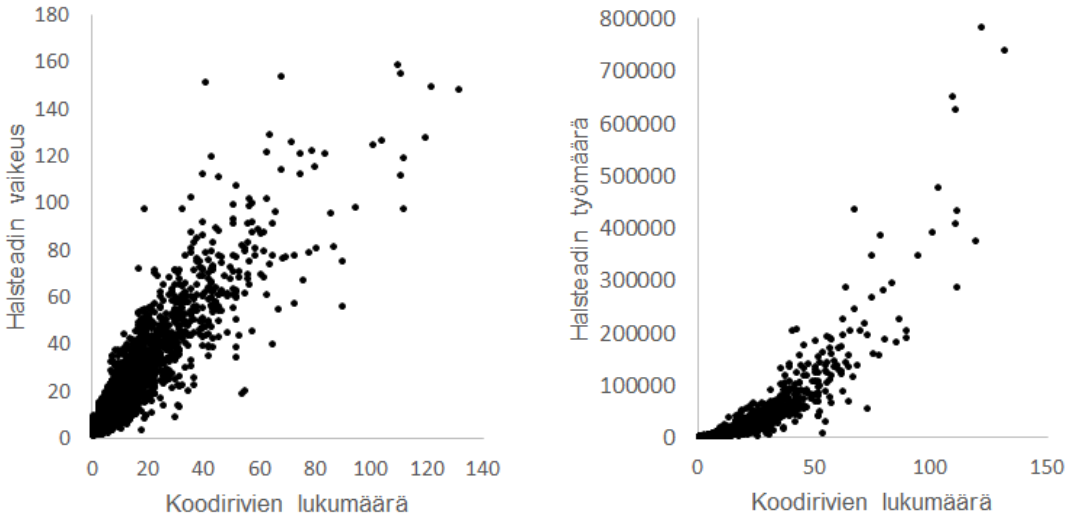
Halsteadin mittareiden ja rivimäärien hajontakaaviot on esitetty kuvassa 4.2. Halsteadin vaikeus näyttäisi kasvavan melko lineaarisesti koodirivien lukumäärän myötä. Mitä suuremmaksi rivimäärät kasvavat, sitä enemmän kaavion pisteet hajaantuvat toisistaan.

Halsteadin työmäärän käyttäytyminen koodirivimäärien kasvaessa muistuttaa eksponentiaalista funktiota. Myös sen hajontakaaviossa pisteet hajaantuvat koodirivien kasvamisen myötä. Hajontakaaviosta nähdään, että Halsteadin työmäärän saaman arvon skaala on suuri verrattuna muihin mittareihin. Tässä mittauksessa saatu suurin arvo Halsteadin työmäärälle oli 779 213 ja pienin arvo noin 4,8.

4.2.2 Huomioita kompleksisista metodeista

Työn toisen vaiheen alussa tehtiin havaintoja kolmesta kompleksisimmasta metodista kullekin mittarille. Mittareiden korrelaatio näkyi tässäkin vaiheessa odotettua enemmän. Sekä syklomaattinen kompleksisuus, koodirivien lukumäärä että ylläpidettävyysindeksi nostivat esille kolme samaa metodia samassa järjestyksessä.

Sen lisäksi, että nämä kaikki metodit olivat pitkiä, käytettiin niissä paljon ehtolauseita ja silmukkarakenteita. Ehtolauseet ja silmukkarakenteet olivat melko kevyitä ja useimmiten niiden sisällä oli vain yksi rivi koodia. Kaikissa metodeissa käytettiin myös switch-lausetta. Kolmanneksi kompleksisimmassa metodissa switch-lause oli huomattavasti pidempi kuin



Kuva 4.2. *Syklomaattisen kompleksisuuden ja ylläpidettävyysindeksin hajontakaaviot rivimäärien kanssa.*

muissa ja se koostui 12 osasta. Pitkän switch-lauseen sisältänyt metodi oli rakenteellisuutensa ansiosta helposti ymmärrettävä. Metodin pilkkominen pienempiin osiin olisi mahdollisesti tässä tapauksessa heikentänyt luettavuutta, vaikka mittareissa se olisi näkynyt pienempänä kompleksisuutena.

Sisäkkäisyyttä metodeissa oli käytetty säästeliäästi lukuun ottamatta kaikista kompleksisimmaksi mitattua metodia. Siinä oli korkeimmillaan yhteensä seitsemän sisäkkäistä valinta- ja silmukkarakennetta. Ennen kyseistä koodirakennetta oli kommentti, jossa valitettiin sen olevan kopioitu toisesta luokasta. Lisäksi kommentissa mainittiin, että kyseinen koodirakenne kuuluisi muualle, mutta sitä ei voida helposti muokata sellaiseksi, että siirtäminen onnistuisi. Vaikea siirrettävyys ja useista sisäkkäisistä elementeistä koostuva monimutkainen koodirakenne viittaa siihen, että kyseinen kohta on ylläpidettävyyden kannalta ongelmallinen.

Koska metodien käyttökonteksteista tai esimerkiksi projektissa käytetyistä koodauskonventioista ei ollut syvällisempää tietoa, ei koodin luettavuutta voitu arvioida kovin perusteellisesti. Ehtolauseiden ja silmukkarakenteiden runsaudesta huolimatta koodi oli luettavaa niiden keveyden ja peräkkäisyyden ansiosta. Osassa tutkituissa metodeissa koodin pilkkominen muutamaan pienempään ja hyvin nimettyyn metodiin olisi voinut parantaa luettavuutta.

Kolmea suurinta Halsteadin vaikeuden arvoa tutkimalla esille nousi kolme täysin eri metodia. Nämä metodit olivat hyvin samankaltaisia kuin aiemmin tutkitut ja sisälsivät paljon ehtolauseita ja silmukkarakenteita. Switch-lauseita ei esiintynyt missään näistä metodeista.

Halsteadin työmäärällä ei löydetty enää uusia metodeja. Sen löytämissä metodeissa oli yksi yhteinen metodi Halsteadin vaikeuden kanssa ja kaksi yhteistä metodia syklomaattisen kompleksisuuden, ylläpidettävyysindeksin ja koodirivimäärien kanssa.

Tutkimusta jatkettiin tutkimalla keskenään yhtä pitkiä metodeja kullekin mittarille, jolloin mittareiden väliset erot saatiin paremmin esille. Suurimman syklomaattisen kompleksisuuden arvon saaneissa metodeissa esiintyi switch-lause, lyhyitä peräkkäisiä ehtolauseita, paljon yhtäsuuruusvertailuja ja Boolen-operaattoreita. Switch-lauseen sisältäneen metodin kanssa yhtä pitkä metodi, jonka syklomaattinen kompleksisuus oli samanpituisista metodeista pienin, sisälsi puolestaan sisäkkäisiä ehtolauseita, poikkeuskäsittelijän, eksplisiittisen tyyppimuunnoksen ja useita metodikutsuja.

Yksinkertaisen switch-lauseen sisältäneen metodin toiminta oli tässä tapauksessa helpommin ymmärrettävissä, koska switch-lause on yksittäinen ohjelmointikielen rakenne, joka toimii kaikkialla samalla tavalla. Pienimmän syklomaattisen kompleksisuuden saanut samanpituinen metodi koostui puolestaan useasta erilaisesta sisäkkäisestä rakenteesta, mikä teki siitä raskaamman lukea. Switch-lauseen sisältäneen metodin syklomaattinen kompleksisuus oli 14, mikä on yli suositellun rajan. Tähän metodiin verratulla samanpituisella metodilla syklomaattisen kompleksisuuden arvo oli vain 5.

Halsteadin vaikeus ja työmäärä käyttäytyivät hyvin eri tavoin kuin syklomaattinen kompleksisuus. Vähiten kompleksisissa metodeissa esiintyi yksinkertaisia switch-lauseita ja lyhyitä peräkkäisiä ehtolauseita. Kompleksisimmissa metodeissa puolestaan käytettiin mm. ehto-operaattoreita (ternary operator), eksplisiittisiä tyyppimuunnoksia ja poikkeuskäsittelijöitä sisäkkäisten ehtolauseiden ja silmukkarakenteiden lisäksi. Niissä rivit olivat huomattavasti pidempiä. Halsteadin vaikeuden ja työmäärän väliltä ei löydetty eroja, sillä niiden avulla löydetyt metodit olivat keskenään samantyyppisiä ja suurimmaksi osaksi samoja.

Ylläpidettävyysindeksi käyttäytyi odotetusti. Koska se pohjautuu Halsteadin mittareihin ja syklomaattiseen kompleksisuuteen, löydettiin sen avulla samoja metodeja kuin syklomaattisella kompleksisuudella, Halsteadin vaikeudella ja Halsteadin työmäärällä oli löydetty. Switch-lauseita esiintyi sekä kompleksisimmissa että vähiten kompleksisissa metodeissa. Mitään selkeitä epäkohtia kompleksisimpien ja vähiten kompleksisten metodien kesken ei huomattu: vähiten kompleksisimmissa metodeissa rivit olivat lyhyempiä ja niissä oli vähemmän muuttujien vertailua verrattua kompleksisimpiin metodeihin.

4.3 Tulosten oikeellisuuden arviointi

Tutkimustuloksia tarkasteltaessa on otettava huomioon moni niiden luotettavuuteen vaikuttava asia. Tutkimusaineisto koostui vain yhdestä avoimen lähdekoodin projektista. Toisaalta aineisto oli siitä huolimatta laaja ja sisälsi 6448 metodia. Luotettavuutta paransi myös se, että aineistona oli laajasti tunnettu ja käytetty Java-projekti, eikä esimerkiksi tutkimusta varten ohjelmoitu esimerkkiprojekti.

Laajan projektin käyttäminen oli mahdollista, koska laskenta suoritettiin ohjelmallisesti. Kompleksisuuden laskeminen ohjelmallisesti vaikutti myös tulosten luotettavuuteen. Ohjelman käyttämisestä ei ollut aikaisempaa kokemusta, joten sen käyttämisessä saatettiin

tehdä virheitä, jotka jäivät huomaamatta. Myöskään kaikkia ohjelman tekemiä oletuksia metodien analyysissä ei välttämättä osattu ottaa huomioon. Yhtenä esimerkkinä tästä on se, että ylläpidettävyysindeksissä käytettiin eri laskutapaa rivimäärille kuin hajontakaaviossa. Halsteadin mittareiden ja siten myös ylläpidettävyysindeksien tuloksissa havaittiin pientä vaihtelua eri analysointiohjelmien välillä. Se johtuu todennäköisesti siitä, että Halsteadin mittareiden operandeille ja operaattoreille ei ole määritelty yksikäsitteistä laskutapaa.

Kokonaisuudessaan tutkimuksen ensimmäisen vaiheen tilastollista analyysia voidaan pitää luotettavampana kuin toisessa vaiheessa tehtyjä huomioita. Toisessa vaiheessa käytiin läpi huomattavasti vähemmän metodeja, mikä voi vääristää tuloksia. Luotettavuutta paransi kuitenkin se, että metodit valittiin SourceMeterin tekemän analyysin pohjalta. Tutkimuksen kompleksisimpien metodien valitseminen vastasi mittareiden yhtä käyttötapaa, jossa mittareita käytetään ongelmakohtien etsimiseen koodista.

5 YHTEENVETO

Taulukossa 5.1 esitetään yhteenvetona työssä tutkittujen kompleksisuusmittareiden tärkeimpiä ominaisuuksia. Taulukossa helpolla laskettavuudella viitataan siihen, että mittarin arvot voidaan helposti laskea yksittäisistä metodeista myös käsin ilman ohjelmallista apua. Mittareiden ominaisuuksien löytämisen lisäksi työn yhtenä tärkeimpänä havaintona oli se, että kaikki työssä esitellyt mittarit olivat melko vahvasti riippuvaisia rivimäärästä. Rivimäärä osoittautui oletettua paremmaksi työkaluksi mitata koodin kompleksisuutta.

Mittareiden toiminnan väliltä löydettiin monia eroja. Syklomaattiselta kompleksisuudeltaan kompleksisimmat metodit sisälsivät switch-lauseita ja paljon lyhyitä ehtolauseita ja silmukkarakenteita. Halsteadin vaikeuden ja työmäärän mukaan kompleksisimmat metodit sisälsivät keskimäärin pidempiä rivejä, ja niissä käytettiin enemmän erilaisia koodirakenteita. Ylläpidettävyysindeksiltään kompleksisimmissa metodeissa esiintyi vaihtelevasti sekä paljon ehtoja että pitkiä rivejä ja erilaisten rakenteiden käyttöä.

Yksikään työssä esitelty mittari ei mitannut metodien kompleksisuuden kaikkia osa-alueita. Kompleksisuus on laaja ja osittain subjektiivinen käsite, joten sen esittäminen kokonaisuudessaan yhtenä mittarin näyttämänä lukuarvona on hyvin haastavaa. Lisäksi useita ominaisuuksia mittaavien mittareiden antaman lukuarvon tulkitseminen ei ole aina helppoa. Esimerkiksi ylläpidettävyysindeksiltään kompleksisessa metodissa ei pelkkää lukuarvoa tarkastelemalla voida tietää, onko sen taustalla liian suuri syklomaattinen kompleksisuus, rivimäärä vai Halsteadin algoritmin koko.

Yksi hyvä lähestymistapa kompleksisuuden seuraamiseen voisi siis olla käyttää erikseen useampaa yksittäistä ominaisuutta mittaavaa mittaria. Toisaalta useamman mittarin seuraaminen on työläämpää kuin yhden. Mittareiden pitäisi olla riittävän yksinkertaisia, jotta niiden lukemia olisi helppo tulkita. Tällaisia ominaisuuksia voisivat olla esimerkiksi rivimäärä, sisäkkäisyys, paikallisten muuttujien määrä ja niin edelleen. Monipuolisesti eri kompleksisuuden osa-alueita mittaavien yksinkertaisten mittarien etsiminen voisi olla hyvä suunta jatkotutkimukselle.

Valituista mittareista riippumatta kompleksisuuden mittaamisessa on riskinä se, että mittareiden annetaan liikaa ohjata ohjelmointia. Vaikka esimerkiksi rivimäärän todettiin korreloivan kaikkien työssä esiteltyjen mittareiden tulosten kanssa, ei metodien pilkkominen pienemmiksi automaattisesti paranna koodin luettavuutta. Koodin luettavuus voi päinvastoin heiketä, jos esimerkiksi yksi hyvin nimetty ja selkeän kokonaisuuden muodostava metodi pilkotaan väkinäisesti useaan huonosti nimettyyn metodiin. Ohjelmistokehittäjän

Taulukko 5.1. *Yhteenveto mittareiden ominaisuuksista.*

Mittari	Ominaisuudet
Syklomaattinen kompleksisuus	Perustuu suorituspolkujen laskemiseen Helppo laskea Määritelty suositellut rajat Korostaa switch-lauseita Moni metodi saa saman arvon
Halsteadin vaikeus ja työmäärä	Perustuu operandien ja operaattoreiden laskemiseen Matemaattisessa määrittelyssä epäkohtia Ei yksikäsitteistä laskutapaa
Ylläpidettävyysindeksi	Yhdistelmä muista mittareista Määritelty suositellut rajat
Koodirivien lukumäärä	Helppo laskea Intuitiivinen ymmärtää Ei huomioi rivien sisäistä kompleksisuutta

tulee siis edelleen pelkkien mittareiden seuraamisen sijaan käyttää myös omaa harkintakykyään ja ammattitaitoaan laadukkaan koodin tuottamiseksi.

LÄHTEET

[1] S. Ajami, Y. Woodbridge ja D. G. Feitelson. Syntax, predicates, idioms — what really affects code complexity? *Empirical Software Engineering* (2018). DOI: 10.1007/s10664-018-9628-3. URL: https://doi.org/10.1007/s10664-018-9628-3.

[2] V. Antinyan, M. Staron ja A. Sandberg. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. *Empirical Software Engineering* 22.6 (2017), 3057–3087. DOI: 10.1007/s10664-017-9508-2. URL: https://doi.org/10.1007/s10664-017-9508-2.

[3] R. D. Banker, S. M. Datar, C. F. Kemerer ja D. Zweig. Software Complexity and Maintenance Costs. *Commun. ACM* 36.11 (1993), 81–94. ISSN: 0001-0782. DOI: 10.1145/163359.163375. URL: http://doi.acm.org/10.1145/163359.163375.

[4] *Code metrics values. Software measurements*. 2007. URL: https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/ (viitattu 24. 02. 2019).

[5] D. Coleman, D. Ash, B. Lowther ja P. Oman. Using metrics to evaluate software system maintainability. *Computer* 27.8 (1994), 44–49. ISSN: 0018-9162. DOI: 10.1109/2.303623.

[6] C. Ebert, J. Cain, G. Antoniol, S. Counsell ja P. Laplante. Cyclomatic Complexity. English. *IEEE Software* 33.6 (2016), 27–29. DOI: 10.1109/MS.2016.147.

[7] *FasterXML*. URL: https://github.com/FasterXML/jackson (viitattu 04. 03. 2019).

[8] M. H. Halstead. *Elements of software science*. eng. New York: Elsevier, 1977, 127 sivua. ISBN: 0-444-00205-7. URL: https://tuni.finna.fi/Record/tutcat.42902.

[9] M. Harsu. *Ohjelmien ylläpito ja uudistaminen*. Helsinki: Talentum, 2003, 57. ISBN: 951-762-829-3. URL: https://tuni.finna.fi/Record/tutcat.163796.

[10] B. Henderson-Sellers. *Object-oriented Metrics: Measures of Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996. ISBN: 0-13-239872-9.

[11] *ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary*. 2010. DOI: 10.1109/IEEESTD.2010.5733835.

[12] A. Jbara ja D. G. Feitelson. On the Effect of Code Regularity on Comprehension. *Proceedings of the 22Nd International Conference on Program Comprehension*. ICPC 2014. Hyderabad, India: ACM, 2014, 189–200. ISBN: 978-1-4503-2879-1. DOI: 10.1145/2597008.2597140. URL: http://doi.acm.org.libproxy.tuni.fi/10.1145/2597008.2597140.

[13] *JHawk Product Overview*. URL: http://www.virtualmachinery.com/jhawkprod.htm (viitattu 05. 03. 2019).

[14] D. Kafura ja G. R. Reddy. The Use of Software Complexity Metrics in Software Maintenance. *IEEE Transactions on Software Engineering* SE-13.3 (1987), 335–343. DOI: 10.1109/TSE.1987.233164.

[15] *Maintainability Index Range and Meaning*. 2018. URL: https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/ (viitattu 24. 02. 2019).

[16] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2.4 (1976), 308–320.

[17] S. McConnell. *Code complete*. eng. 2nd ed. Redmond, WA: Microsoft Press, 2004, 914 sivua. ISBN: 0-7356-1967-0. URL: https://tuni.finna.fi/Record/oma.111462.

[18] *Metric Definitions. Complexity*. URL: https://docs.sonarqube.org/latest/user-guide/metric-definitions/ (viitattu 27. 12. 2018).

[19] H. D. Mills. Mathematical foundations for structured programming (1972). URL: https://trace.tennessee.edu/utk%5C_harlan/56/.

[20] S. Y. Moon, B. K. Park ja R. Y. C. Kim. Code Complexity on Before and After Applying Design Pattern through SW Visualization. *2016 International Conference on Platform Technology and Service (PlatCon)*. 2016, 1–5. DOI: 10.1109/PlatCon.2016.7456787.

[21] P. Oman ja J. Hagemeister. Metrics for assessing a software system's maintainability. *Proceedings Conference on Software Maintenance 1992*. IEEE, 1992, 337–344. DOI: 10.1109/ICSM.1992.242525.

[22] M. E. Pesonen. *Verkkoteorian alkeita*. Itä-Suomen Yliopisto. Joensuu, 2013. URL: http://cs.uef.fi/matematiikka/kurssit/MathematicsVisualizationMedia/CourseMaterial/VerkkoteoriaaSciFestiin2013.pdf.

[23] R. Al-Qutaish ja A. Abran. An Analysis of the Design and Definitions of Halstead's Metrics. English. *In Proceedings of the 15th International Workshop on Software Measurement*. 2005, 337–352.

[24] *SourceMeter 8.2 for Java*. URL: https://www.sourcemeter.com/resources/java/ (viitattu 04. 03. 2019).

[25] I. Uk. *Software Complexity Measurement: A Critical Review*. Vol. 01. 2016, 12–16.

[26] E. J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering* 14.9 (1988), 1357–1365. ISSN: 0098-5589. DOI: 10.1109/32.6178.

[27] K. Yamashita, C. Huang, M. Nagappan, Y. Kamei, A. Mockus, A. E. Hassan ja N. Ubayashi. *Thresholds for Size and Complexity Metrics: A Case Study from the Perspective of Defect Density*. 2016, 191–201. DOI: 10.1109/QRS.2016.31.