Ilari Kauko

Staattiset koodimetriikat ja niiden yhteys muutoksen määrään TIMissä

Tietotekniikan pro gradu -tutkielma

18. joulukuuta 2023

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Ilari Kauko

Yhteystiedot: ilari.o.kauko@student.jyu.fi

Ohjaaja: Tommi Mikkonen

Työn nimi: Staattiset koodimetriikat ja niiden yhteys muutoksen määrään TIMissä

Title in English: Static code metrics and their relation to the number of changes in TIM

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmisto- ja tietoliikennetekniikan opintosuunta

Sivumäärä: 78+0

Tiivistelmä: Staattiset koodimetriikat ovat tapoja mitata ohjelmakoodia suorittamatta sitä. Yleensä niitä käytetään arvioimaan ohjelmakoodin laatua, kuten ymmärrettävyyttä ja ylläpidettävyyttä, mutta niiden soveltuvuus siihen on melko kyseenalaista. Aihetta on tutkittu noin 50 vuotta ilman yksiselitteisiä tuloksia. Ohjelmistoteollisuus käyttää niitä melko vakituisesti koodin validointiin, mutta akateeminen tutkimus niistä on antanut melko ristiriitaisia tuloksia. Jyväskylän yliopistossa koulutuskäytön web-sovellukseksi kehitetyn TIMin lähdekoodia tutkittiin saatavilla olevilla staattisten koodimetriikoiden mittaustyökaluilla ja paljastui, että useat metriikat ovat merkittävästi yhteydessä koodia muuttaneiden Git-commitien määrään. Toisaalta tämän tutkielman aineiston analyysissä korostuivat jonkin verran eri metriikat kuin aiemmassa tutkimuskirjallisuudessa, mikä voi kertoa TIMin taustalla olevasta ohjelmointiparadigmasta.

Avainsanat: ohjelmisto, TIM, staattinen koodimetriikka, staattiset koodimetriikat, Git, Python, Flask, SonarQube, PyPI, McCabe-kompleksisuus, kognitiviinen kompleksisuus, ohjelmistotiede, Halsteadin metriikat, metodien koheesion puute, LCOM, fan-out, ohjelmiston laatu, ylläpidettävyys, ymmärrettävyys

Abstract: Static code metrics are ways to measure software code without executing it. Typically, they are used to evaluate code quality, such as understandability and maintainabiliy, but their applicability to it is somewhat questionable. The topic has been under research for

about 50 years without univocal results. Software industry uses them quite steadily to validate code, but academic research about them has given rather contradictory results. The source code of TIM, an educational web application developed in the University of Jyväskylä, was researched by the available metric tools and it turned out that their relation to the amount of Git commits in the Git log was remarkable. However, the metrics the research emphasized were somewhat different from what previous research has emphasized, which may tell about the programming paradigm used in TIM.

Keywords: software, TIM, static code metric, static code metrics, Git, Python, Flask, SonarQube, PyPI, McCabe complexity, cognitive complexity, software science, Halstead's metrics, lack of cohesion of methods, fan-out, software quality, maintainability, understandability

Kuviot

Kuvio 1. Graafinen kuvaus TIMistä. Kuviosta poiketen Caddy kommunikoi myös osin muiden konttien kuin TIMin pääkontin kanssa. 19
Kuvio 2. Tiedostojen fan-out-mittojen ja commitien määrän yhteyden kuvaava sirontakuvio. Kolme ääriesimerkkiä on korostettu omilla väreillään. Punainen ympyrä kuvaa timApp/util/error_handlers.py-tiedostoa, vihreä timApp/tim.py-tiedostoa ja sininen timApp/modules/cs/jsframe.py-tiedostoa. 49
Kuvio 3. Tiedostojen koodirivien ja commitien määrän yhteyden kuvaava sirontakuvio. Kolme ääriesimerkkiä on korostettu omilla väreillään. Punainen ympyrä kuvaa timApp/modules/cs/simcir/check/simcirtest.py-tiedostoa, sininen timApp/answer/routes.py-tiedostoa ja vihreä timApp/launch.py-tiedostoa. 51
Kuvio 4. Tiedostojen Halsteadin V-mitan ja commitien määrän yhteyden kuvaava sirontakuvio. Kolme ääriesimerkkiä on korostettu omilla väreillään. Vihreä ja sininen ympyrä kuvaavat samoja tiedostoja kuin kuviossa 3. Punainen ympyrä kuvaa timApp/plugin/userselect/action_queue.py-tiedostoa. 53
Kuvio 5. Tiedostojen Halsteadin E-mitan ja commitien määrän yhteyden kuvaava sirontakuvio. Kolme ääriesimerkkiä ovat samat kuin kuviossa 4 ja ne on korostettu samoilla väreillä. 54
Kuvio 6. Luokkien metodien koheesion puutteen (LCOM) ja commitien määrän yhteyden kuvaava sirontakuvio. Sinisellä on korostettu timApp/velp/annotations.py-tiedoston AnnotationVisibility-luokka, vihreällä timApp/document/docinfo.py-tiedoston DocInfo-luokka ja punaisella timApp/sisu/parse_display_name.py-tiedoston SisuDisplayName-luokka. 55
Kuvio 7. SonarQuben mittaamien funktioiden McCabe-kompleksisuuksien ja commitien määrän yhteyden kuvaava sirontakuvio. Punaisella on korostettu timApp/plugin/jsrunner/util.py-tiedoston save_fields-funktio, vihreällä timApp/item/routes.py-tiedoston render_doc_view-funktio ja sinisellä timApp/tim.py-tiedoston start_app-funktio. 57
Kuvio 8. Multimetricin mittaamien tiedostojen McCabe-kompleksisuuksien ja commitien määrän yhteyden kuvaava sirontakuvio. Punaisella on korostettu timApp/plugin/jsrunner/util.py, vihreällä timApp/answer/routes.py ja sinisellä timApp/document/docsettings.py. 58
Kuvio 9. SonarQuben mittaamien kognitiivisten kompleksisuuksien ja commitien määrän yhteyden kuvaava sirontakuvio. Punaisella, vihreällä ja sinisellä korostetut funktiot ovat samat kuin kuviossa 7. 60

Taulukot

Taulukko 1. Nimetyistä työkaluista, jotka Arvanitou ym. (2017) löysivät, monipuolisimmin metriikoita mittaavat heidän mukaansa Columbus, aToucan ja Tooms, mutta yksikään niistä ei tue Pythonia. He löysivät myös useita Eclipse-liitännäisiä koodimetriikoiden mittaamiseen käyttäviä tutkimuksia erittelemättä niitä tarkemmin, mutta taulukko 5 tutkii tarkemmin Eclipse-liitännäisten mahdollisuutta staattisten koodimetriikoiden mittaamiseen Python-koodista. 22
Taulukko 2. QAC:n ja Understandin ominaisuudet. 23
Taulukko 3. CPPDependin ja SonarQuben ominaisuudet. 24
Taulukko 4. Eclipse Metrics Pluginin ja SourceMonitorin ominaisuudet. 25
Taulukko 5. 29.9.2023 hakufraasilla "*python metrics*"Eclipsen virallisesta liitännäiskaupasta löytyneiden liitännäisten ominaisuudet. 26
Taulukko 6. PyPI:n metriikoita koskevia laajennoksia (PyPI:n termistöllä projekteja (P. S. Foundation 2023b)). Inheritannce-explorerista on huomattava, että se vaatii tietyn "perusluokan", josta muut luokat on peritty, manuaalista tunnistusta. Se ei myöskään mittaa varsinaisesti staattista koodimetriikkaa, koska sen käyttö vaatii tutkittavan luokan tuomista osana suoritettavaa Python-koodia, jolloin siis vähintään luokan konstruktori suoritetaan. (Wachowski 2023a) 27
Taulukko 7. SonarQuben, PyPI:n mccaben ja PyPI:n multimetricin mittaamien McCabe-kompleksisuuksien Pearsonin korrelaatiot. Kahden ensimmäisen välinen korrelaatio on funktiokohtainen, muut tiedostokohtaisia. 46
Taulukko 8. Pearsonin korrelaatiot mitattuun ohjelmistoyksikköön kohdistuneiden commitien määrän ja mitattujen metriikoiden välillä. Kognitiivinen kompleksisuus ja McCabe ovat funktiokohtaisia, LCOM luokkakohtainen ja muut metriikat tiedostokohtaisia metriikoita. 48
Taulukko 9. Spearmanin korrelaatiot mitattuun ohjelmistoyksikköön kohdistuneiden commitien määrän ja mitattujen metriikoiden välillä. Korrelaatioiden taustadata on sama kuin taulukossa 8. 48

Sisällys

1 JOHDANTO ... 1

2 KIRJALLISUUSKARTOITUS ... 2
2.1 Ohjelmiston laatu ... 2
2.2 Ylläpidettävyys ... 3
2.3 Ohjelmakoodin laadun mittaaminen ... 5
2.3.1 Halsteadin ohjelmistotiede ... 5
2.3.2 Kognitiotieteen lähestymistapa ... 5
2.4 Staattiset koodimetriikat ... 7
2.4.1 Halsteadin metriikoiden johdannaisia ... 7
2.4.2 McCabe-kompleksisuus ... 8
2.4.3 Kognitiivinen kompleksisuus ... 9
2.4.4 Oliosuuntautuneelle ohjelmistolle suunnattuja metriikoita ... 10
2.5 Staattiset koodimetriikat ja laatu ... 12
2.6 Yhteenveto ... 15

3 TUTKIMUSASETELMA ... 17
3.1 TIM ... 17
3.1.1 Palvelinkoodi ... 18
3.1.2 Tiedonhallinta ... 18
3.1.3 Kehityskäytännöt ... 18
3.2 Tutkimuskysymys ... 20
3.3 Tutkimusmenetelmät ... 20

4 AINEISTON KERUUN SUUNNITTELU ... 22
4.1 Työkalujen valinta ... 22
4.2 Kehitettävyyden ja ylläpidettävyyden mittaaminen ... 28

5 AINEISTON KERUU ... 29
5.1 Tutkimusalueen määrittely ... 29
5.2 Tiedostojen, luokkien ja funktioiden keruu ... 30
5.3 Metriikoiden keruu ... 34
5.3.1 McCabe ... 34
5.3.2 Kognitiivinen kompleksisuus ... 36
5.3.3 LCOM ... 38
5.3.4 Halsteadin metriikat, fan-out ja koodirivien määrä ... 39
5.4 Commitien laskeminen ... 40

6 AINEISTON ANALYYSI ... 44
6.1 Työkalujen yhteneväisyys ... 44
6.1.1 McCabe ... 44
6.1.2 Kognitiivinen kompleksisuus ... 46
6.2 Metriikat ja commitien määrä ... 47

6.2.1 Fan-out ... 50
6.2.2 Koodiriven määrä (LOC) ... 51
6.2.3 Halsteadin metriikat ... 53
6.2.4 Metodien koheesion puute (LCOM) ... 56
6.2.5 McCabe ... 57
6.2.6 Kognitiivinen kompleksisuus ... 60
6.3 Johtopäätökset ... 61

7 YHTEENVETO ... 64

LÄHTEET ... 67

1 Johdanto

Staattiset koodimetriikat ovat noin 50 vuotta tutkittuja ja kehitettyjä tapoja mitata ohjelmakoodista automaattisesti ominaisuuksia suorittamatta sitä. Yleensä niillä yritetään selvittää koodin ongelmallisuutta ja laatua, mutta näyttö niiden hyödyllisyydestä etenkään yksittäin käytettyinä ei ole kovin vahvaa. Tässä tutkielmassa niiden hyödyllisyyttä tutkitaan yhden avoimen lähdekoodin web-sovelluksen, TIMin, lähdekoodia ja sen kehitysprosessia tutkimalla.

TIM on Jyväskylän yliopistossa kehitetty koulutusohjelmisto, jolla voi hallinnoida vuorovaikutteisten tehtävien antamista, tekemistä, palauttamista ja arvostelua. Koska lähdekoodi on avoin, Git-versionhallinta mahdollistaa koko kehitysprosessin analysoinnin ja oppilaitoksen sisällä on helppo tavoittaa ohjelmistosta ja sen kehitysprosessista tietäviä. Siksi TIM sopii esimerkiksi käytännön ohjelmistokehityksestä metriikoita analysoitaessa. Suurin osa TIMin toiminnallisuuksista toteutuu palvelimella, ja sen merkittävimmät ohjelmointikielet ovat Python, Haskell ja JavaScript. (JYU 2023b)

Luvussa 2 käydään läpi staattisten koodimetriikoiden tutkimusta koko niiden tutkimushistorian ajalta. Luvussa 3 esitellään tarkemmin TIMiä, muotoillaan tutkimuskysymys sen ja luvun 2 perusteella ja suunnitellaan tarkemmin, miten tutkimus toteutetaan. Luvussa 4 tutkitaan saatavilla olevia työkaluja staattisten koodimetriikoiden mittaamiseen, valitaan niistä TIMin tutkimiseen soveltuvat ja eritellään, miten muuten koodin laatua on tarkoitus mitata. Luvussa 5 määritellään tarkkaan, mikä osa TIMin lähdekoodista kerättävään aineistoon sisällytetään, esitellään mitattavien ohjelmiston yksiköiden keräämiseen sekä staattisten koodimetriikoiden ja Git-commitien määrän mittaamiseen tarkoitetut Python-skriptit ja kuvataan, miten ne onnistuivat tehtävässään. Luvussa 6 tutkitaan, missä määrin samaa nimellistä metriikkaa mittaavat työkalut antoivat samat mittaustulokset sekä verrataan mitattuja metriikoita laskettuun Git-commitien määrään sekä tutkitaan tarkemmin ääritapauksia. Luvussa 6 myös eritellään, mitä johtopäätöksiä aineistosta voi tehdä. Luku 7 on tutkielman yhteenveto ja se erittelee, miten hyödylliseltä mikäkin tutkittu staattinen koodimetriikka vaikuttaa, mainitsee tutkielman puutteita sekä esittää jatkotutkimusaiheita.

2 Kirjallisuuskartoitus

2.1 Ohjelmiston laatu

ISO 9126 -standardi määrittelee ohjelmiston laaduksi "kohteen ominaisuuksien yhdistelmän, joka määrää sen kyvyn tyydyttää mainitut ja odotetut tarpeet"(*the totality of characteristics of an entity that bears on its ability to satisfy stated and implied needs*) (Nilson, Antinyan ja Gren 2019).

ISO, IEC ja IEEE antavat laadulle myös seuraavia, vaihtoehtoisia määritelmiä. Määritelmien erot ovat kuitenkin melko pieniä (Wagner 2013):

1. Se, missä määrin järjestelmä, komponentti tai prosessi täyttää määritellyt tarpeet (*The degree to which a system, component or process meets specified requirements*).

2. Tuotteen, palvelun, järjestelmän, komponentin tai prosessin kyky täyttää käyttäjän tai asiakkaan tarpeet, odotukset tai vaatimukset (*The ability of a product, service, system, component or process to meet customer or user needs, expectations or requirements*).

3. Sopivuus käyttäjän odotuksiin ja vaatimuksiin, asiakkaiden tyytyväisyys, vakaus ja olemassaolevien virheiden taso (*Conformity to user expectations, conformity to user requirements, customer satisfaction, reliability and level of defects present*).

4. Se, missä määrin joukko luontaisia ominaisuuksia täyttää vaatimukset (*The degree to which a set of inherent characteristics fulfils requirements*).

5. Se, missä määrin järjestelmä, komponentti tai prosessi täsmää käyttäjän tai asiakkaan tarpeisiin tai vaatimuksiin (*The degree to which a system, component or process meets customer or user needs or expectations*).

ISO 9126 -standardi jaottelee laadun sisäiseen ja ulkoiseen. Sisäinen laatu kuvaa kehittäjien näkökulmaa ohjelmistoon, kuten miten helppoa ohjelmistoa on jatkokehittää ja ylläpitää. Ulkoinen laatu puolestaan kuvaa ohjelmiston käyttäytymistä. Standardin on sittemmin vanhentanut ISO/IEC 25010 -standardi, joka määrittelee ohjelmiston kahdeksan päälaatutekijää luo-

kittelematta niitä sisäisiin ja ulkoisiin: toiminnallisuus (*functional suitability*), vakaus (*reliability*), tehokkuus (*performance efficiency*), käytettävyys (*usability*), turvallisuus (*security*), ylläpidettävyys (*maintainability*), siirrettävyys (*portability*) ja yhteensopivuus (*compability*) (Wagner 2013).

Toisaalta Wagner (2013) toteaa, että sekä ohjelmiston laatu että sen alakäsitteet ovat epäeksakteja käsitteitä. On esimerkiksi epäselvää, onko ylläpidettävyys sisäistä vai ulkoista laatua. Nilson, Antinyan ja Gren (2019) ja Jabangwe ym. (2015) mainitsevat sen ulkoisena laatuattribuuttina, Schnappinger, Fietzke ja Pretschner (2021) sisäisenä.

Toiminnallisuus on melko kiistattomasti todettavissa: vaadittu asia joko onnistuu ohjelmistolla tai ei. Myös vakautta voi mitata melko kiistattomilla menetelmillä, koska kyse on pohjimmiltaan todennäköisyydestä, että ohjelmisto toimii, kuten pitää. Tehokkuuttakin voi melko kiistatta numeerisesti mitata, koska siinä on kyse ohjelmiston vaatimasta ajasta ja laitteistoresursseista. (Wagner 2013)

Käytettävyys liittyy vahvasti sekä toiminnallisuuteen, vakauteen että tehokkuuteen. Toisaalta se on melko subjektiivista, koska eri käyttäjät arvostavat ohjelmistossa eri asioita. Ohjelmiston turvallisuuden arviointi on kokonaan oma, laaja tutkimusalansa, joten se sivuutetaan tässä tutkielmassa. Siirrettävyys ja yhteensopivuus ovat myös melko kiistatta todettavissa ohjelmiston ja sen käyttöympäristön ominaisuuksia tutkimalla. (Wagner 2013)

Ylläpidettävyyden mittaaminen on kuitenkin mutkikkaampaa ja siinä käytetyt menetelmät ovat kyseenalaisempia. (Wagner 2013)

2.2 Ylläpidettävyys

Ylläpito tarkoittaa ohjelmiston sovittamista ympäristönsä muuttuviin vaatimuksiin ja sen parantamista käyttöönottonsa jälkeen. Käytännössä ylläpidettävyys kuvaa, kuinka helppoa ohjelmakoodia on ymmärtää, muuttaa ja laajentaa. Ylläpidettävyydestä käytetään myös nimiä *code quality* ja *internal quality*. Muista ISO 25010-standardin päälaatutekijöistä poiketen ylläpidettävyys keskittyy kokonaan kehittäjien näkökulmaan. (Wagner 2013)

ISO 25010 -standardi mainitsee ylläpidettävyyden alikäsitteinä modulaarisuuden (*modu-*

larity), uudelleenkäytettävyyden (*reusability*), muokattavuuden (*modifiability*), analysoitavuuden (*analysability*) ja testattavuuden (*testability*) (Wagner 2013). Vanhempi ISO 9126 -standardi piti ylläpidettävyyden alikäsitteinä niistä ainoastaan kahta viimeistä, sekä muutettavuutta (*changeability*), vakautta (*stability*) ja ohjeenmukaisutta (*compliance*) (Arvanitou ym. 2017). Ylläpidettävyyden ja kehitettävyyden ero ei ole kovin selkeä etenkään jatkuvan ohjelmistotuotannon osalta, ja niiden käytännön vaatimukset ohjelmistolle ovat pitkälti samat.

Ohjelmakoodin luettavuus (*readability*) kuvaa kykyä ymmärtää koodista kaikki, mikä pelkän saatavilla olevan koodin ja käytetyn ohjelmointikielen perussyntaksin perusteella on mahdollista ymmärtää. Luettavuus on silti eri asia kuin ymmärrettävyys. Luettavassa koodissa voidaan käyttää esimerkiksi heikosti tunnettuja kirjastoja tai rajapintoja, mutta jos lukija ei niitä tunne, ei koodia voi kutsua ymmärrettäväksi (Scalabrino ym. 2019).

Ymmärrettävyys (*understandability*) mielletään laajalti omaksi laatuattribuutikseen, vaikka se liittyy vahvasti luettavuuteen ja ylläpidettävyyteen. Ymmärrettävyyden ympärille on kehittynyt oma tutkimusyhteisönsä (Arvanitou ym. 2017). Ymmärrettävä koodi mahdollistaa myös ohjelmakoodin koko tarkoituksen ymmärtämisen. Sitä voi mitata koehenkilöiden onnistumisella bugien löytämisessä ja niiden korjaamisessa. (Muñoz Barón, Wyrich ja Wagner 2020) Noin 70 % kehittäjien ajasta kuluu ohjelmakoodin ymmärtämiseen, joten sen oikea mittaaminen on arvokas taito, ja ymmärrettävyys onkin juuri ylläpidettävyyden kannalta ensisijaisen tärkeää (Scalabrino ym. 2019).

Bansiya ja Davis (2002) kehittivät oliosuuntautuneen ohjelmistosuunnitelman laadun arviointiin QMOOD-mallin, joka tekee eron käsitteiden joustavuus (*flexibility*), laajennettavuus (*extensibility*) ja ylläpidettävyys välille. Viimeisin jätettiin mallista täysin pois, koska mallin tekijöiden mukaan se koskee pelkästään jo toteutettua ohjelmistoa.

2.3 Ohjelmakoodin laadun mittaaminen

2.3.1 Halsteadin ohjelmistotiede

Idea ohjelmakoodin systemaattisesta arvioinnista syntyi 1970-luvun alussa nimellä ohjelmistotiede (englanniksi *software science*). M. H. Halstead luetteli monografissaan eri ohjelmakoodista mitattavia suureita, jotka johdetaan seuraavista neljästä perussuureesta (Shen, Conte ja Dunsmore 1983):

1. $n1$ eli uniikkien operaattorien määrä ohjelmistossa.

2. $n2$ eli uniikkien operandien määrä ohjelmistossa.

3. $N1$ eli operaattorien esiintymiskerrat yhteensä ohjelmistossa.

4. $N2$ eli operandien esiintymiskerrat yhteensä ohjelmistossa.

$N1+N2$ voidaan lyhentää N:ksi.

Operaattori tarkoittaa ohjelmointikielen symbolia tai avainsanaa, joka ilmaisee laskennallista operaatiota. Operandi tarkoittaa avainsanaa, joka ilmaisee dataa. Käsitteiden tarkka määrittely ei ole yksiselitteistä ja se riippui vahvasti ohjelmointikielestä jo 1980-luvulla. Kyseenalaista on esimerkiksi, ovatko kaksi GO TO -komentoa uniikkeja operaatoreita aina, kun ne käskevät siirtymään eri koodiriveille.

Ohjelmistotiede kehitettiin alun perin algoritmien eikä kokonaisten ohjelmistojen arviointiin, mutta sitä on sovellettu jälkimmäiseenkin (Shen, Conte ja Dunsmore 1983). Ohjelmistotiede on käsitteenä käytännössä vanhentunut, vaikka ohjelmakoodin laadun numeerista mittaamista tutkitaan yhä.

2.3.2 Kognitiotieteen lähestymistapa

Cant, Jeffery ja Henderson-Sellers (1995) totesivat, että siihenastinen ohjelmistojen mutkikkuuden tutkimus oli ollut hyvin epäyhtenäistä ja tuloksetonta. Ohjelmistojen mutkikkuutta ja laatuattribuuteista etenkin ylläpidettävyyttä mittaavia metriikoita oli julkaistu useita, mutta yksikään niistä ei heidän mukaan ole kovin hyvä eikä niiden taustalla ole kovin tarkkaa

määritelmää sille, mitä mitataan.

Cant, Jeffery ja Henderson-Sellers (1995) muotoilivat siihenastisen kognitiotieteen ja ohjelmistokehityksen tutkimuksen perusteella puhtaan teoreettisen mallin ohjelmakoodin mutkikkuudelle keskittyen ohjelmakoodin ymmärtämisen prosessiin eikä sen lopputulokseen. Malli ensinnäkin jaottelee ohjelmakoodin lohkoihin (*chunk*) tietyn, epätäydellisen säännöstön perusteella, yhdistää lohkoista suurempia kokonaisuuksia ja toteaa, että koko ohjelmiston tai tietyn lohkon mutkikkuus on lohkon ja sen yläpuolella lohkohierarkiassa olevien lohkojen mutkikkuuden ja niiden jäljitettävyyden summa.

Cant, Jeffery ja Henderson-Sellers (1995) esittävät yksinkertaisia laskukaavoja sekä mutkikkuuden että jäljitettävyyden arviointiin, mutta ne eivät ole kovin tyhjentäviä. Lisäksi arvioinnissa huomioidaan koodin sopivuus yleisiin käytänteisiin, kuten muuttujannimien johdonmukaisuus ja kommenttien määrä ja sisältö. Kirjoittajat korostavat, että artikkeli on vain ensiaskel uuden mallin kehittämiseksi. Tuloksena saatu malli keräsi hiukan empiiristä tukea, mutta kirjoittajat korostavat, että malli kaipaa vielä paljon lisää jatkokehitystä ja validointia.

Cantin ym. malli ei vaikuta saaneen paljoa huomiota tai jatkotutkimusta, mutta sille on kehitetty kilpailijoita. Duran, Sorva ja Leite (2018) ovat kehittäneet kognitiotieteen hierarkkisen kompleksisuuden malliin (*Model of Hierarchical Complexity, MHC*) perustuvan tavan arvioida ohjelmien mutkikkuutta. Malli perustuu kolmeen yleisluontoiseen sääntöön, joille on löytynyt empiiristä tukea monelta alalta. Duran, Sorva ja Leite (2018) kehittivät sen pohjalta kognitiivisen kompleksisuuden (*Cognitive Complexity of Computer Programs, CCCP*) mallin, joka mittaa ohjelmakoodista kaksi suuretta: käsitteiden interaktiivisuuden ja käsitteiden syvyyden. Esimerkillä osoitetaan, että ainakin edellistä voi manipuloida ilman, että ohjelman varsinainen merkitys muuttuu. Duran, Sorva ja Leite (2018) keskittyivät ohjelmoinnin opetukseen ja yksinkertaisten, opintomateriaaliksi suunnattujen ohjelmien mutkikkuuteen. Duran, Sorva ja Leite (2018) eivät testanneet malliaan empiirisesti, ja kuvattujen suureiden tarkka määrittely ja työkalut niiden mittaamiseksi automaattisesti vaativat lisätyötä.

2.4 Staattiset koodimetriikat

Staattiset koodimetriikat (*static code metrics*), joihin Halsteadin suureetkin kuuluvat, mittaavat ohjelmakoodista suorittamatta sitä laskennallisesti ominaisuuksia, jotka saattavat liityä ohjelmakoodin laatuun ja muihin käytännön piirteisiin. Niiden etu on yksiselitteisyys ja mahdollisuus mitata niitä melko helposti automaattisesti. Staattiset metriikat antavat uutta näkökulmaa ohjelmakoodin arviointiin, ja tavallisesta poikkeavat metriikoiden antamat mittaustulokset voivat kieliä kohdista, jotka vaativat muita tarkempaa huomiota. (Gray ym. 2009) Esimerkiksi monet Yhdysvaltain hallituksen toimeksisaajat käyttävät niitä ohjelmistokomponenttien laadun arviointiin (Menzies ym. 2007).

Staattisista koodimetriikoista käytetty nimi ei ole vakiintunut. Esimerkiksi Wagner (2013) ei mainitse termiä *static code metrics*, mutta käyttää termiä staattiset analyysityökalut (*static analysis tools*) ohjelmistoista, jotka mittaavat ohjelmakoodista ominaisuuksia suorittamatta sitä. Tässä pro gradu -tutkielmassa käytetään kuitenkin nimeä staattiset koodimetriikat.

Tunnettuja staattisia koodimetriikoita ovat Halsteadin 1970-luvulla kuvaamien operaattorien ja operandien määrien ja niiden johdannaisten lisäksi esimerkiksi koodirivien määrä (*Lines of code, LOC*), McCabe-kompleksisuus (McCabe 1976), eri mahdollisten datavuoiden määrää mittaavat *fan-in*- ja *fan-out*-mitat (Henry ja Kafura 1981) sekä oliosuuntautuneelle ohjelmistolle suunnatut luokan koheesion puute (*Lack of Cohesion of Methods, LCOM*) ja luokan perintäpuun syvyys (*Depth of Inheritance Tree*) (Chidamber ja Kemerer 1994).

Koska CCCP-mallin tai Cantin ym. mallin mukaisia automaattisia mittaustyökaluja ei ole, malleja ei voi varsinaisesti kutsua staattisiksi koodimetriikoiksi. Toisaalta Duran, Sorva ja Leite (2018) korostavat, että heidän mallinsa mittaa koodia syvällisemmin kuin esimerkiksi McCabe-kompleksisuus.

2.4.1 Halsteadin metriikoiden johdannaisia

Halsteadin perusmetriikat *n1*, *n2*, *N1* ja *N2* ovat sinänsä staattisia koodimetriikoita, mutta myös useat niistä johdettavat metriikat ovat saaneet tunnustusta. Shen, Conte ja Dunsmore (1983) totesivat tutkimuksessaan, että eniten empiiristä näyttöä keränneitä ohjelmistotieteen teorioita on pituusyhtälö (*length equation*):

$$N = N1 + N2 \approx n1 * log_2 n1 + n2 * log_2 n2$$

Ohjelmistotieteen empiiristä näyttöä keränneitä teorioita on myös, että ohjelmiston ymmärrettävyyttä ja ylläpidettävyyttä voi mitata E:ksi kutsutulla muuttujalla, joka määritellään

$$E = \frac{V^2}{V*}$$

missä

$$V = N * log_2 n$$

ja $V*$ on pienin mahdollinen V sellaiselle ohjelmalle, jolla on samat ominaisuudet kuin mitattavalla ohjelmalla. Shen, Conte ja Dunsmore (1983) esittävät kaksi yhtälöä $V*$:n arviointiin pelkkien perusmetriikoiden avulla, mutta toinen niistä olettaa, että mitattava ohjelma mielletään pelkäksi tavaksi muodostaa paluuarvoja syötteen perusteella.

Huomattavaa on myös, että koodirivien määrästä (LOC) voidaan regressioanalyysillä arvioida ohjelmiston kehittämisen, ymmärtämisen ja ylläpidon vaatimaa aikaa:

$$T = a * LOC^b + c$$

missä 1970–80-luvuilla tehdyissä analyyseissä b sai pienimmillään arvon 0,91 ja korkeimmillaan 1,83 (Shen, Conte ja Dunsmore 1983).

2.4.2 McCabe-kompleksisuus

Staattisista koodimetriikoista tunnetuimpia ja vanhimpia on Halsteadin metriikoiden ohella McCabe-kompleksisuus (*McCabe's cyclomatic complexity*) (Cant, Jeffery ja Henderson-Sellers 1995) (McCabe 1976). Se on suunniteltu yksittäisten funktioiden tai funktioiden joukkojen mutkikkuuden arviointiin. Metriikka muotoilee funktioista suunnatun graafin G,

jossa solmut vastaavat tietyn funktion aina tietyssä järjestyksessä suoriettavaa osuutta. Solmusta A on kaari solmuun B, jos ja vain jos ne kuuluvat samaan funktioon ja ohjelman suorituksen on mahdollista siirtyä A:sta suoraan B:hen.

McCabe-kompleksisuus lasketaan:

$$v(G) = e - n + 2p$$

missä e on kaarten määrä, n solmujen määrä ja p yhdistettyjen komponenttien määrä, joka vastaa käytännössä funktioiden määrää (McCabe 1976).

McCabe-kompleksisuuden hyödyllisyydestä ei vielä 1990-luvun puolivälissä ollut paljoa empiiristä näyttöä ainakaan Cantin ym. mukaan (Cant, Jeffery ja Henderson-Sellers 1995).

2.4.3 Kognitiivinen kompleksisuus

Tunnetuimpia staattisia koodimetriikoita on myös SonarSourcen 2017 julkaisema kognitiivinen kompleksisuus (*cognitive complexity*) (Muñoz Barón, Wyrich ja Wagner 2020). Se muistuttaa paljon McCabe-kompleksisuutta (McCabe 1976), mutta pyrkii korjaamaan sen heikoimpina pidettyjä piirteitä. Se perustuu tarkan, matemaattisen teorian sijaan joukkoon nyrkkisääntöjä, jotka on suunniteltu nykyaikaisille ohjelmointikielille. Kolme perussääntöä ovat (Campbell 2021):

1. **Sivuuta lyhenteet**. Jos usean lauseen koodi on lyhennettävissä yhdeksi lauseeksi, lyhennetty muoto ei nosta kompleksisuutta yhtä paljon kuin "lyhentämätön".

2. **Nosta yhdellä jokaista lineaarisen vuon rikkomista kohti**. Lineaarisen vuon rikkovat `if`- ja `catch`-lauseet, silmukat ja `switch-case`-rakenteet. `else`-, `try`- ja `finally`-lauseet eivät nosta kompleksisuutta, koska niihin liittyy aina `if`- tai `catch`-lause, jonka yhteydessä rakenteen "kognitiivinen hinta"on jo maksettu. Loogiset operaattorit nostavat kompleksisutta tiettyjen, melko yksinkertaisten sääntöjen perusteella. Rekursio nostaa kompleksisuutta yhdellä jokaista rekursiosilmukkaan kuuluvaa funktiokutsua kohti. Myös `break`- ja `continue`-lauseet nostavat kompleksisuutta yhdellä, joskaan kesken funktion ilmenevä `return`-lause ei nosta.

3. **Nosta yhdellä jokaista vuon rikkomista, johon vuon rikkominen on sisennetty, kohti**. Jos siis kohdassa 2 kuvattu lause on toisen kuvausta vastaavan lauseen aiheuttaman ohjelmalohkon sisällä, kompleksisuus nousee kahdella eikä yhdellä. Esimerkiksi kaksi sisäkkäistä `for`-silmukkaa nostavat kompleksisuutta yhteensä kolmella, mutta peräkkäin ne nostavat sitä vain kahdella.

Toisin kuin McCabe-kompleksisuus, kognitiivinen kompleksisuus ei nouse pelkällä funktion olemassaololla. Esimerkiksi pelkän kaksi lukua yhteen laskevan funktion McCabe-kompleksisuus on yksi, mutta kognitiivinen kompleksisuus nolla. Esimerkiksi luokan kognitiivinen kompleksisuus voidaan siis johdonmukaisesti määritellä laskemalla metodien kognitiiviset kompleksisuudet yhteen, mutta McCabe-kompleksisuuksien summasta ei voi päätellä, onko luokka vain joukko `get`- ja `set`-funktioita vai sisältääkö se muutaman todella mutkikkaan funktion. (Campbell 2021)

2.4.4 Oliosuuntautuneelle ohjelmistolle suunnattuja metriikoita

Chidamber ja Kemerer (1994) esittelivät kuusi uutta oliosuuntautuneelle ohjelmistolle suunnattua koodimetriikkaa. Näistä eniten huomiota ovat herättäneet (Arvanitou ym. 2017) metodien koheesion puute (*Lack of Cohesion in Methods, LCOM*) ja luokan perintäpuun syvyys (*Depth of Inheritance Tree*). Jälkimmäinen kertoo pisimmän perintäketjun pituuden, joka luokalla on. Mitä pitempi ketju on, sitä enemmän perittyä ja siten ymmärrystä vaativaa sisältöä luokalla on.

Metodien koheesion puute kuvaa luokkakohtaisesti, missä määrin sen eri metodit käsittelevät samoja instanssikohtaisia attribuutteja. Olkoon luokan metodien lukumäärä n, ja $I_1, I_2, ..., I_n$ luokan kunkin metodin käsittelemien instanssikohtaisten attribuuttien joukko. Alkuperäinen kaava mittarille on:

$$LCOM = \begin{cases} |P| - |Q|, & \text{jos } |P| > |Q| \\ 0 & \text{muuten} \end{cases}$$

missä

$$P = \begin{cases} \emptyset, & \text{jos } I_1 = I_2 = \ldots = I_n = \emptyset \\ \{\{I_i, I_j\} | I_i \cap I_j = \emptyset\} & \text{muuten.} \end{cases}$$

ja

$$Q = \{\{I_i, I_j\} | I_i \cap I_j \neq \emptyset\}$$

Mittari siis kertoo, paljonko luokassa on enemmän sellaisia metodipareja, jotka eivät käsittele samoja instanssikohtaisia attribuutteja kuin sellaisia, jotka käsittelevät vähintään yhtä samaa. Jos edellisiä on vähemmän kuin jälkimmäisiä tai yksikään metodi ei käsittele instanssikohtaisia attribuutteja, mittarin lukema on 0. Korkea *LCOM* kielii siitä, ettei luokka käsittele yhtenäistä kokonaisuutta eli sen koheesio on heikko. Luokan koheesio on oliosuuntautuneessa ohjelmoinnissa toivottua, koska se edistää kapselointia (Chidamber ja Kemerer 1994).

Chidamber ja Kemerer (1994) esittelivät myös neljä muuta oliosuuntautuneelle ohjelmistolle suunnattua koodimetriikkaa, jotka ovat melko yksinkertaisesti ja yksiselitteisesti laskettavissa:

- painotettu metodien summa luokassa (*Weighed Methods per Class, WMC*) eli luokan metodien kompleksisuuksien summa (Kompleksisuuden mittauskeinoon ei metriikka ota kantaa),
- lasten lukumäärä (*Number of Children, NOC*) eli niiden luokkien määrä, jotka suoraan perivät mitattavan luokan,
- kohteiden välinen paritus (*Coupling between Object Classes, CBO*) eli niiden parien määrä, joihin luokka kuuluu, kun luokat muodostavat parin, jos toinen niistä käyttää toisen metodeja tai attribuutteja ja
- luokan vaste (*Response for a Class, RFC*) eli luokan metodien ja niiden kutsumisen seurauksena kutsuttavien metodien yhteismäärä.

Nämä neljä metriikkaa eivät kuitenkaan ole herättäneet yhtä paljoa tutkimusta (Arvanitou ym. 2017).

2.5 Staattiset koodimetriikat ja laatu

Staattisten koodimetriikoiden käytännön merkityksen tutkimus ohjelmistoteollisuudessa jatkuu yhä. Arvanitou ym. (2017) löysivät systemaattisessa kirjallisuuskartoituksessaan 154 alan arvostetuimmissa tieteellisisssä julkaisuissa julkaistua tutkimusta, joista 136 käsittelee metriikoita (*"software metrics"*). 136 metriikoita käsittelevästä tutkimuksesta 112 antaa jonkinlaista empiiristä validaatiota jotakin metriikkaa koskien. Osa tutkimuksista validoi metriikoita vain teoreettisesti. Arvanitou ym. (2017) luokittelevat metriikoiden saamat empiiriset validaatiot kuuteen tasoon, jotka Alves ym. (2010) määrittelivät:

1. Ei validaatiota (*No evidence*)

2. Demonstraatioista tai "leluesimerkeistä"tunnistettu validaatio (*Evidence obtained from demonstration or working out toy examples*)

3. Havainnoista tai asiantuntijamielipiteistä tunnistettu validaatio (*Evidence obtained from expert opinions or observations*)

4. Akateemisista tutkimuksista tunnistettu validaatio (*Evidence obtained from academic studies, e.g., controlled lab experiments*)

5. Teollisuuden tutkimuksista tunnistettu validaatio (*Evidence obtained from industrial studies, e.g., causal case studies*)

6. Teollisesta näytöstä tunnistettu validaatio (*Evidence obtained from industrial practice*), joka tarkoittaa, että jokin ohjelmistotuotannon organisaatio käyttää metriikkaa rutiininomaisesti (Alves ym. 2010).

Hierarkia perustuu tutkimusasetelmien hierarkiaan, jonka Kitchenham (2004) johtivat kahdesta muusta hierarkiasta: Yorkin yliopiston arviointi- ja levityskeskus -terveystutkimuslaitoksen (*Centre for Reviews and Dissemination, CRD*) näytön vahvuuden hierarkiasta ja Australian kansallisen terveys- ja lääketiedetutkimusvaltuuston (*Australian National Health and Medical Research Council, ANHMRC*) käyttämästä hierarkiasta. Ne on tarkoitettu lääketieteellisten tutkimusten arviointiin, mutta Kitchenhamin ym. mukaan niitä voi soveltaa myös ohjelmistotuotannon tutkimuksen arviointiin. Kitchenhamin ym. hierarkian tasot ovat:

1. Vähintään yhdestä hyvin suunnitellusta, satunnaistetusta ja kontrolloidusta kokeesta saatu näyttö (*Evidence obtained from at least one properly-designed randomised controlled trial*)

2. Hyvin suunnitelluista, pseudosatunnaistetuista, kontrolloiduista kokeista saatu näyttö (*Evidence obtained from well-designed pseudo-randomised controlled trials (i.e. non-random allocation to treatment)*).

3. Vertailevista tutkimuksista saatu näyttö (*Evidence obtained from comparative studies*).

4. Tekoympäristössä tehdystä kokeesta tai kontrolloimattomasta tapaustutkimuksesta saatu näyttö (*Evidence obtained from a (quasi-)randomised experiment performed in an artificial setting* tai *evidence obtained from case series, either post-test or pre-test/post-test*).

5. Teoriaan tai konsensukseen perustuvasta asiantuntijamielipiteestä saatu näyttö (*Evidence obtained from expert opinion based on theory or consensus*).

Kitchenham (2004) jakaa lisäksi tason 3 kahteen ja tason 4 kolmeen eri alitasoon.

Alves ym. (2010) huomauttavat kuitenkin, että ohjelmistokehityksen tutkimuksessa satunnaistetut, kontrolloidut kokeet ovat hankalia ja siksi harvinaisia. Siksi he korvaavat Kitchenhamin ym. tasot 1 ja 2 "teollisen käytännön"tasolla. Lisäksi heikkoon päähän he nimeävät kaksi uutta tasoa. Kuitenkin tasoa 2 eli "demonstraatioista tai leluesimerkeistä tunnistettua validaatiota"Alves ym. (2010) eivät tarkemmin määrittele.

On myös kyseenalaista, onko tietyn menetelmän omaksuminen ohjelmistoteollisuudessa vahvempi näyttö sen validiudesta kuin mikä tahansa akateeminen tutkimus. Alves ym. (2010) vain uskovat, että "*daily engineering practice shows a convincing proof that something works, so we rank it the strongest in the hierarchy*".

Vahvimmalla, kuudennella tasolla vähintään kahdessa kirjallisuuskartoituksen tutkimuksessa validoidut metriikat ovat perintäpuun syvyys, lasten lukumäärä, luokan vaste, metodien koheesion puute ja painotettu metodien summa luokassa (Arvanitou ym. 2017).

Halsteadin muuttujista N, V, $N1$ ja $N2$ sekä McCabe-kompleksisuus on validoitu useissa tutkimuksissa tasolla 4. (Arvanitou ym. 2017). Toisaalta vielä tason 4 validointi vaikuttaa melko kyseenalaiselta. Eräs kartoituksessa löydetty tutkimus kehitti mallin, joka käyttää Halsteadin

metriikoita yhtenä useista mittareista, joilla oliosuuntautuneen ohjelmiston kompleksisuus mitataan luokkakohtaisesti. Malli validoitiin sillä, että erään oikeassa käytössä olevan ohjelmiston ensimmäinen versio oli sen mukaan kompleksisempi kuin toinen, ja toisen version tarkoitus oli parantaa koodin laatua ja ohjelmiston toiminnallisuutta ensimmäiseen nähden. Lisäksi kehittäjien henkilökohtaiset mielipiteet koodin kompleksisuudesta olivat yhteydessä mallin tuloksiin. (Coskun ja Grabowski 2001) Arvanitou ym. (2017) eivät mainitse kognitiivista kompleksisutta, joskin se julkaistiin vasta samana vuonna kuin heidän artikkelinsa.

Nilson, Antinyan ja Gren (2019) löysivät kirjallisuuskartoituksessaan 292 tutkimusta, jotka käsittelevät koodimetriikoiden yhteyttä sisäiseen laatuun. Ne tutkivat yhteensä 30 metriikkaa, joista 12 ne jokseenkin validoivat eli niillä todettiin olevan jonkinlainen yhteys ohjelmiston laatuun. Metriikoita ovat esimerkiksi perintäpuun syvyys, luokan vaste, metodien koheesion puute, fan-in ja fan-out sekä kohteiden välinen paritus. Nilson, Antinyan ja Gren (2019) eivät kuitenkaan jakaneet validaatiota eri tasoihin.

Muñoz Barón, Wyrich ja Wagner (2020) osoittavat, että kognitiivisella kompleksisuudella on selvä yhteys koodin ymmärrettävyyteen, kun ymmärrettävyyttä mitataan opiskelijoiden ja ohjelmistokehityksen ammattilaisten subjektiivisella arviolla ja ohjelmakoodin ymmärrettävyyttä mittaavissa kokeissa annettuihin vastauksiin kuluneella ajalla. Toisaalta kun ymmärrettävyyttä mitattiin koevastausten oikeellisuudella ja koehenkilöiden aivokuvannuksella heidän lukiessa koodia, yhteyttä ei havaittu. Lisäksi koodin ymmärrettävyys on vain osa koodin laatua. Myös tutkitut koodin ymmärrettävyysmittarit ovat kyseenalaisia, koska ne antavat ristiriitaisia tuloksia.

Scalabrino ym. (2019) osoittivat, että 121 tunnistetusta koodimetriikasta yhdelläkään ei ollut merkittävää yhteyttä kymmenestä eri ohjelmistosta otettujen 30-70 rivin Java-metodien ymmärrettävyyteen. Metriikoiden joukossa oli myös ei-teknisiä ominaisuuksia, kuten kommenttien sisältöä, muuttujannimiä ja sisennyksiä, mittaavia metriikoita. Ymmärrettävyyttä mitattiin opiskelijoiden ja ammattikehittäjien metodien ymmärtämistä mittaaviin kokeisin käyttämällä ajalla ja heidän antamilla vastauksilla kysymyksiin metodien toiminnasta. Kokeessa huomattiin myös, että kehittäjänsisäiset erot koetuloksissa olivat hiukan pienemmät kuin metodinsisäiset. Erot koetuloksissa selittyivät siis pikemminkin kehittäjien kuin metodien eroilla. Scalabrino ym. (2019) huomauttavat myös, että kaikki käytetyt metriikat ovat

melko pinnallisia. Tekoälyä hyödyntämällä voi hyvinkin olla mahdollista arvioida koodin ymmärrettävyyttä tarkemmin.

Schnappinger, Fietzke ja Pretschner (2021) osoittivat, että lähinnä todella yksinkertaiset metriikat, kuten koodirivien määrä ja luokan suurin metodikoko, ennustavat vahvasti koodin ylläpidettävyyttä, kun ennusteita validoidaan vertaamalla asiantuntija-arvioihin. Toisaalta heidän tutkimuksessaan metriikoita ei sellaisenaan tutkittu vaan ne toimivat koneoppivan mallin syötteenä. Ylläpidettävyyskin on vain osa laatua.

Vaikuttaa muutenkin siltä, että yllättävän yksinkertaiset mittarit, kuten koodirivien määrä (Lines of Code, *LOC*), kertovat suurimman osan koodin alttiudesta ongelmille. Muita ongelmista kieliviä mittareita ovat esimerkiksi luokan vaste ja kohteiden välinen paritus. Alkuperäisellä McCabe-kompleksisuudellakin voi olla jonkinlainen yhteys ongelmiin. (Jabangwe ym. 2015)

Toisaalta (Lincke, Lundberg ja Löwe 2008) osoittavat, että samaa nimellistä metriikkaa mittavat työkalut voivat antaa samalle ohjelmakoodille eri tuloksia. (Jabangwe ym. 2015) toteavatkin, että metriikan validius on siis eri asia kuin tietyn työkalun antaman metriikkatuloksen validius.

2.6 Yhteenveto

Staattisia koodimetriikoita ja niiden yhteyttä ohjelmiston laatuun on tutkittu noin 50 vuotta, ja jotkin tutkimukset antavat viitteitä yhteydestä, mutta näyttö ei ole kovin vahvaa. Vielä 1990-luvun puolivälissä tutkimus oli hyvin tuloksetonta. Esimerkiksi kompleksisuutta mittaavia metriikoita oli kehitetty huomattava määrä, mutta mittausten kohdetta ei ollut usein tarkkaan määritelty ja jokaista positiivista validaatiota kohti oli negatiivinen validaatio. (Cant, Jeffery ja Henderson-Sellers 1995)

Tutkimus on siitä huolimatta jatkunut. Lähteistä, joita Arvanitou ym. (2017) käyttivät metriikoiden validointia koskevassa systemaattisessa kirjallisuuskartoituksessaan, suurin osa on 2000-luvulta. He toteavat systemaattisen kirjallisuuskartoituksensa perusteella, että monia oliosuuntautuneelle ohjelmistolle suunnattuja metriikoita käytetään teollisuudessa ja siksi

he pitävät niitä parhaalla mahdollisella tavalla validoituina. Taustalla on kuitenkin oletus, että teollisuus erehtyy selvästi harvemmin kuin akateeminen tutkimus, minkä voi kyseenalaistaa. Osa kartoituksessa löydetyistä tutkimustuloksista vaikuttaa myös melko itsestäänselviltä. Esimerkiksi koodirivien määrän huomattiin olevan yhteydessä ylläpidettävyyteen, ymmärrettävyyteen ja testattavuuteen monessa teollisuuden tutkimuksessa.

Samanlaiseen tulokseen päätyivät Nilson, Antinyan ja Gren (2019) systemaattisessa kirjallisuuskatsauksessaan. Etsiessään empiirisiä tutkimuksia koodimetriikoiden yhteydestä sisäiseen laatuun he löysivät 292 relevanttia tutkimusta, jotka käsittelivät 30 staattista koodimetriikkaa. Jokseenkin validoituja niistä oli heidän mukaan 12.

Tutkimusta haittaa, että laatu on epäeksakti käsite ja sen mittaaminen melko subjektiivista. Erityisen vaikeaa on ylläpidettävyyden mittaaminen. Esimerkiksi ymmärrettävyyden, jota voidaan pitää ylläpidettävyyden alikäsitteenä (Wagner 2013), mittaamiseen käytetyt menetelmät antavat hyvin ristiriitaisia tuloksia (Muñoz Barón, Wyrich ja Wagner 2020).

3 Tutkimusasetelma

3.1 TIM

TIM (lyhenne sanoista *The Interactive Material*) on Jyväskylän yliopistossa kehitetty web-sovellus vuorovaikutteisten opintomateriaalien luomiseen ja jakamiseen ja tehtävien tekemiseen, palauttamiseen ja arvostelemiseen. TIMin näyttämiä ja sillä tallenettavia sivuja kutsutaan dokumenteiksi, jotka jakautuvat lohkoiksi. Dokumenttiin voidaan upottaa myös erilaisia plugineja, kuten taulukoita ja vuorovaikutteista sisältöä. (JYU 2023b)

TIMin keskeisin toimintalogiikka toteutuu palvelimella, jonka toiminnallisuudet on jaettu Docker-kontteihin eristettyihin kontteihin. Keskeisin kontti on TIMin pääkontti, ja useimmat muut kontit kommunikoivat ainoastaan sen kautta (JYU 2023c). Muut kontit voidaan jakaa karkeasti neljään ryhmään (JYU 2023c):

- **Plugin-kontit**, joilla käsitellään osa plugineista. Esimerkiksi automaattisesti tarkistettavat, matemaattiset kysymys-vastaustyppiset tehtävät tarjoava stack-plugin on toteutettu omassa Docker-kontissaan. Omina konteinaan toteutetut pluginit palauttavat pluginin HTML-koodin lisäksi mahdollisten CSS- ja JavaScript-tiedostojen osoitteet TIMin pääkontille (JYU 2023b). Osa plugineista, kuten kalenteri, on toteutettu TIMin pääkontin sisällä.
- **Ulkoiset kontit**, joilla tarjotaan pääkontille muunlaisia toiminnallisuuksia, kuten tietokanta, välimuisti ja työkaluja matemaattiseen mallinnukseen. Esimerkiksi Dumbo-kontti muuntaa Pandoc-työkalua käyttäen dokumenttien ja pluginien Markdown-sisällön HTML-merkkijonoksi. Osittain HTML-koodi generoidaan Jinja-muoteilla ja osin Angularilla selaimessa, mutta varsinaisten dokumenttien sisältö generoidaan Dumbolla (JYU 2023a). Toinen esimerkki on Caddy, joka vastaanottaa selaimelta tulevat pyynnöt ja välittää suurimman osan niistä TIMin pääkontille ja vastaukset niihin takaisin selaimelle. Osan pyynnöistä Caddy käsittelee itse ja osan se välittää suoraan ne käsittelevään konttiin.
- **Ajonaikaiset kontit**, joita muut kontit voivat luoda tilapäisesti järjestelmän suorituksen ajaksi.

- **Testikontit**, joita käytetään vain automaatiotestaukseen.

Selaimessa suoritettava koodi on enimmäkseen TypeScriptiä (JYU 2023a). Kuvio 1 kuvaa TIMin rakennetta graafisesti.

3.1.1 Palvelinkoodi

TIMin pääkontti käyttää Flask-sovelluskehystä, ja pääkontti jaottelee sovelluksen eri toiminnallisuuksia käsitteleviin blueprinteihin (*blueprint*). Käytännössä blueprintit jaottelevat sovelluksen kokonaisuuksiin, joista jokaista voi käsitellä omana Flask-sovelluksenaan ja joihin reititetään HTTP-kutsut tiettyn joukkoon URL-osoitteita. TIM-sivut kuvaava HTML-koodi generoidaan dynaamisilta osiltaan osin palvelimella Jinja-muottien avulla, mutta kehittäjäohjeet kehottavat suosimaan HTML:n luontia selaimessa (JYU 2023b).

Keskeinen blueprint TIMissä on view_page, joka hallinnoi dokumenttien näyttämistä ja luomista. Esimerkiksi dokumentin näyttämiseen vaadittu HTTP GET -kutsu tulee tehdä /view/<DOKUMENTIN POLKU>-polkuun ja dokumentin luova HTTP POST -kutsu /createItem-polkuun. Dokumenttien sisällön muuttamista koskevat HTTP-kutsut käsittelee edit_page-blueprint. (JYU 2023a)

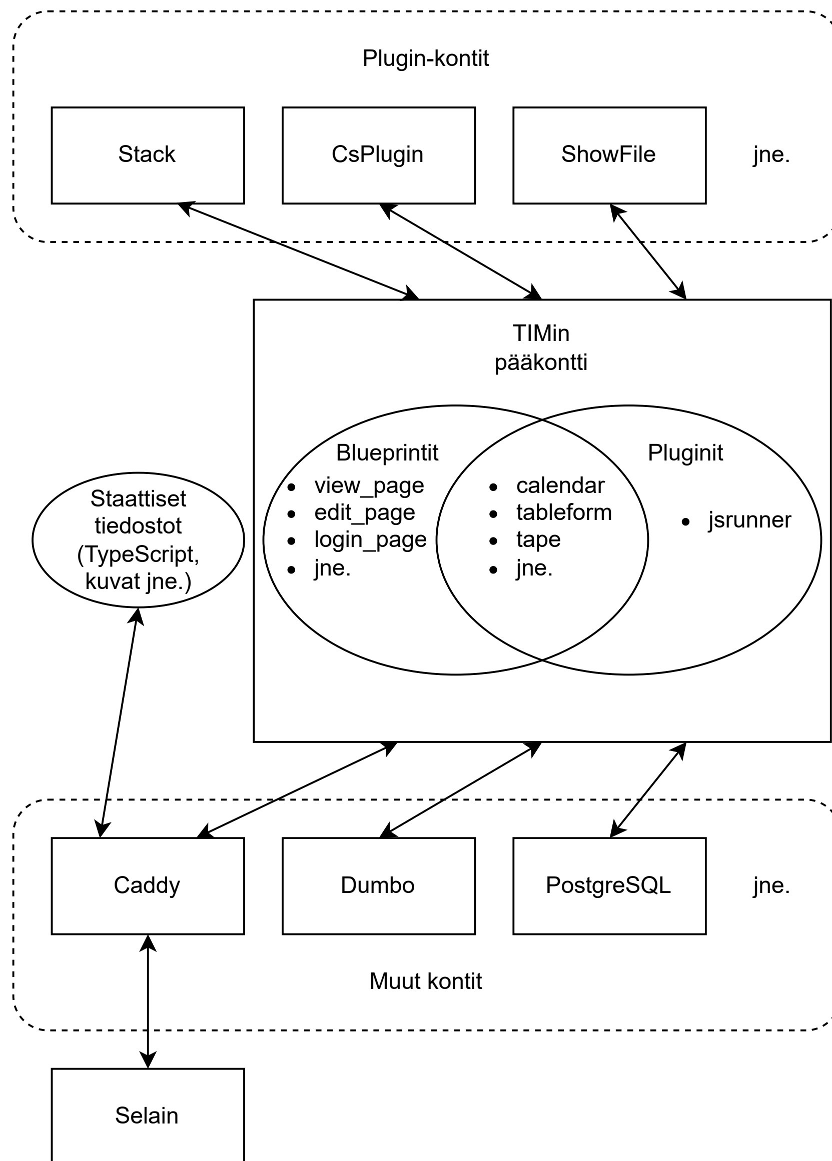
3.1.2 Tiedonhallinta

TIMin dokumenttisisältö tallennetaan raakateksti- ja JSON-tiedostihin. Dokumentteja koskeva metatieto tallennetaan versiokohtaisesti erillisiin raakatekstitiedostoihin, joissa luetellaan dokumenttiversion lohkojen tunnukset. Lohkojen sisältö puolestaan tallennetaan JSON-tiedostoina versiokohtaisesti. (JYU 2023a)

Muu TIMin hallinnoima data tallennetaan PostgreSQL-tietokantaan Pythonin SQLAlchemy-työkalun välityksellä (JYU 2023b).

3.1.3 Kehityskäytännöt

TIMin Pythonilla toteutetun palvelinohjelmiston lähdekoodin sisältävään GitHub-säiliöön on 25. lokakuuta 2023 mennessä tehnyt muutoksia 72 kehittäjää, jos kehittäjät yksilölliste-



Kuvio 1: Graafinen kuvaus TIMistä. Kuviosta poiketen Caddy kommunikoi myös osin muiden konttien kuin TIMin pääkontin kanssa.

tään heidän itse commitien yhteydessä ilmoittamiensa etu- ja sukunimen mukaan. Edeltäneen vuoden aikana muutoksia on kuitenkin tehnyt heistä vain 12, ja heistä viisi on tehnyt 85 prosenttia kyseisen ajan commiteista. Yhteensä commiteja on tehty TIMin päähaaraan 19422, joista vanhimmat ovat toukokuulta 2014. Päähaaraa muuttavia vetopyyntöjä on 25. lokakuuta 2023 edeltäneen kuukauden aikana yhdistetty yhdeksän, mikä kuvaa, miten usein tuotantoon julkaistaan uusia muutoksia. (JYU 2023a) Toisaalta "pieniä muutoksia ja bugikorjauksia"voi tehdä suoraan päähaaraan (*TIMin koodikäytänteet* 2023).

TIMin kehittäjäohjeet käskevät kehittäjiä käyttämään PEP8-tyyliohjeistusta Python-koodissa sekä formatoimaan Python-koodin Black-työkalulla ennen commitin tekemistä. Myös mypy-tyypintarkistustyökalua on käytettävä. Toisaalta koska käytänteitä on päivitetty TIMin elinkaaren varrella, vanhemmat osat TIMin koodista eivät uusimpia käytänteitä noudata. (*TIMin koodikäytänteet* 2023) Commitien käyttöön käytänteet eivät ota kantaa muuten kuin commit-viestien sekä päähaaraan puskemisen ja vetopyyntöjen osalta.

3.2 Tutkimuskysymys

Tutkimus on tapaustutkimus, jossa on tavoitteena soveltaa tunnettuja staattisia koodimetriikoita TIMin lähdekoodiin ja verrata sen tuloksia TIMin kehityskulkuun. TIM ei ole kovin erityinen web-sovellus millään tapaa, joten siitä saadut tulokset ovat odotettavasti jossain määrin yleistettävissä ainakin muuhun web-kehitykseen.

Tutkimusalueen selkeyttämiseksi ja staattisten koodimetriikoiden mittaamisen yhdenmukaistamiseksi tutkimusalue rajataan Pythonilla toteutettuun palvelinohjelmistoon.

Tutkimuskysymyksen voi muotoilla: *Miten TIMin Pythonilla toteutettu palvelinohjelmisto ja sen kehitysprosessi tukevat väitettä, että staattiset koodimetriikat ja ohjelmakoodiin kohdistuneiden muutosten tarve korreloivat keskenään?*

3.3 Tutkimusmenetelmät

Tutkimuksen ensimmäinen vaihe on staattisten koodimetriikoiden ja niitä mittaavien työkalujen etsintä. Nilson, Antinyan ja Gren (2019) löysivät vapaasti Internetistä hakemalla

130 työkalua, jotka väittävät mittaavansa ohjelmistoja, mutta kattavuutensa ja saatavuutensa perusteella niistä vain kuusi valikoitui tutkimukseen: QA-C, Understand, CPPDepend, SonarQube, Eclipse m. Plugin ja SourceMonitor. On huomioitava, että artikkeli on jo muutaman vuoden vanha. Lähdeviittausten kautta on odotettavaa löytää lisää mahdollisia metriikoita ja työkaluja. Arvanitou ym. (2017) etsivät systemaattisessa kirjallisuuskatsauksessaan koodimetriikoita ja niitä mittaavia työkaluja systemaattisen kirjallisuuskatsauksen avulla. Tarkkaa kirjallisuuskatsauksessa löydettyjen artikkelien kuvaamien metriikoiden määrää artikkeli ei mainitse, mutta niiden mittaamiseen on niissä kuvattu 19 nimettyä työkalua. Suurimmalla osalla artikkeleissa kuvatuista työkaluista ei ole varsinaista nimeä, ja ne ovat tavallisesti akateemiseen käyttöön suunnattuja tai vain prototyyppejä. Seuraavana vaiheena on analysoida löydetyillä työkaluilla TIMin lähdekoodia.

Koska TIMin lähdekoodia ylläpidetään Git-säiliössä, on myös mahdollista tutkia tiettyyn osaan lähdekoodia kohdistuneita commiteja. Se onnistuu `git log –no-merges -L[ENSIMMÄINEN RIVI],[TOINEN RIVI]:[TIEDOSTON NIMI]`-komennolla (Git-yhteisö, n.d.). Toisaalta koska kehittäjäohjeet eivät ota kuvaa tarkasti, millaisia kokonaisuuksia tulisi yhteen commitiin sisällyttää (*TIMin koodikäytänteet* 2023), kuhunkin osaan lähdekoodia kohdistuneiden commitien määrä riippuu luultavasti myös kyseistä osaa kehittäneiden kehittäjien mieltymyksistä.

Koska Git-loki paljastaa myös kehittäjien ainakin joskus voimassa olleet sähköpostiosoitteet, voi ainakin niiden kautta lähettää myös haastattelukutsuja. Toisaalta se lisäisi työmäärää huomattavasti.

Git-loki- ja metriikkatuloksiin on tarkoituksenmukaista soveltaa tilastotiedettä. Myös laadullista analyysiä voinee tehdä.

4 Aineiston keruun suunnittelu

4.1 Työkalujen valinta

Ensimmäinen vaihe aineiston keruun suunnittelussa on analysoida saatavilla olevia staattisia koodimetriikkatyökaluja ja valita niistä sopivimmat. Staattisia koodimetriikoita mittaavia työkaluja etsitään tässä tutkimuksessa kolmen lähteen perusteella:

- Arvanitou ym. 2017. Artikkeli julkaistiin vertaisarviointia vaativassa tieteellisessä julkaisussa. Itse artikkeli ei kuvaa työkaluja kovin tarkkaan, mutta sen sisältämästä lähdeluettelosta löytyvät artikkelit kuvaavat niitä tarkemmin.
- Nilson, Antinyan ja Gren 2019. Artikkeli vertaisarvioitiin tieteellisessä konferenssissa, ja se luettelee itse mittaamiseen käytettyjä työkaluja.
- PyPI-pakettivarasto, jonka kautta Python-yhteisö jakaa avoimen lähdekoodin ohjelmistoa (P. S. Foundation 2023c). Pythonin saavuttaman suosion vuoksi myös PyPIä voi pitää uskottavana ohjelmistojen lähteenä.

Taulukoiden 1–6 havaintojen perusteella TIMin lähdekoodin mittaamiseen valitaan seuraavat työkalut:

- **SonarQube**: Työkalulla voidaan mitata McCabe- ja koginitiivinen kompleksisuus sekä koodirivien määrä.

Nimi	Kohdealueet
Columbus	C, C++ (Tiwari 2020)
aToucan	UML (Yue, Briand ja Labiche 2015)
Tooms	vaatimusmäärittelyjen luonti (Bucci ja Nesi 1995)

Taulukko 1: Nimetyistä työkaluista, jotka Arvanitou ym. (2017) löysivät, monipuolisimmin metriikoita mittaavat heidän mukaansa Columbus, aToucan ja Tooms, mutta yksikään niistä ei tue Pythonia. He löysivät myös useita Eclipse-liitännäisiä koodimetriikoiden mittaamiseen käyttäviä tutkimuksia erittelemättä niitä tarkemmin, mutta taulukko 5 tutkii tarkemmin Eclipse-liitännäisten mahdollisuutta staattisten koodimetriikoiden mittaamiseen Python-koodista.

Nimi	Kielet	Nykytila	Hinta	Metriikat
QAC	-	Työkalulle ei artikkelissa linkatulta työkalun kotisivulta löytynyt syyskuussa 2023 samannimistä vastinetta, mutta sivulta löytyivät vain C:tä ja C++:aa mittaavat QA-MISRA- ja Astrée-työkalut (QA-systems 2023).	-	-
Understand	Ada, Assembly, C, C++, C#, FORTRAN, Java, JOVIAL, Delphi/Pascal, Python, VHDL, Visual Basic[.NET], PHP, HTML, CSS, JavaScript, TypeScript, XML (SciTools 2023b)	Kehitetään ja tuetaan edelleen (SciTools 2023a).	100–120$/vuosi, hinta ja muut tilauksen yksityiskohdat on erikseen neuvoteltava. Ilmainen kokeiluversio tukee vain ohjelmiston mukana tulevaa koodinäytettä. (SciTools 2023a)	111 metriikkaa: kirjallisuuskartoituksessa mainituista McCabe-kompleksisuus, fan-in- ja fan-out-mitat, koodirivien määrä, perintäpuun syvyys, painotettu metodien summa luokassa, lasten lukumäärä, kohteiden välinen paritus, luokan vaste ja useita niiden johdannaisia (SciTools 2023c)

Taulukko 2: QAC:n ja Understandin ominaisuudet.

Nimi	Kielet	Nykytila	Hinta ja saatavuus	Metriikat
CPPDepend	C, C++, Java, VB6, VBA (CoderGears 2023a)	Kehitetään ja tuetaan edelleen (CoderGears 2023a).	Ilmainen kokeilujakso kestää 30 päivää ja vuositilaus yksittäiselle kehittäjälle maksaa 599 $ (CoderGears 2023b).	48 metriikkaa: kirjallisuuskartoituksessa mainituista kaikki paitsi Halsteadin T, fan-in ja fan-out, painotettu metodien summa luokassa, kohteiden välinen paritus ja luokan vaste
SonarQube	Java, Kotlin, C#, VB.Net, JavaScript, TypeScript, Python, PHP, Terraform, CloudFormation, CSS, Flex, Go, HTML5, Ruby, Scala, XML (S.A. 2023c)	Kehitetään ja tuetaan edelleen (S.A. 2023c).	Community Edition -versio on ilmainen ja tukee mainittuja kieliä ja metriikoita sekä Linuxia (S.A. 2023c) (S.A. 2023a), jota gradun tekijä käyttää.	kirjallisuuskartoituksessa kuvatuista metriikoista koodirivien määrä ja kognitiivinen sekä McCabe-kompleksisuus

Taulukko 3: CPPDependin ja SonarQuben ominaisuudet.

Nimi	Kielet	Nykytila	Hinta ja saatavuus	Metriikat
Eclipse Metrics Plugin	-	Ohjelmisto vaikutti syyskuussa 2023 hylätyltä. URL-osoitteesta, jonka Nilson, Antinyan ja Gren (2019) mainitsevat, löytyi esimerkiksi vanhentunut linkki (Walton 2023), eikä koko liitännäistä löytynyt Eclipsen virallisesta liitännäiskaupasta (E. Foundation 2023).	-	-
SourceMonitor	C++, C, C#, VB.NET, Java, Delphi, Visual Basic (VB6), HTML (Campwood Software 2023b)	Kehitys on lopetettu kehittäjän jäätyä eläkkeelle (Campwood Software 2023a) (Campwood Software 2023b).	Ohjelmiston lähdekoodi on nykyään avoin, mutta se toimii vain Windowsilla (Campwood Software 2023a) (Campwood Software 2023b).	Ei selviä ohjelmiston kotisivulla (Campwood Software 2023b).

Taulukko 4: Eclipse Metrics Pluginin ja SourceMonitorin ominaisuudet.

Nimi	Kielet	Nykytila	Hinta ja saatavuus	Metriikat
SonarLint (S.A. 2023b)	Java, JavaScript, PHP, Python, HTML	Kehitetään edelleen (S.A. 2023b).	avoimen lähdekoodin Eclipse-liitännäinen	Nimen ja kuvauksen perusteella SonarLint vaikuttaa tutkivan koodista yksinkertaisempia asioita kuin saman valmistajan SonarQube.
PyDev	Python, Jython, IronPython (Software 2023a)	Kehitetään edelleen (Software 2023a).	avoimen lähdekoodin Eclipse-liitännäinen (Software 2023a)	Mittaa koodista vain hyvin yksinkertaisia asioita, kuten ratkaisemattomia import-lauseita (Software 2023b).

Taulukko 5: 29.9.2023 hakufraasilla "*python metrics*"Eclipsen virallisesta liitännäiskaupasta löytyneiden liitännäisten ominaisuudet.

Nimi	Nykytila	Metriikat
mccabe	Viimeisin versio on tammikuulta 2022 (Batchelder 2023).	McCabe (Batchelder 2023)
cognitive-complexity	Viimeisin versio on elokuulta 2022 (Lebedev 2023).	kognitiivinen kompleksisuus, joskin kuvauksen mukaan "*This is not precise realization of original algorithm proposed by G. Ann Campbell, but it gives rather similar results*"(Lebedev 2023)
multimetric	Viimeisin versio on elokuulta 2023 (Weihmann 2023).	McCabe, Halsteadin V ja E, fan-out-mitta, koodirivien määrä ja useita metriikoita, joita ei kirjallisuuskartoituksessa löytynyt, mutta vain tiedostokohtaisesti (Weihmann 2023)
lcom	Viimeisin versio on lokakuulta 2017 (Wachowski 2023b).	metodien koheesion puute (Wachowski 2023b)
inheritance-explorer	Viimeisin versio on heinäkuulta 2023. (Wachowski 2023a)	luokkien väliset perintäketjut niiden syvyys mukaan lukien (Wachowski 2023a)

Taulukko 6: PyPI:n metriikoita koskevia laajennoksia (PyPI:n termistöllä projekteja (P. S. Foundation 2023b)). Inheritannce-explorerista on huomattava, että se vaatii tietyn "perusluokan", josta muut luokat on peritty, manuaalista tunnistusta. Se ei myöskään mittaa varsinaisesti staattista koodimetriikkaa, koska sen käyttö vaatii tutkittavan luokan tuomista osana suoritettavaa Python-koodia, jolloin siis vähintään luokan konstruktori suoritetaan. (Wachowski 2023a)

- PyPI-projekteista:
  - **mccabe**
  - **cognitive-complexity**: Vaikka kognitiivista kompleksisuutta voi mitata myös SonarQubella, on aiheellista selvittää, eroavatko mittaustulokset keskenään ja saadanko tukea havainnolle, jonka Lincke, Lundberg ja Löwe (2008) tekivät.
  - **multimetric**: Työkalulla voidaan mitata McCabe-kompleksisuus, Halsteadin E- ja V-muuttujat, fan-out ja koodirivien määrä. McCabe-kompleksisuustulosta voi verrata SonarQuben ja mccabe-projektin tuloksiin.
  - **lcom**

McCabe-kompleksisuus ja kogntiivinen kompleksisuus on suunnattu funktioiden mittaamiseen, joten SonarQuben, mccaben ja cognitive-complexityn mittaustulokset kerätään funktiokohtaisesti. Multimetricin mittaustulokset kerätään tiedostokohtaisesti työkalun rajallisuuden mukaan. Lcomin mittaustulokset kerätään luokkakohtaisesti metriikan luonteen mukaan.

4.2 Kehitettävyyden ja ylläpidettävyyden mittaaminen

Kirjallisuuskartoituksen perusteella laadun osa-alueista ylläpidettävyyden mittaaminen on kyseenalaisinta, vaikka se onkin periaatteessa melko objektiivista. Koska se on myös kuvatuista laatuattribuuteista ainoa, joka ilmenee suoraan vain kehittäjille ja siitä käytetään myös nimeä *code quality* (Wagner 2013), voi uskoa, että nimenomaan lähdekoodia mittaavat työkalut arvioivat siitä.

Ohjelmistokomponenttiin, kuten funktioon tai tiedostoon, kohdistuneiden muutosten määrä mittaa sen kehitys- ja ylläpitotarpeita. Kerralla oikein tehtyä ohjelmistoa ei sen itsensä takia tarvitse muuttaa, koska ohjelmisto ei kulu käytettäessä. Koska TIMin koko kehityshistoria tunnetaan Git-lokin perusteella, muutosten määrä voidaan mitata Git-commitien lukumäärällä.

5 Aineiston keruu

5.1 Tutkimusalueen määrittely

Aineiston keruualueeksi määritellään tutkimuskysymyksessä TIMin Pythonilla toteutettu palvelinohjelmisto, mutta käsitteen tarkempi määrittely on jonkin verran kyseenalaista. Ilmeistä kuitenkin on, että kolmannen osapuolen toteuttamia Node-paketteja ei sisällytetä tutkimusalueeseen, koska niiden kehitysprosessi on erillinen. TIMin Pythonilla toteutettuja timApp/tests-hakemiston testiskriptejä ei sisällytetä tutkimusalueeseen, koska testikoodin luonne on erilainen. Myöskään timApp/migrations-hakemiston tiedostoja ei sisällytetä, koska ne on luotu automaattisesti (JYU 2023b). Nämä poislukien TIMin GitHub-pääsäiliön juurihakemiston timApp-hakemistossa oli 25. lokakuuta 2023 92132 riviä Python-tiedostoja, jotka sisällytetään tutkimusalueeseen. Tutkimusalueella ovat TIMin pääkontin ja ainakin osittain Pythonilla toteutetuista muista konteista CsPluginin, Dragin, Feedbackin, Fieldsin, ImageX:n, Palin ja ShowFilen lähdekoodit. (JYU 2023a)

Osa TIMin palvelinohjelmiston käyttämästä Python-koodista, kuten rajapinta dokumenttilistojen muuttamiseen HTML-koodiksi Dumbolla, on tallennettu juurihakemiston tim_common-hakemistoon, jossa oli 25. lokakuuta 2023 4059 riviä Python-tiedostoja. tim_common sisällytetään tutkimusalueeseen. Käynnissä olevan TIM-instanssin hallinnointiin komentorivillä tarkoitetut Python-skriptit on tallennettu cli-hakemistoon, mutta koska TIM-sovellus ei varsinaisesti käytä niitä vaan TIMin ylläpitäjä tai kehittäjä käyttää niitä TIMiä hallinnoidakseen, cli-hakemisto jätetään tutkimusalueesta pois. TIMin juurihakemistossa ei 25. lokakuuta 2023 ollut yhtään Python-tiedostoa. (JYU 2023a)

TIMiin suoraan liittyvää ohjelmistoa on kehitetty myös hiukan muissa GitHub-säiliöissä, mutta ne ovat pääsäiliöön verrattuna hyvin pieniä ja liittyvät enemmän TIMin kehittämisen helpottamiseen kuin varsinaiseen käyttöön (JYU 2023d).

On huomioitava, että kerättävää aineistoa kehitetään jatkuvasti. Esimerkiksi 1.-20. lokakuuta 2023 TIMin pääsäiliön päähaaraan on tehty 59 muutosta (JYU 2023a). Datan vertailukelpoisuuden ylläpitämiseksi kaikki data mitataan TIMin 25. lokakuuta 2023 haetusta versiosta,

vaikka kesken keräämisen julkaistaisiinkin uusi versio.

5.2 Tiedostojen, luokkien ja funktioiden keruu

TIMin lähdekoodista kerättävä data tallennetaan kolmeen SQLite-tauluun. SQLiten etu on, ettei erillistä asiakas-palvelinarkkitehtuuria tarvita. Toisaalta ohjelmiston yksikköihin liittyvä ja niistä mitattu data on niin rakenteista, että relaatiotietokanta sopii hyvin sen käsittelyyn.

Listing 5.1: Tämä Python-skripti hakee kaikki Python-tiedostot valitusta hakemistosta (timApp tai tim_common) ja sen alihakemistoista poislukien mainitut alihakemistot, jotka eivät kuulu tutkimusalueeseen.

```
1  import subprocess, sqlite3, sys
2
3  tiedostot = list(filter(lambda polku: not
       polku.startswith('timApp/tests') and 'node_modules' not in polku
       and not polku.startswith('timApp/migrations') and polku != '',
4      subprocess.run(['find', sys.argv[2], '-name', '*.py'],
           stdout=subprocess.PIPE).stdout.decode('utf-8').split('\n')))
5
6  con = sqlite3.connect(sys.argv[1])
7  cur = con.cursor()
8
9  for tiedosto in tiedostot:
10     cur.execute('insert into tiedostot values(?,0,0,0,0,0,0)',
           (tiedosto,))
11
12 con.commit()
13 cur.close()
14 con.close()
```

Listing 5.2: Tämä hakee Python-funktiot ja -luokat SQLite-tietokannan Python-tiedostoista ja tallentaa niiden nimet, tiedostot ja alku- ja loppurivit siihen. Funktioista tallennetaan myös mahdollinen luokka, johon ne kuuluvat. Luokan sisäiset luokat ja funktiot huomioidaan, mutta funktion sisäisiä luokkia ja funktioita ei.

```
1  import sys, sqlite3, ast
2
```

```
3  con = sqlite3.connect(sys.argv[1])
4  cur = con.cursor()
5
6  # Jotta sisäkkäiset luokat ja funktiot huomioitaisiin, luokat luetaan
       ja niiden attribuutit ja sisältö tallennetaan tietokantaan
       rekursiivisesti.
7  def parse_luokka(luokka):
8      cur.execute('insert into luokat values(?,?,?,?,0,0)',(polku,
           luokka.name, luokka.lineno, luokka.end_lineno,))
9      for funktio in [n for n in luokka.body if isinstance(n,
           ast.FunctionDef)]:
10         cur.execute('insert into funktiot
               values(?,?,?,?,?,0,0,0,0,0)', (polku, luokka.name,
               funktio.name, funktio.lineno, funktio.end_lineno,))
11     for aliluokka in [n for n in luokka.body if isinstance(n,
           ast.ClassDef)]:
12         parse_luokka(aliluokka)
13
14 # Luokkia ja funktioita haetaan niistä tiedostoista, jotka
       tietokannasta jo löytyvät.
15 cur.execute('select polku from tiedostot;')
16 polut = [tiedosto[0] for tiedosto in cur.fetchall()]
17 for polku in polut:
18     with open(polku, 'r') as sisalto:
19         try:
20             puu = ast.parse(sisalto.read())
21         except SyntaxError as e:
22             print('Syntaksivirhe: {}'.format(e))
23         luokat = [n for n in puu.body if isinstance(n, ast.ClassDef)]
24         for luokka in luokat:
25             parse_luokka(luokka)
26         funktiot = [n for n in puu.body if isinstance(n,
               ast.FunctionDef)]
27         for funktio in funktiot:
28             cur.execute('insert into funktiot
                   values(?,?,?,?,?,0,0,0,0,0)', (polku, '', funktio.name,
                   funktio.lineno, funktio.end_lineno,))
```

```
29
30  con.commit()
31  cur.close()
32  con.close()
```

Skriptit 5.1 ja 5.2 löysivät 25. lokakuuta 2023 TIMin pääsäiliön päähaarasta tutkimusaluen osasta hakemistorakennetta 410 Python-tiedostoa, 833 Python-luokkaa ja 3924 Python-funktiota. Funktioista 1908 kuului johonkin luokkaan ja 2016 ei.

Listing 5.3: Tämä hakee tietokannan Python-tiedostoista funktion sisäisiä luokkia ja funktioita ja luokan sisäisiä luokkia ja huomauttaa niistä tulosteessa, muttei muokkaa tietokantaa. Luokan sisäisistä funktioista ei huomauteta, koska ne ovat niin odotettavia.

```
1   import sys, sqlite3, ast
2
3   con = sqlite3.connect(sys.argv[1])
4   cur = con.cursor()
5
6   def tutki_luokka(polku, luokka, luokkarekursio, funktiorekursio):
7       if luokkarekursio != 0:
8           print('Luokan sisäinen luokka', polku, luokka.name,
                luokkarekursio)
9       if funktiorekursio != 0:
10          print('Funktion sisäinen luokka', polku, luokka.name,
                funktiorekursio)
11      luokat = [n for n in luokka.body if isinstance(n, ast.ClassDef)]
12      for aliluokka in luokat:
13          tutki_luokka(polku, aliluokka, luokkarekursio + 1,
                funktiorekursio)
14      funktiot = [n for n in luokka.body if isinstance(n, ast.ClassDef)]
15      for funktio in funktiot:
16          tutki_funktio(polku, funktio, luokkarekursio + 1,
                funktiorekursio)
17
18  def tutki_funktio(polku, funktio, luokkarekursio, funktiorekursio):
19      if funktiorekursio != 0:
20          print('Sisäkkäinen funktio', polku, funktio.name,
                funktiorekursio)
```

```
        funktiot = [n for n in funktio.body if isinstance(n,
            ast.FunctionDef)]
        for alifunktio in funktiot:
            tutki_funktio(polku, alifunktio, luokkarekursio,
                funktiorekursio + 1)
        luokat = [n for n in funktio.body if isinstance(n, ast.ClassDef)]
        for aliluokka in luokat:
            tutki_luokka(polku, aliluokka, luokkarekursio, funktiorekursio
                + 1)

cur.execute('select polku from tiedostot;')
polut = [tiedosto[0] for tiedosto in cur.fetchall()]
for polku in polut:
    with open(polku, 'r') as sisalto:
        try:
            puu = ast.parse(sisalto.read())
        except SyntaxError as e:
            print('Syntaksivirhe: {}'.format(e))
    luokat = [n for n in puu.body if isinstance(n, ast.ClassDef)]
    funktiot = [n for n in puu.body if isinstance(n, ast.FunctionDef)]
    for luokka in luokat:
        tutki_luokka(polku, luokka, 0, 0)
    for funktio in funktiot:
        tutki_funktio(polku, funktio, 0, 0)
```

On huomattavaa, ettei funktion sisäisiä luokkia tai funktioita tallennettu tietokantaan. Skriptillä 5.3 kuitenkin tutkittiin myös niitä. Tutkimusalueelta löytyi kaksi funktion sisäistä luokkaa, mutta kumpikaan niistä ei ole kovin suuri. Funktion sisäisiä funktioita sen sijaan löytyi skriptillä 110 eli huomattava määrä. Toisaalta käytettävät metriikkatyökalut eivät vaikuta yksimielisesti tunnustavan niitä. PyPI:n mccabe-projekti ei kokeilujen perusteella mittaa sisäisiä funktioita erikseen vaan vain laskee niiden McCabe-kompleksisuuden osaksi ulomman funktion McCabe-kompleksisuutta. PyPI:n cognitive-complexity-projekti kuitenkin vaikuttaa käsittelevän omina funktioinaan sellaisia funktion sisäisiä funktioita, joita ulompi funktio käyttää. Yhdenmukaisuuden vuoksi ja koska alkuperäiset McCabe-kompleksisuuden ja kognitiivisen kompleksisuuden määritelmät eivät huomioi funktion sisäisiä luokkia tai funktiota,

ne sivuutetaan aineistoa kerätessä.

5.3 Metriikoiden keruu

5.3.1 McCabe

PyPI:n mccabe-projekti löysi skriptillä 5.4 kohdealueen Python-funktioista McCabe-kompleksisuuden kaikille paitsi 25 funktiolle. Näistä 12:llä ei ollut varsinaista sisältöä vaan funktion esittelyrivin jälkeen koodissa oli pelkkä ..., joten tulos on ymmärrettävä. Loput olivat @property-dekoraattorilla varustettuja attribuutin kuvaavia funktioita, joita seurasi vastaava @[ATTRIBUUTIN_NIMI].setter-dekoraattorilla varustettu funktio. Ei ole täysin selvää, miksi juuri tämä kuvio estää McCabe-kompleksisuuden mittaamisen, mutta vain 0,3 prosenttia aineistosta koskeva puute on niin epäolennainen, että sen voi sivuuttaa.

Listing 5.4: Tämä hakee McCabe-kompleksisuudet PyPI:n mccabe-projektilla tietokannasta löytyvien Python-tiedostojen funktioille.

```
import subprocess, sqlite3, sys, re

con = sqlite3.connect(sys.argv[1])
cur = con.cursor()
cur.execute('select polku from tiedostot;')
polut = [tiedosto[0] for tiedosto in cur.fetchall()]
for polku in polut:
    funktiot = subprocess.run(['python3', '-m', 'mccabe', polku],
        stdout=subprocess.PIPE).stdout.decode('utf-8').split('\n')
    for funktio in funktiot:
        sarakkeet = funktio.split(' ')
        if 1 < len(sarakkeet) and re.match('\d+:\d+', sarakkeet[0]):
            rivialku = int(sarakkeet[0].split(':')[0])
            mccabe = int(sarakkeet[-1])
            cur.execute('update funktiot set mccabe=? where polku=?
                and alku=?',(mccabe,polku,rivialku,))

con.commit()
cur.close()
con.close()
```

PyPI:n multimetric-projekti puolestaan löysi skriptillä 5.5 kohdealueen Python-tiedostoista McCabe-kompleksisuuden 305 tiedostolle. Muihin 105 tiedostoon ei liittnyt yhtään funktiota, joten niiden McCabe-kompleksisuutta ei edes yritetty laskea.

Listing 5.5: Tämä hakee McCabe-kompleksisuudet PyPI:n multimetric-projektilla tietokannasta löytyville Python-tiedostoille, joihin liittyy ainakin yksi tietokannan funktio.

```
import sqlite3, json, subprocess, sys, os

con = sqlite3.connect(sys.argv[1])
cur = con.cursor()
cur.execute('select polku from tiedostot;')
polut = [tiedosto[0] for tiedosto in cur.fetchall()]
for polku in polut:
    if cur.execute('select count(*) from funktiot where polku = ?',
        (polku,)).fetchone()[0] == 0:
         continue
    output = json.loads(subprocess.run(['multimetric', polku],
        stdout=subprocess.PIPE).stdout)
    if output is not None:
        mccabe = output['files']['{}/{}'.format(os.getcwd(),
            polku)]['cyclomatic_complexity']
        cur.execute('update tiedostot set mccabe_multimetric=? where
            polku=?',(mccabe,polku,))

con.commit()
cur.close()
con.close()
```

SonarQubea ei ole suunniteltu käytettäväksi pelkällä komentorivillä ainakaan ilmaisen Community Edition -version osalta vaan lähinnä selaimessa. Selainkäyttöliittymän kautta täytyi säätää TIM-projektin skannauksessa käytettävän profiilin parametreja eli McCabe-kompleksisuutta mitatessa "suurin sallittu"McCabe-kompleksisuus oli säädettävä nollaan. Skannauksen jälkeen kaikkia "ongelmia"eli tässä tapauksessa mitä tahansa mitattua funktiota ja sen McCabe-kompleksisuutta pystyi hakemaan SonarQuben API:n kautta, joka antoi tulokset JSON-tiedostoina. JSON-tiedostoista tulokset luettiin ja tallennettiin SQLite-tietokantaan

skriptillä 5.6. SQLiteen jo tallennetuista funktioista SonarQube löysi McCabe-kompleksisuuden kaikille paitsi 18 funktiolle. Nämä 18 funktiota ovat kaikki timApp/answer/answers.py-tiedoston funktiot. Ilmeistä syytä juuri tämän tiedoston sivuuttamiselle ei ole, mutta puute on niin pieni, että se sivuutetaan.

Listing 5.6: Tämä skripti hakee SonarQuben API:sta ladatuista, tiettyyn hakemistoon tallennetuista JSON-tiedostoista funktioiden McCabe-kompleksisuudet funktion polun ja alkurivin perusteella. Kompleksisuudet kopioidaan SQLite3-tietokantaan.

```
import json, sys, parse, os, sqlite3

kompleksisuus_string = 'Function has a complexity of {} which is
    greater than 0 authorized.'
polku_string = 'TIM:{}'

con = sqlite3.connect(sys.argv[2])
cur = con.cursor()

for tiedosto in os.listdir(sys.argv[1]):
    for funktio in json.load(open('{}/{}'.format(sys.argv[1],
        tiedosto)))['issues']:
        tiedosto = parse.parse(polku_string, funktio['component'])[0]
        rivi = funktio['line']
        kompleksisuus = parse.parse(kompleksisuus_string,
            funktio['message'])[0]
        cur.execute('update funktiot set sonarqube_mccabe=? where
            polku=? and alku=?', (kompleksisuus, tiedosto, rivi,))

con.commit()
cur.close()
con.close()
```

5.3.2 Kognitiivinen kompleksisuus

Funktioiden kognitiivisten kompleksisuuksien mittaaminen SonarQubella ja tulosten tallentaminen tietokantaan SonarQuben API:n kautta skriptillä 5.7 tapahtui hyvin vastaavasti kuin McCabe-kompleksisuuksien. Skannauksessa käytettävän profiilin "suurin sallittu"kognitiivi-

nen kompleksisuus oli säädettävä -1:teen ja skannauksen jälkeen API:sta saattoi hakea tulokset JSON-tiedostoina, joista mitat pystyi tallentamaan SQLiteen. SonarQube löysi kognitiivisen kompleksisuuden kaikille paitsi timApp/answer/answers.py-tiedoston 18 funktiolle. Nämä funktiot ovat samat kuin joille SonarQube ei löytänyt McCabe-kompleksisuutta. Puute on erikoinen, mutta niin vähäinen, että se sivuutetaan.

Listing 5.7: Tämä skripti hakee SonarQuben API:sta ladatuista, tiettyyn hakemistoon tallennetuista JSON-tiedostoista funktioiden kognitiiviset kompleksisuudet funktion polun ja alkurivin perusteella. Kompleksisuudet kopioidaan SQLite3-tietokantaan.

```
import json, sys, parse, os, sqlite3

kompleksisuus_string = 'Refactor this function to reduce its
    Cognitive Complexity from {} to the -1 allowed.'
polku_string = 'TIM:{}'

con = sqlite3.connect(sys.argv[2])
cur = con.cursor()

for tiedosto in os.listdir(sys.argv[1]):
    for funktio in json.load(open('{}/{}'.format(sys.argv[1],
        tiedosto)))['issues']:
        polku = parse.parse(polku_string, funktio['component'])[0]
        rivi = funktio['line']
        kompleksisuus = parse.parse(kompleksisuus_string,
            funktio['message'])[0]
        cur.execute('update funktiot set sonarqube_cognitive=? where
            polku=? and alku=?', (kompleksisuus, polku, rivi,))

con.commit()
cur.close()
con.close()
```

PyPI:n cognitive-complexity-projekti löysi skriptillä 5.8 kognitiivisen kompleksisuuden kaikille tietokannan funktioille. Tulokset sopivat hyvin yhteen SonarQuben tulosten kanssa. Kummankin mukaan keskimääräinen kognitiivinen kompleksisuus on noin 4,03 ja kummankin mukaan kompleksisin funktio on timApp/modules/cs/cs.py-tiedostosta löytyvä

do_all_t-funktio, jonka kognitiivinen kompleksisuus on cognitive-complexityn mukaan 293 ja SonarQuben mukaan 289.

Listing 5.8: Tämä skripti hakee PyPI:n cognitive-complexity-projektia käyttävällä flake8-laajennuksella kaikkien tietokannasta löytyvien tiedostojen Python-funktioiden kognitiiviset kompleksisuudet ja tallentaa ne tietokantaan.

```
import subprocess, sqlite3, sys, parse

kompleksisuus_string = '{}:{}:{}: CCR001 Cognitive complexity is too
    high ({} > -1)'

con = sqlite3.connect(sys.argv[1])
cur = con.cursor()
cur.execute('select polku from tiedostot;')
polut = [tiedosto[0] for tiedosto in cur.fetchall()]
for polku in polut:
    rivit = subprocess.run(['flake8',
        '--max-cognitive-complexity=-1', polku],
        stdout=subprocess.PIPE).stdout.decode('utf-8').split('\n')
    for rivi in rivit:
        kompleksisuus = parse.parse(kompleksisuus_string, rivi)
        if kompleksisuus is not None:
            cur.execute('update funktiot set cognitive=? where
                polku=? and alku=?',
                (kompleksisuus[3],kompleksisuus[0],kompleksisuus[1],))

con.commit()
cur.close()
con.close()
```

5.3.3 LCOM

Skripti 5.9 löysi metodien koheesion puutteen kaikille paitsi 125 tietokannan luokalle. Mitään ilmeistä yhteistä ja muista luokista erottavaa ominaisuutta niillä ei ole, mutta muiden sopivien työkalujen puutteessa tyydytään sen tuloksiin. Suurin yksittäinen metodien koheesion puute on timApp/upload/uploadedfile.py-tiedostosta löytyvällä UploadedFile-

luokalla, jolla se on 6, mutta keskimääräinen mitattu LCOM on vain noin 0,41.

Listing 5.9: Tämä skripti hakee PyPI:n lcom-projektilla metodien koheesion puutteen tietokannasta löytyvien tiedostojen luokille ja tallentaa ne samaan tietokantaan. Projekti tulostaa tulokset muodosssa, josta data on kätevimmin kerättävissä säännöllisellä lausekkeella.

```
import sqlite3, subprocess, sys, re

con = sqlite3.connect(sys.argv[1])
cur = con.cursor()
cur.execute('select polku from tiedostot;')
polut = [tiedosto[0] for tiedosto in cur.fetchall()]
for polku in polut:
    tulos = subprocess.run(['lcom', polku],
        stdout=subprocess.PIPE).stdout.decode('utf-8').split('\n')
    for rivi in tulos:
        if re.fullmatch('\| {}\.[A-Z|a-z]+( )*\| \d+(
            )*\|'.format(polku.replace('/', '\.').replace('.py',
            '')), rivi):
             [_,luokka,lcom,_] = rivi.split('|')
             luokka = luokka.replace(' ', '').split('.')[-1]
             lcom = lcom.replace(' ', '')
             cur.execute('update luokat set lcom=? where polku=? and
                 nimi=?', (lcom,polku,luokka,))

con.commit()
cur.close()
con.close()
```

5.3.4 Halsteadin metriikat, fan-out ja koodirivien määrä

Koska sekä Halsteadin metriikat, fan-out-mitta että koodiriven määrä kerätään samalla työkalulla tiedostokohtaisesti, ne on kätevintä kerätä samalla kertaa. Skripti 5.10 löysi kaikki mitattavat mitat kaikille paitsi 63 tyhjälle tiedostolle. Fan-out-mitan multimetric jakaa "sisäiseen"ja "ulkoiseen"fan-outiin sen mukaan, tuodaanko olio samasta "lähdepuumoduulista"(*source tree module*) (Weihmann 2023), mutta tarkemman kuvauksen puutteessa ne lasketaan yhteen. Tyhjät tiedostot olivat kaikki muualle sovellukseen tuotavan, yhteen hakemis-

toon tallennetun Python-paketin juurihakemistossa olevia __init__.py-tiedostoja.

Listing 5.10: Tämä skripti hakee multimetric-projektilla kullekin tietokannan tiedostolle Halsteadin E- ja V-mitat, fan-out-mitan ja koodirivien määrän.

```
import sqlite3, subprocess, sys, json, os

con = sqlite3.connect(sys.argv[1])
cur = con.cursor()
cur.execute('select polku from tiedostot;')
polut = [tiedosto[0] for tiedosto in cur.fetchall()]
for polku in polut:
    if os.stat(polku).st_size == 0:
        continue
    output = json.loads(subprocess.run(['multimetric', polku],
        stdout=subprocess.PIPE).stdout)
    if output is not None:
        data = output['files']['{}/{}'.format(os.getcwd(), polku)]
        cur.execute('update tiedostot set e=?, v=?, fan_out=?, loc=?
            where polku=?',(
             data['halstead_effort'],
             data['halstead_volume'],
             data['fanout_internal'] + data['fanout_external'],
             data['loc'],
             polku,
        ))

con.commit()
cur.close()
con.close()
```

5.4 Commitien laskeminen

Toinen tapa mitata ohjelmistoyksiköiden ongelmallisuutta on niihin kohdistuneen muutoksen tarve, jota mitataan niihin kohdistuneiden Git-commitien määrällä. Gitin log-komento mahdollistaa lokien haun tietyn tiedoston ja myös tietyn rivialueen osalta. Skripti 5.11 löysi commitien määrän kaikille tietokannan tiedostoille ja luokille ja kaikille paitsi kolmelle funk-

tiolle. Näitä kolmea funktiota koskenutta git log-tulostetta ei pystytty dekoodaamaan UTF-8-merkistöllä, mutta käyttämällä komentoa manuaalisesti niillekin saatiin laskettua commitien määrät. Keskimäärin funktioon oli kohdistunut noin 7,7, luokkaan 8,2 ja tiedostoon 24 commitia. Muokatuin funktio oli dokumentin HTML-sivuksi muuntava timApp/item/routes.py-tiedoston render_doc_view-funktio, johon oli kohdistunut 321 commitia.

Listing 5.11: Tämä skripti hakee Git-lokin kullekin tietokannan tiedostolle, funktiolle ja luokalle ja laskee kuhunkin tiedostoon, funktioon ja luokkaan kohdistuneiden commitien määrän.

```
1  import sqlite3, sys, subprocess, re
2
3  dbpath = sys.argv[1]
4  con = sqlite3.connect(dbpath)
5  cur = con.cursor()
6
7  cur.execute('select count(*) from funktiot')
8  funktioita = cur.fetchone()[0]
9  cur.execute('select polku,nimi,alku,loppu from funktiot')
10 i = 0
11 for funktio in cur.fetchall():
12     if i % 100 == 0:
13         print('Funktioista luettu: {}/{}'.format(i, funktioita))
14     i += 1
15     try:
16         gitloki = subprocess.run(['git', 'log', '--no-merges',
               '-L{},{}:{}'.format(funktio[2], funktio[3], funktio[0])],
               stdout=subprocess.PIPE).stdout.decode('utf-8')
17     except UnicodeDecodeError:
18         print('Tulosteen tulkinta epäonnistui: ', funktio)
19         continue
20     commiteja = len(re.findall('([0-9]|[a-f]){40}', gitloki))
21     cur.execute('update funktiot set commiteja=? where polku=? and
           nimi=?', (commiteja, funktio[0], funktio[1],))
22
23 cur.execute('select count(*) from luokat')
24 luokkia = cur.fetchone()[0]
25 cur.execute('select polku,nimi,alku,loppu from luokat')
```

```
26     i = 0
27     for luokka in cur.fetchall():
28         if i % 100 == 0:
29             print('Luokista luettu: {}/{}'.format(i, luokkia))
30         i += 1
31         try:
32             gitloki = subprocess.run(['git', 'log', '--no-merges',
                   '-L{},{}:{}'.format(luokka[2], luokka[3], luokka[0])],
                   stdout=subprocess.PIPE).stdout.decode('utf-8')
33         except UnicodeDecodeError:
34             print('Tulosteen tulkinta epäonnistui: ', luokka)
35             continue
36         commiteja = len(re.findall('([0-9]|[a-f]){40}', gitloki))
37         cur.execute('update luokat set commiteja=? where polku=? and
               nimi=?', (commiteja, luokka[0], luokka[1],))
38 
39     cur.execute('select count(*) from tiedostot')
40     tiedostoja = cur.fetchone()[0]
41     cur.execute('select polku from tiedostot')
42     i = 0
43     for tiedosto in cur.fetchall():
44         if i % 100 == 0:
45             print('Tiedostoista luettu: {}/{}'.format(i, tiedostoja))
46         i += 1
47         try:
48             gitloki = subprocess.run(['git', 'log', '--no-merges',
                   tiedosto[0]],
                   stdout=subprocess.PIPE).stdout.decode('utf-8')
49         except UnicodeDecodeError:
50             print('Tulosteen tulkinta epäonnistui: ', tiedosto)
51             continue
52         commiteja = len(re.findall('([0-9]|[a-f]){40}', gitloki))
53         cur.execute('update tiedostot set commiteja=? where polku=?',
               (commiteja, tiedosto[0],))
54 
55     con.commit()
56     cur.close()
```

```
57      con.close()
```

6 Aineiston analyysi

6.1 Työkalujen yhteneväisyys

Eri työkalujen on havaittu mittaavan samaa nimellistä metriikkaa eri tavoin (Lincke, Lundberg ja Löwe 2008), joten ensin tutkitaan, missä määrin tämä pätee McCabe-kompleksisuuteen ja kognitiiviseen kompleksisuuteen. Edellistä mitattin kolmella ja jälkimmäistä kahdella työkalulla. Muita metriikoita mitattiin vain yhdellä työkalulla.

6.1.1 McCabe

PyPI:n mccabe-projekti ja SonarQube laskivat 3924 funktiosta 1277:lle eri McCabe-kompleksisuuden. 451 funktiolla on SonarQuben mukaan pienempi kompleksisuus kuin mccaben mukaan ja 783:lla suurempi. SonarQuben mukaan keskimääräinen kompleksisuus on 3,28 ja McCaben mukaan 3,21. Kompleksisin funktio on kummankin mukaan timApp/modules/cs/cs.py-tiedoston do_all_t-funktio, mutta SonarQuben mukaan sen McCabe-kompleksisuus on 155 ja mccaben mukaan "vain"139. Pearsonin korrelaatio SonarQuben ja mccaben tulosten välillä on kuitenkin 0,945, joten yleisesti ottaen ne ovat hyvin yhteneviä.

Ainakin and- ja or-lauseita SonarQube ja mccabe vaikuttavat kohtelevan eri tavoin. Esimerkiksi skripteissä 6.1 ja 6.2 kuvattujen kahden funktion McCabe-kompleksisuus on SonarQuben mukaan 2 mutta mccaben mukaan vain 1:

Listing 6.1: Tämä funktio on timApp/util/get_fields.py-tiedoston TallyField-luokassa.

```
1     @property
2     def effective_doc_id(self):
3         return self.doc_id or self.default_doc.id
```

Listing 6.2: Tämä funktio on timApp/user/usergroup.py-tiedoston UserGroup-luokassa.

```
1     @property
2     def is_sisu_student_group(self):
3         return self.is_sisu and
              self.external_id.external_id.endswith("-students")
```

Alkuperäisen McCabe-kompleksisuuden kuvaavan artikkelin mukaan SonarQube on oikeassa. Jos ensimmäinen ehto on and-lausessa tosi tai or-lauseessa epätosi, aliohjelman suoritus siirtyy toiseen ehtoon ja muuten aliohjelman suoritus loppuu heti. Tiloja on siis 4: alku, ensimmäistä ehtoa tutkiva tila, jälkimmäistä tutkiva tila ja loppu. Mahdollisia siirtymiä niiden välillä on myös 4: alusta ensimmäiseen ehtoon, ensimmäisestä ehdosta loppuun, ensimmäisestä ehdosta toiseen ja toisesta ehdosta loppuun. $4-4+2*1=2$ eli McCabe-kompleksisuus on 2. (McCabe 1976) PyPI:n mccabe vaikuttaa kohtelevan Boolen operaattoreilla yhdistettyjä ehtolauseita yhtenä tilana.

Multimetricin tuloksissa on hiukan enemmän eroavaisuutta. Pearsonin korrelaatio sen tulosten ja tiedostokohtaisten, mccaben laskemien kompleksisuuksien summien välillä on 0,86. Merkittävä ero on esimerkiksi timApp/launch.py-tiedostossa, joka käynnistää varsinaisen TIMin Flask-sovelluksen. Suurin osa tiedostosta ei ole osa mitään funktiota vaan Python-skriptiä, joka suoritetaan, jos tiedosto suoritetaan skriptinä eikä sitä ole tuotu moduulina. Tiedostossa on vain yksi yhden rivin funktio, jota kutsutaan, jos PYCHARM_HOSTED-ympäristömuuttuja on asetettu. PyPI:n mccabe laski funktion kompleksisuudeksi 1 ja jätti muun tiedoston huomiotta, mutta multimetric ilmeisesti laski muun tiedoston omaksi funktiokseen ja laski koko tiedoston kompleksisuudeksi 6.

Päinvastainen esimerkki on tim_common/utils.py-tiedosto. Multimetricin mukaan sen McCabe-kompleksisuus on 1 ja PyPI:n mccaben mukaan sen funktioiden kompleksisuuksien summa on 16. Tiedostossa on 8 funktiota, joten multimetricin tulos ei edes teoriassa voi olla oikein. Toisaalta funktioista kaikki paitsi yksi ovat joko luokan sisäisiä, sisältävät pelkästään ...-rivin esittelyrivin jälkeen tai heti esittelyrivin jälkeen on merkkijonona kuvattu funktiodokumentaatio. Multimetricin tapa tunnistaa funktioita vaikuttaa melko kapealta.

Pearsonin korrelaatio multimetricin ja SonarQuben mittaamien McCabe-kompleksisuuksien välillä on 0,883 eli hiukan vahvempi. Erot vaikuttavat melko samoilta kuin edellisen ja PyPI:n mccaben välillä. Esimerkiksi tim_common/utils.py-tiedoston funktioiden kompleksisuuksien summa on myös SonarQuben mukaan 16 ja timApp/launch.py-tiedoston 1.

Koska ainakin and- ja or-lauseet SonarQube vaikuttaa käsittelevän paremmin kuin PyPI:n mccabe ja yhteys niiden tulosten välillä on niin vahva, metriikoita analysoidessa huomioi-

SonarQube - PyPI:n mccabe	SonarQube - multimetric	PyPI:n mccabe - multimetric
0,945	0,883	0,860

Taulukko 7: SonarQuben, PyPI:n mccaben ja PyPI:n multimetricin mittaamien McCabe-kompleksisuuksien Pearsonin korrelaatiot. Kahden ensimmäisen välinen korrelaatio on funktiokohtainen, muut tiedostokohtaisia.

daan vain SonarQuben tulokset. Multimetricin tulokset eroavat sen verran, että ne huomioidaan erikseen.

Taulukko 7 luettelee eri työkalujen samoista ohjelmistoyksiköistä mittaamien McCabe-kompleksisuuksien väliset korrelaatiot.

6.1.2 Kognitiivinen kompleksisuus

SonarQuben ja PyPI:n cognitive-complexity-projektin tulokset kognitiivisesta kompleksisuudesta ovat hyvin yhteneväisiä. Pearsonin korrelaatio tulosten välillä on peräti 0,998. Eri tuloksen työkalut antoivat esimerkiksi timApp/readmark/readmarkcollection.py-tiedoston class_str-funktiolle, joka on kuvattu skriptissä 6.3. Työkalut vaikuttavat kohtelevan joko yhden rivin if-else-lausetta tai for-lauseella muodostettua uutta listaa eri tavoin.

Listing 6.3: timApp/readmark/readmarkcollection.py-tiedoston class_str-funktio. Tämän kognitiivinen kompleksisuus on SonarQuben mukaan 2 ja PyPI:n cognitive-complexity-projektin mukaan 3.

```
1      @property
2      def class_str(self):
3          s = "readline"
4          for c in (
5              r.type.class_str + ("-modified" if r.modified else "") for r
                   in self.marks
6          ):
7              s += " " + c
8          return s
```

Päinvastainen esimerkki on skriptissä 6.4 kuvattu timApp/modules/cs/languages.py-tie-

doston LanguageError-luokan konstruktori, jonka suorituksessa on pelkät try- ja except-lohkot sekä else-lohko niiden jälkeen.

Listing 6.4: SonarQuben mukaan tämän funktion kognitiivinen kompleksisuus on 2 ja PyPI:n cognitive-complexityn mukaan 1.

```python
    def __init__(self, query, error_str, sourcecode=""):
        try:
            super().__init__(query, sourcecode)
        except Exception as e:
            print("Error:", str(e))
            print_exc()
            self.valid = False
            self.own_error = str(e)
        else:
            self.valid = True
            self.own_error = None

        self.query = query
        self.error = error_str
```

Kognitiivisen kompleksisuuden kuvaavan alkuperäisartikkelin perusteella SonarQube näyttää olevan oikeassa. Sekä except- että else-lauseet nostavat kompleksisuutta yhdellä. Artikkelissa mainitaan else-lause vain if-lauseen seuraajana, mutta myös except-lohkon perässä se rikkoo lineaarisen vuon. (Campbell 2021) Tulos on sikäli odotettu, että sekä SonarQube että kognitiivinen kompleksisuus ovat SonarSourcen kehittämiä. Siksi analysoitaessa kognitiivista kompleksisuutta metriikkana huomioidaan vain SonarQuben tulokset.

6.2 Metriikat ja commitien määrä

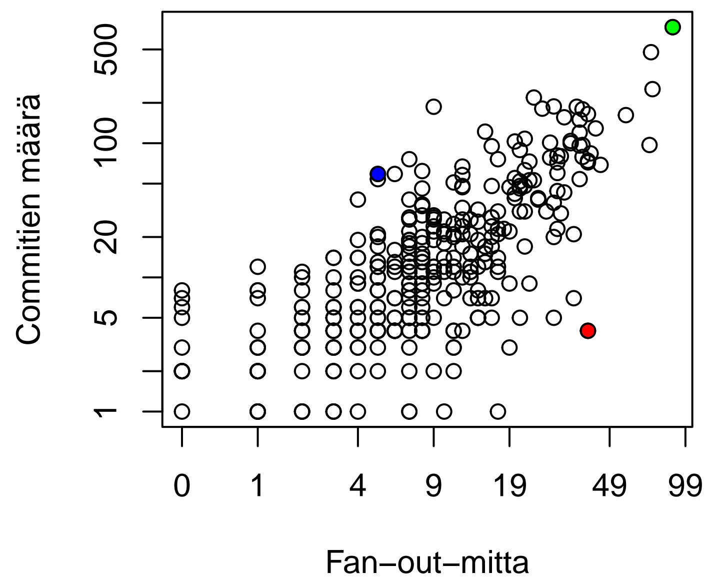
Taulukko 8 kuvaa staattisten koodimetriikoiden ja laskettujen Git-commitien määrien väliset Pearsonin ja taulukko 9 Spearmanin korrelaatiot.

Kognitiivinen kompleksisuus	0,479
McCabe	0,500
LCOM	0,292
Multimetricin McCabe	0,563
Halsteadin E	0,470
Halsteadin V	0,621
Fan-out	0,739
LOC	0,654

Taulukko 8: Pearsonin korrelaatiot mitattuun ohjelmistoyksikköön kohdistuneiden commitien määrän ja mitattujen metriikoiden välillä. Kognitiivinen kompleksisuus ja McCabe ovat funktiokohtaisia, LCOM luokkakohtainen ja muut metriikat tiedostokohtaisia metriikoita.

Kognitiivinen kompleksisuus	0,414
McCabe	0,401
LCOM	0,477
Multimetricin McCabe	0,496
Halsteadin E	0,757
Halsteadin V	0,775
Fan-out	0,711
LOC	0,779

Taulukko 9: Spearmanin korrelaatiot mitattuun ohjelmistoyksikköön kohdistuneiden commitien määrän ja mitattujen metriikoiden välillä. Korrelaatioiden taustadata on sama kuin taulukossa 8.



Kuvio 2: Tiedostojen fan-out-mittojen ja commitien määrän yhteyden kuvaava sirontakuvio. Kolme ääriesimerkkiä on korostettu omilla väreillään. Punainen ympyrä kuvaa timApp/util/error_handlers.py-tiedostoa, vihreä timApp/tim.py-tiedostoa ja sininen timApp/modules/cs/jsframe.py-tiedostoa.

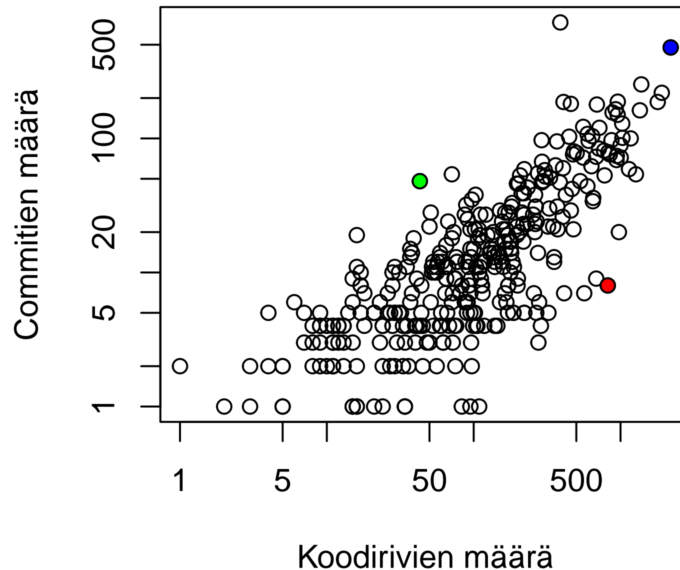
6.2.1 Fan-out

Pearsonin korrelaatiolla mitattuna ohjelmistoyksikköön kohdistuneiden muutosten määrän kanssa vahvimmin korreloi fan-out-mitta. Multimetricin dokumentaatio määrittelee fan-out-mitan olevan pelkkä "*number of imports*"(Weihmann 2023), ja ainakin tim_common-hakemistossa jokaisen tiedoston fan-out-mitta täsmää import-lauseen sisältävien rivien määrän kanssa. Fan-outin alkuperäisen määritelmän mukaan "*the fan-out of procedure A is the number of local flows from procedure A plus the number of data structures which procedure A updates*"(Henry ja Kafura 1981), joten ilmeisesti multimetric näkee tiedostot proseduureina ja vuon olevan niiden välillä, jos toisen tiedoston sisältöä käytetään suoraan toisesta käsin. Toisaalta samassa import-lauseessa voidaan tuoda periaatteessa miten monta oliota tahansa, mitä multimetric ei vaikuta noteeraavan.

Todennäköinen selitys yhteydelle on, että korkea fan-out kielii riippuvuudesta useista muista tiedostoista. Kun yhtä tiedostoa muutetaan, sen käyttötapa ulkopuolelta helposti muuttuu ja siten tiedostoa käyttävä tiedosto vanhenee. Korrelaatio puoltaa näkemystä, että eri komponenttien riippuvuuksia toisistaan tulisi minimoida. Toisaalta Spearmanin korrelaatiolla mitattuna fan-out on vasta neljänneksi vahvimmin korreloiva metriikka.

Kolme ääriesimerkkiä fan-outin ja muutostarpeen välisestä yhteydestä ovat timApp/tim.py, timApp/modules/cs/jsframe.py ja timApp/util/error_handlers.py, jotka on korostettu kuviossa 2. Ensin mainittu on sekä muokatuin että fan-outiltaan suurin tiedosto. Yhteys on sikäli looginen, että tiedoston olennaisin tarkoitus on luetteloida kaikki blueprintit ja koota ne yhteen Flask-sovellukseen. Mikä tahansa muutos blueprintien kokoonpanossa aiheuttaa siis muutostarpeen kyseisessä tiedostossa. Tiedostossa myös esimerkiksi ladataan selainpuolen skriptit osaksi selaimelle annettavaa sivua, mikä nostaa sekä tuotavien komponenttien määrää että muutostarvetta.

timApp/modules/cs/jsframe.py on ääriesimerkki pienestä fan-outista mutta suuresta muutostarpeesta. Tiedostoon ei tuoda yhdestäkään toisesta TIMin tiedostosta mitään. Tiedosto vaikuttaa erikoistuvan pelkän csPlugin-pluginin tarvitseman HTML-, JavaScript- ja CSS-koodin hallinnointiin merkkijonoina. Merkkijonojen hallinnointi ei ole sinänsä riippuvainen mistään muusta osasta TIMiä, mutta merkkijonojen manipulointi tulee helposti työlääksi.



Kuvio 3: Tiedostojen koodirivien ja commitien määrän yhteyden kuvaava sirontakuvio. Kolme ääriesimerkkiä on korostettu omilla väreillään. Punainen ympyrä kuvaa timApp/modules/cs/simcir/check/simcirtest.py-tiedostoa, sininen timApp/answer/routes.py-tiedostoa ja vihreä timApp/launch.py-tiedostoa.

Päinvastainen esimerkki on timApp/util/error_handlers.py, joka hallinnoi poikkeusten käsittelyä koko TIMissä. Suuri osa sen import-lauseista tuo pelkkiä poikkeusluokkia muualta TIMistä. Commitien määrä osoittautuu kuitenkin skriptin 5.11 virheeksi. Marraskuuhun 2022 asti tiedoston sisältö oli timApp/errorhandlers.py-tiedostossa, johon kohdistui git log-komennon perusteella 25 commitia. 30 commitin perusteella tiedosto sopisi hyvin muuhun aineistoon. Toisaalta timApp/errorhandlers.py-tiedosto oli alun perin luotu siirtämällä sisältöä timApp/tim.py-tiedostosta.

6.2.2 Koodiriven määrä (LOC)

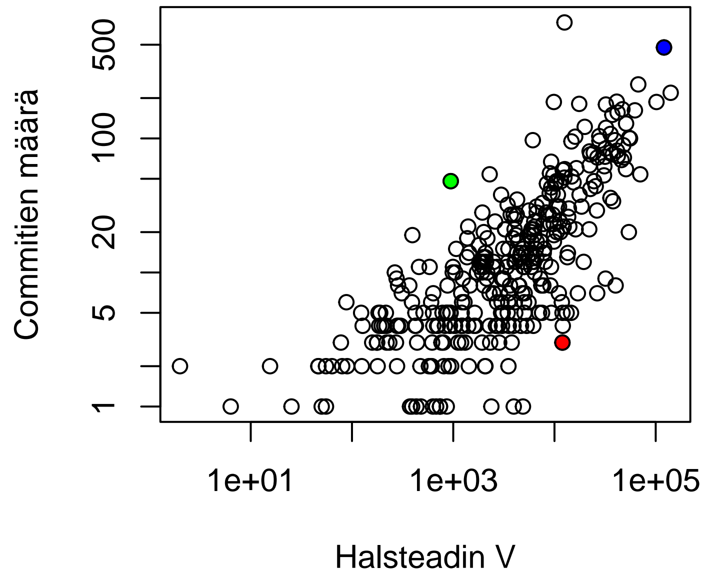
Multimetricin mittaama koodirivien määrä osoittautui olevan Spearmanin korrelaatiolla mitattuna vahvimmin ja Pearsonin korrelaatiolla toiseksi vahvimmin yhteydessä muutosten

määrään. Yhteys vaikuttaa ehkä triviaalilta, mutta voidaan perustella teoreettisesti seuraavasti: jos odotetaan, että jokaisen rivin todennäköisyys tulla muokatuksi aikayksikössä on p kontekstista riippumatta, tiedoston todennäköisyys tulla muokatuksi aikayksikössä on binomijakauman nojalla $1-(1-p)^{LOC}$. Odotusarvoisesti muokkausten määrä on saman jakauman nojalla $p * LOC$ ja yhteyden tulisi olla siis lineaarinen. Toisaalta Spearmanin korrelaatio on Pearsonia jonkin verran vahvempi, vaikka vain jälkimmäinen mittaa nimenomaan lineaarista yhteyttä. Kolme ääriesimerkkiä on korostettu kuviossa 3.

Ääriesimerkki suuresta koodirivien ja commitien määrästä yhdessä tiedostossa on timApp/answer/routes.py, joka määrittelee TIMissä annettavien tehtävien vastauksia hallinnoivan answers-blueprintin. Tiedosto on TIMin koodirivimäärältään suurin ja toiseksi muokatuin Python-tiedosto. Kaikki blueprintin 26 reititystä on määritelty yhdessä tiedostossa ja siinä on myös joitakin apufunktioita vastausten hallinnointiin. Vastausten käsittely vaikuttaa selvästi mutkikkaalta prosessilta. Manuaalisen tutkimuksen perusteella tiedosto oli aiemmin eri polussa ja siihen on silloin kohdistunut 134 commitia lisää eli yhteys koodirivien ja commitien määrän välillä vaikuttaa todellisuudessa hieman vahvemmalta.

Esimerkki suuresta koodirivien määrästä ja pienestä commitien määrästä on timApp/modules/cs/simcir/check/simcirtest.py, joka testaa virtapiirejä simuloivan simcir-pluginin toiminnallisuutta. Itse simcir on HTML:llä, CSS:llä ja JavaScriptillä toteutettu, mutta testiskripti on Pythonia. Koska itse simcir-komponentti on avointa lähdekoodia ja siten ulkopuolelta määritelty, on ilmeisesti myös kiistatonta, miten testit on pitänyt toteuttaa, vaikka ne vievätkin paljon koodirivejä. git log –follow-komennon perusteella tiedostoon on kohdistunut 12 eikä 8 commitia, mutta senkin perusteella tiedosto erottuu joukosta.

Päinvastainen esimerkki on koko TIM-sovelluksen käynnistävä timApp/launch.py-tiedosto. Koodirivejä on siinä vain kymmeniä, mutta se määrittelee TIMin palvelimen käynnistyksen yksityiskohdat, kuten millä ehdoilla käytetään Gunicorn-palvelinrajapintaa. Sinänsä käynnistyksen logiikka ei ole niin mutkikasta, että se vaatisi pitkää skriptiä, mutta sitä koskevat vaatimukset ilmeisesti vain muuttuvat usein. git log –follow-komennon perusteella tiedostoon on kohdistunut 54 eikä 48 commitia, vaikka palvelimen käynnitysskripti on ollut TIM-kehityksen alkuajoista lähtien samassa polussa.

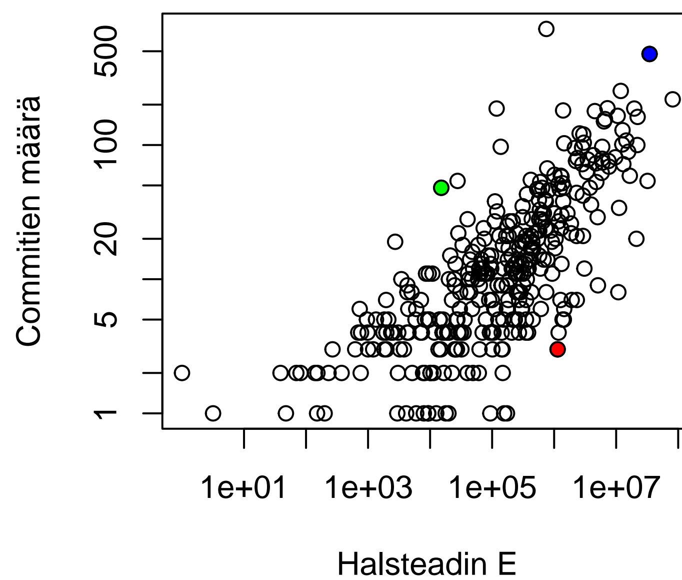


Kuvio 4: Tiedostojen Halsteadin V-mitan ja commitien määrän yhteyden kuvaava sirontakuvio. Kolme ääriesimerkkiä on korostettu omilla väreillään. Vihreä ja sininen ympyrä kuvaavat samoja tiedostoja kuin kuviossa 3. Punainen ympyrä kuvaa $\mathrm{timApp/plugin/userselect/action_queue.py}$-tiedostoa.

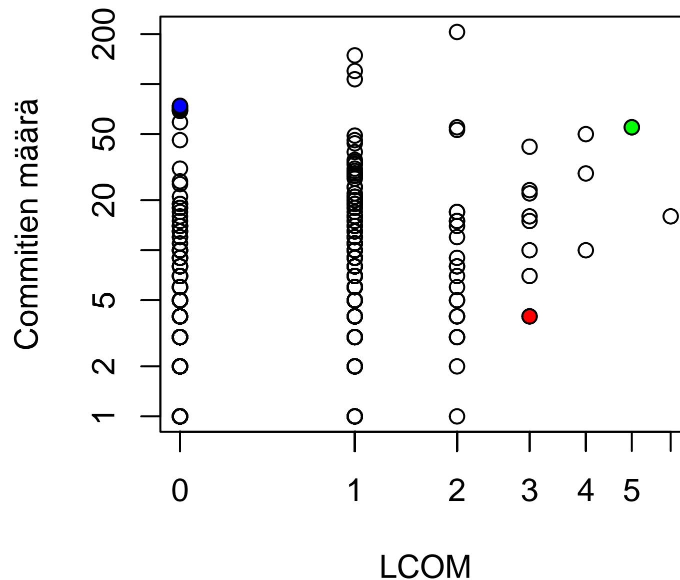
6.2.3 Halsteadin metriikat

Spearmanin korrelaatiot sekä Halsteadin E-että V-muuttujien ja muutosten määrän välillä osoittautuivat huomattaviksi, mutta Pearsonin korrelaatiot ovat selvästi heikompia.

Multimetricin PyPI-sivu ei kuvaa tarkkaan, miten Halsteadin perusmetriikat on määritelty Pythonille (Weihmann 2023). V-mitta vaikuttaa kuvion 4 sekä Spearmanin korrelaatiokertoimen perusteella kuitekin olevan vahvasti yhteydessä koodirivien määrään, mikä sopii hyvin V:n määritelmään. E-mitta vaikuttaa kuvion 5 perusteella olevan hyvin vahvasti yhteydessä V-mittaan, joskin se jakautuu suuremmalle lukuvälille. Tämä viittaa siihen, että mitatut V-arvot ovat keskimäärin selvästi suurempia kuin mitatut V*-arvot, koska jos $V = V*$, määritelmän perusteella $E = V$. Multimetric-työkalu siis arvioi, että suurin osa TIMin lähdekoodista olisi ainakin teoriassa toteutettavissa vähäisemmällä koodimäärällä. (Shen, Conte



Kuvio 5: Tiedostojen Halsteadin E-mitan ja commitien määrän yhteyden kuvaava sirontakuvio. Kolme ääriesimerkkiä ovat samat kuin kuviossa 4 ja ne on korostettu samoilla väreillä.



Kuvio 6: Luokkien metodien koheesion puutteen (LCOM) ja commitien määrän yhteyden kuvaava sirontakuvio. Sinisellä on korostettu timApp/velp/annotations.py-tiedoston AnnotationVisibility-luokka, vihreällä timApp/document/docinfo.py-tiedoston DocInfo-luokka ja punaisella timApp/sisu/parse_display_name.py-tiedoston SisuDisplayName-luokka.

ja Dunsmore 1983)

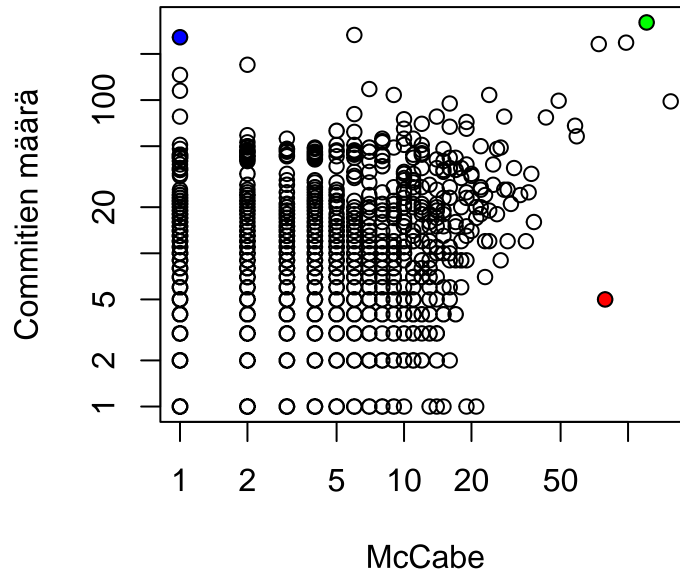
Sekä *V*- että *E*-mittojen yhteyttä commitien määrään kuvaavissa sirontakuvioissa korostuvat samat ääriesimerkit kuin koodirivien määrän. Poikkeuksena on timApp/plugin/userselect/action_queue.py-tiedosto, jolla on suuret *V*- ja *E*-mitat mutta siihen nähden vähän commiteja. git log –follow-komennon perusteella skripti 5.11 laski commitit oikein. Tiedosto määrittelee userselect-pluginin sisäisen tapahtumien rekisteröinnin ja käsittelyn logiikan, joka vaikuttaa alusta lähtien hyvin määritellyltä prosessilta.

6.2.4 Metodien koheesion puute (LCOM)

Metodien koheesion puutteen (LCOM) yhteys muutosten määrään on Pearsonin korrelaatiolla mitaten heikoin. Lievä positiivinen yhteys on huomattavissa, mutta kuvion 6 perusteella lähinnä vain vähiten koheesien luokkien vuoksi. Ääriesimerkkinä vähän koheesista ja paljon muokatusta luokasta on timApp/document/docinfo.py-tiedoston DocInfo-luokka, joka on dokumenttia kuvaavan DocEntry- ja käännettyä dokumenttia kuvaavan Translation-luokan perusluokka. Kaikki kolme käsittelevät dokumentin metatietoja, kuten viimeisintä muokkausajankohtaa tai polkua. Vaikka loogisesti metatiedot ovat kaikki dokumenttikohtaisia, suurimman osan funktioista ei tarvitse käsitellä niitä kaikkia.

Esimerkki vähän koheesista ja vähän muokatusta luokasta on timApp/sisu/parse_display_name.py-tiedoston SisuDisplayName-luokka. Sitä käytetään ainoastaan Sisu-järjestelmästä haetun merkkijonomuotoisen kurssidatan muotoiluun opettajille ja hallintohenkilöille lähetettävää sähköpostia varten. Data järjestetään luokassa useisiin attribuutteihin, kuten kurssikoodiin, päivämääräväliin ja kuvaukseen. Kaikki funktiot ovat @property-funktioita ja ne kuvaavat vain merkkijonomuotoisen, muista attribuuteista johdetun attribuutin. Sisusta saatu merkkijonodata ja sen käyttötarkoitus ovat ilmeisesti vain niin pysyvästi määriteltyjä, ettei kerran toteutettua luokkaa ole jälkikäteen tarvinnut muuttaa. Manuaalisesti git-lokia tutkimalla selviää, että luokka on aiemmin ollut eri tiedostossa ja siihen on silloin kohdistunut 3 commitia, joita automaattinen skripti ei huomannut. Siitä huolimatta luokkaa on metriikkaan nähden melko vähän muokattu.

Päinvastainen esimerkki on timApp/velp/annotations.py-tiedoston AnnotationVisibility-luokka. Luokassa ei ole funktioita vaan se on vain esimerkki Pythonin enum-standardikirjaston Enum-luokan perimisestä eikä siten ole oikeastaan varsinainen luokka (P. S. Foundation 2023a). Dokumentteihin lisättävillä, annotation-nimellä kutsuttavilla sivuhuomautuksilla voi olla neljä näkyvyystasoa, joista jokaista vastaa yksi kokonaisluku. Kokonaislukua käytetään esimerkiksi HTTP-kutsuissa ja tietokannassa, mutta ilmeisesti kehittäjille on mielekkäämpää yhdistää kokonaislukuihin niitä kuvaava muuttujannimi. Sinänsä luokka on siis yksinkertainen. Manuaalisen git log-komennon käytön perusteella suuri commitien määrä vaikuttaa Gitin bugilta. Suurin osa sen luettelemista commiteista ei käsittele AnnotationVisibility-luokkaa.

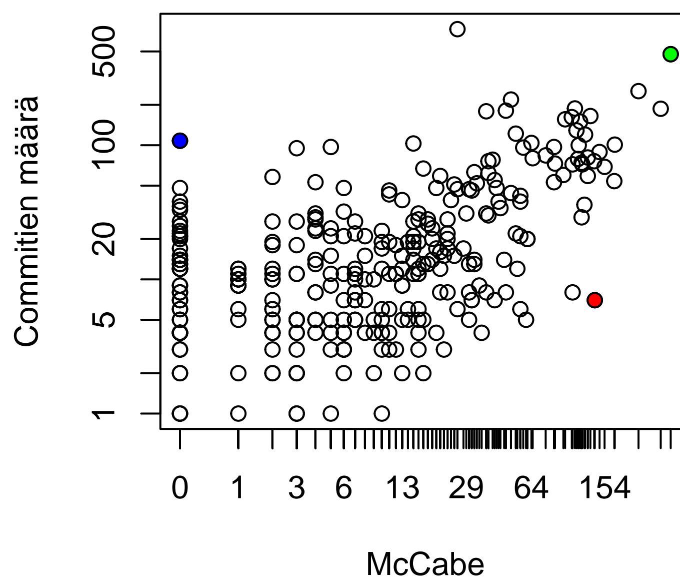


Kuvio 7: SonarQuben mittaamien funktioiden McCabe-kompleksisuuksien ja commitien määrän yhteyden kuvaava sirontakuvio. Punaisella on korostettu timApp/plugin/jsrunner/util.py-tiedoston save_fields-funktio, vihreällä timApp/item/routes.py-tiedoston render_doc_view-funktio ja sinisellä timApp/tim.py-tiedoston start_app-funktio.

6.2.5 McCabe

McCabe-kompleksisuuden yhteys muutosten määrään on sekä Pearsonin että Spearmanin korrelaatioilla ja sekä SonarQubella että multimetricillä mitaten kohtalainen. Pearsonin korrelaatiot ovat hiukan vahvemmat. Sekä SonarQuben että multimetricin tuloksia kuvaavista sirontakuvioista paljastuu kuitenkin, että kompleksisuuden ylittäessä noin 15 yhteys muutosten määrään nousee hyvin vahvaksi. Pienemmillä kompleksisuuksilla yhteyttä ei näytä olevan suuntaan eikä toiseen.

Merkittävä poikkeus on timApp/plugin/jsrunner/util.py-tiedosto ja sen save_fields-funktio, joka on koko tutkimusalueen neljänneksi kompleksisin funktio mutta joka on skriptin 5.11 perusteella tehty vain viidellä commitilla. Toisaalta manuaalisella tutkimuksella selviää, että automaattiskriptiltä jäi huomaamatta funktion ja koko tiedoston historia. Aiemmin



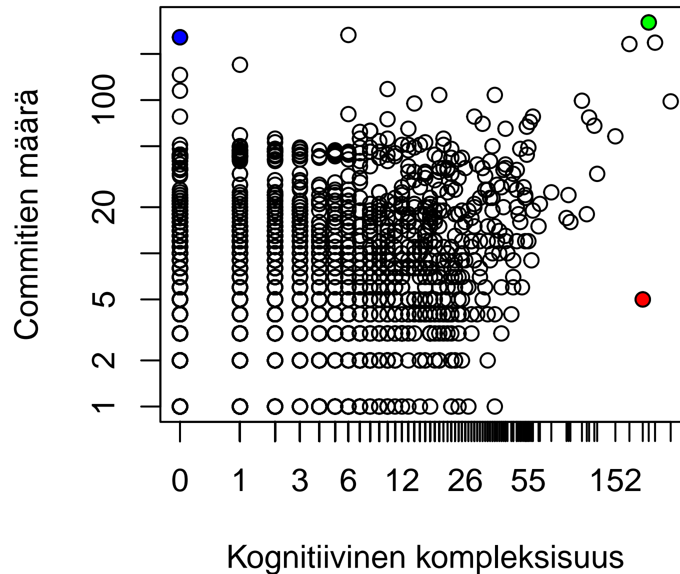
Kuvio 8: Multimetricin mittaamien tiedostojen McCabe-kompleksisuuksien ja commitien määrän yhteyden kuvaava sirontakuvio. Punaisella on korostettu timApp/plugin/jsrunner/util.py, vihreällä timApp/answer/routes.py ja sinisellä timApp/document/docsettings.py.

funktio oli timApp/util/jsrunner_util.py-nimisessä tiedostossa, jolloin siihen kohdistui 2 commitia, ja sitä ennen timApp/answer/routes.py-tiedostossa, jolloin siihen kohdistui satoja commiteja.

Ääriesimerkki kompleksisesta ja muokatusta funktiosta on timApp/item/routes.py-tiedoston render_doc_view-funktio, joka muuntaa DocInfo-luokalla kuvatun dokumentin selaimelle annettavaksi HTML-sivuksi. Funktion kompleksisuus syntyy useista dokumentin näyttämisen yksityiskohdista, kuten että näytetäänkö koko dokumentti vai vain tietty lohköväli tai mitä dokumentista on ajantasaisena välimuistissa. Vaikuttaa ymmärrettävältä, että näin keskeinen funktio kasvaa mutkikkaaksi ja sen toiminta tulee riippuvaiseksi monista muista osista. HTML-sivuksi dokumentin muuntava funktio oli alun perin timApp/tim.py-tiedostossa eikä skripti huomioinut sen commiteja, mutta funktio siirrettiin omaan tiedostoonsa jo TIMin kehityksen alkuaikoina 2014. Dokumentin näyttäminen on silti muuttunut niin paljon, että on kyseenalaista, voiko skriptiltä laskematta jääneet commitit laskea nykyisen render_doc_view-funktion commiteiksi.

Ääriesimerkki muokatusta mutta vähän kompleksisesta funktiosta on puolestaan timApp/-tim.py-tiedoston start_app-funktio, joka käynnistää koko TIMiä hallinnoivan Flask-sovelluksen. Funktiossa määrätään esimerkiksi IP-osoiteavaruus, josta sovellus on saavutettavissa, sekä pääkontin TCP-porttinumero. Funktiota kutsutaan ainoastaan timApp/launch.py-tiedostosta. Funktion McCabe-kompleksisuus on vain 1, mutta se on koko tutkimusalueen kolmanneksi muutetuin funktio. Muutostarpeen taustalla on oletettavasti samanlainen yksityisikohtien muuntelu kuin timApp/launch.py-tiedostossa. git log-komennolta jäi huomaamatta TIM-kehityksen alkuajoilta yksi commit, jossa Flask-sovellus ensi kerran käynnistettiin, mutta yhdellä commitilla ei ole muutenkin näin muokatussa yksikössä merkitystä.

Suuret funktiotason McCabe-kompleksisuudet saavat myös vastaavat, multimetricilla mitatut tiedostojen McCabe-kompleksisuudet korostumaan. Esimerkiksi timApp/plugin/jsrunner/util.py-tiedoston kompleksisuudesta yli puolet selittyy save_fields-funktiolla. Kompleksisin tiedosto on multimetricin mukaan timApp/answer/routes.py, mutta mutkikkaimman funktion sisältävä timApp/modules/cs/cs.py on heti toiseksi kompleksisin. Kuviossa 8 on havaittavissa sama kuin kuviossa 7: kun kompleksisuus on alle noin 15, yhteyttä sen ja muutosten määrän välillä ei ole nähtävissä, mutta muuten yhteys on hyvin selvä.



Kuvio 9: SonarQuben mittaamien kognitiivisten kompleksisuuksien ja commitien määrän yhteyden kuvaava sirontakuvio. Punaisella, vihreällä ja sinisellä korostetut funktiot ovat samat kuin kuviossa 7.

Kuviossa 8 sinisellä korostettu tiedosto vaikuttaa multimetricin bugilta. def-määreiden perusteella tiedostossa on yli sata funktiota, joten nolla ei ole edes teoriassa mahdollinen McCabe-kompleksisuuksien summa.

6.2.6 Kognitiivinen kompleksisuus

Kognitiivinen kompleksisuus on hyvin vahvasti yhteydessä McCabe-kompleksisuuteen. Kuviossa 9 väreillä korostetut funktiot ovat samat kuin kuviossa 7, ja muutenkin kuviot näyttävät samanlaisilta. Noin arvoon 30 asti yhteyttä ei näytä olevan, mutta sen jälkeen se on selvästi tunnistettavissa. Kognitiivinen kompleksisuus myös jakautuu laajemmalle alueelle.

Kognitiivisen kompleksisuuden ei sen kuvaavan alkuperäisartikkelin perusteella ole tarkoitus olla yleisesti suurempi kuin McCabe-kompleksisuuden, mutta perussääntö 3 vaikuttaa

hyvältä selitykseltä, miksi TIMin kohdalla siltä näyttää. McCabe-kompleksisuus nousee yhdellä jokaista lineaarisen vuon rikkomista kohti riippumatta kontekstista, mutta kognitiivinen kompleksisuus nostaa sitä samalla yhdellä jokaista sisennystä kohti. Jos koodissa on siis paljon sisennyksiä, useimmat `if`-lauseet tai silmukat nostavat kompleksisuutta yli yhdellä, mutta McCabe-kompleksisuutta aina vain yhdellä.

6.3 Johtopäätökset

Kaikki mitattujen metriikoiden ja laskettujen commitien määrien väliset korrelaatiot ovat merkittäviä.

Fan-out-mitan yhteys commitien määrään näyttää superlineaariselta: kun fan-out on 0 tai 1, commiteja on aina alle 20. Jos se on 10, tyypillinen commitien määrä on myös noin 10. Toisaalta kun fan-out lähestyy sataa, commitien määrä on jo useissa sadoissa. Yksi import-lause lisää siis nostaa muutosten määrää todennäköisesti sitä enemmän, mitä enemmän import-lauseita tiedostossa on ennestään. Tämä tukee näkemystä, että tiedostojen välisiä riippuvuuksia tulisi minimoida. Toisaalta riippuvuuksien tarve yksittäisessä tiedostossa voi olla myös tietoinen arkkitehtuuripäätös. Ainakin timApp/tim.py-tiedoston suuri riippuvuuksien määrä on väistämätön seuraus siitä, että sovellus on jaettu useisiin blueprinteihin, jotka sisällytetään samaan Flask-sovellukseen. Toisaalta näyttää siltä, että ainakin teoriassa muutostarvetta olisi vähennettävissä lisäämällä Flask-sovelluksen jakoon blueprinteihin useita hierarkiakerroksia. timApp/tim.py-tiedostoon tuodut blueprintit siis ryhmiteltäisiin loogisesti ja jokaisen ryhmän blueprintit tuotaisiin vain ryhmänsä yhdeksi blueprintiksi yhdistävään omaan tiedostoonsa, ja vain yhdistetty blueprint tuotaisiin timApp/tim.py-tiedostoon.

Koodirivien määrän ja commitien määrän yhteys puolestaan vaikuttaa hyvin lineaariselta. Jos rivejä on noin 50, commiteja on keskimäärin noin 5, ja jos niitä on noin 500, commiteja on keskimäärin noin 50. Koodirivin todennäköisyys muuttua ei siis riipu siitä, montako muuta koodiriviä tiedostossa on.

Myös Halsteadin V:n yhteys commitien määrään näyttää hyvin lineaariselta. Jos V on 1000, tyypillinen commitien määrä on 5, ja jos se on noin satakertainen, on myös commitien määrä tyypillisesti noin satakertainen. Toisaalta määritelmän perusteella yhteyden pitäisi olla

hieman superlineaarinen. Elleivät tiedostojen a ja b käyttämät operaattorit ja operandit ole täysin samat, niiden yhdistäminen yhdeksi tiedostoksi nostaa V:n suuremmaksi kuin a- ja b-tiedostojen V:iden summa. Ei ole selvää, miten Multimetric määritteli operaattorit ja operandit Pythonille, mutta yhteyden perusteella TIMissä ei ole perusteetta toteutettu eri asioita samoissa tiedostoissa.

Halsteadin E sen sijaan vaikuttaa hieman superlineaariselta. Kun E on noin 1000, on commiteja keskimäärin noin 3, satakertaisena noin 10 ja 10 000 -kertaisena noin 100. Koska ei ole selvää, miten E:n laskussa tarvittava V^* laskettiin, kovin tarkkaa spekulaatiota superlineaarisuuden syystä ei voi tehdä. Jos V^*:n oletetaan olevan suoraan verrannollinen V:hen eli jokainen tiedosto olisi toteutettavissa yhtä monta prosenttia toteutettua pienemmällä koodimäärällä, myös E:n tulisi olla suoraan verrannollinen V:hen eli V^* näyttää kertovan TIMin kehitysprosessista jotakin merkittävää. TIM siis tukee hieman väitettä, että Halsteadin E-metriikka mittaa tosiaan tiedoston vaatimaa vaivaa (englanniksi *effort*), josta E on saanut nimensä (Shen, Conte ja Dunsmore 1983).

Metodien koheesion puute vaikuttaa käytetyistä metriikoista turhimmalta. Ensinnäkin se jakautuu metriikoista pienimmälle lukuvälille. Toisekseen etenkin Pearsonin korrelaatio commitien määrän kanssa ei ole niin vahva, ja yhteys ilmenee lähinnä vain, jos LCOM-mitta on vähintään 3. Suurimmalla osalla luokista LCOM on alle sen. Metriikan kuvaavassa artikkelissa vaikuttaa olevan taustalla oletus, että mitattava sovellus noudattaa tarkkaan oliosuuntautunutta paradigmaa, jossa jokainen luokka sekä kokoaa tietoa että manipuloi sitä funktioissa (Chidamber ja Kemerer 1994). TIMissä huomattava osa luokista kuitenkin vain kokoaa tietoa loogisiin kokonaisuuksiin. Esimerkiksi ääriesimerkkinä kuvattu SisuDisplayName-luokka vain muodostaa merkkijonosta luetusta datasta joukon attribuutteja.

McCabe vaikuttaa hyödylliseltä metriikalta, kun se on noin yli 15: sitä pienemmillä arvoilla sillä ja funktioiden muutostarpella ei näy merkittävää yhteyttä, mutta suuremmilla arvoilla yhteys näyttää hyvin selvältä. SonarSourcen mielestäkin luvussa 15 on perää: ainakin aineiston keruussa käytetty SonarQuben versio huomauttaa liian suuresta McCabe-kompleksisuudesta oletuksena, jos se on yli 15. SonarQuben dokumentaatio ei perustele arvoa 15 millään eikä kirjallisuuskartoituksessa löytynyt millekään tietylle rajalle tukea, mutta TIMin perusteella arvossa 15 on perää.

Myös kognitiivisen kompleksisuuden suurin varoittamaton arvo on SonarQubessa oletuksena 15. Aineiston perusteella yhteys sen ja commitien määrän välillä muuttuu selväksi kuitenkin vasta, kun se on noin yli 25. Yleisesti ottaen se saa TIMissä huomattavasti McCabea suurempia arvoja. Vaikka sen tarkoitus on korjata McCabe-kompleksisuuden heikoimpina pidettyjä piirteitä, se vaikuttaa yhtä informatiiviselta. Sekä Pearsonin että Spearmanin korrelaatiot niiden ja commitien määrän välillä ovat hyvin lähellä toisiaan.

Toisaalta McCabe- ja kognitiivisen kompleksisuuden ja commitien määrien välillä havaitut yhteydet eivät ehdottomasti puolla näkemystä, ettei yksittäisen funktion kompleksisuus saisi olla yli tietyn raja-arvon. Vaikka suurilla arvoilla yhteydet ovat selviä, ne näyttävät lineaarisilta. McCabe-kompleksisuudeltaan mutkikkaan funktion jakaminen useisiin, vähemmän kompleksisiin funktioihin kuitenkin nostaa kompleksisuuksien summan alkuperäistä kompleksisuutta suuremmaksi, koska se nostaa yhdistettyjen komponenttien määrää. Toisaalta muuten kompleksisen funktion jakaminen pienempiin funktioihin ei vaikuta kompleksisuuteen, joten yhteyden kompleksisuuden ja commitien määrän välillä tulisi olla superlineaarinen, jotta kompleksisten funktioiden välttäminen olisi perusteltua.

Kognitiivisten kompleksisuuksien summa sen sijaan pienenee, kun mutkikas funktio jaetaan osiin, jos siinä on sisäkkäisiä lineaarisen vuon rikkomisia. Kolmannen perussäännön nojalla esimerkiksi kaksi sisäkkäistä for-silmukkaa nostavat kompleksisuutta kolmella, mutta jos sisempi silmukka eristetään omaan funktioonsa, kompleksisuuksien summa on vain kaksi. Koska kokonaisuuksien yhdistäminen suuriin funktioihin siis nostaa kognitiivista kompleksisuutta superlineaarisesti, yhteyden sen ja commitien määrän välillä pitäisi myös olla superlineaarinen, jotta ennustettua commitien kokonaismäärää voisi vähentää kompleksisia funktioita jakamalla.

Suuri McCabe- tai kognitiivinen kompleksisuus siis kielii, että funktio vaatii erityistä huomiota, mutta funktion jakamisesta pienempiin kokonaisuuksiin ei näytä olevan erityistä hyötyä. Toisaalta niitä voi usein laskea funktiota refaktoroimalla, jolloin odotettavasti funktion ymmärrettävyys paranee ja virheiden riski ja niistä syntyvä muutostarve vähenevät.

7 Yhteenveto

Staattisia koodimetriikoita on tutkittu noin 50 vuotta, mutta täyttä konsensusta niiden hyödyllisyydestä ei ole saavutettu. Aiempi tutkimus vaikuttaa osittain ristiriitaiselta myös tämän pro gradu -tutkielman tulosten kanssa, vaikka yhtäläisyyksiäkin on. Kirjallisuuskartoituksen perusteella parhaiten validoitu, tässä pro gradu -tutkielmassa testattu metriikka on metodien koheesion puute, mutta aineiston analyysissä se vaikutti vähiten validilta. Toisaalta metriikat on testattu TIMissä, joka ei noudata oliosuuntautunutta ohjelmointiparadigmaa kovin täydellisesti vaan useat luokat ovat pelkkiä tapoja kapseloida tietoa. Fan-out, McCabe-kompleksisuus, koodirivien määrä ja Halsteadin V on myös validoitu joissakin aiemmissa tutkimuksissa, ja ne vaikuttavat melko valideilta myös tämän pro gradu -tutkielman perusteella. Toisaalta metriikan validiudelle ei ole kirjallisuudessa yksiselitteistä määritelmää, ja tässä pro gradu -tutkielmassa validiutta testattiin vain vertaamalla metriikoita Git-lokista löytyvien muutosten määrään.

Vaikka kognitiivinen kompleksisuus on tarkoitettu korjaamaan McCabe-kompleksisuuden puutteita, se vaikuttaa yhtä hyödylliseltä kuin McCabe-kompleksisuus. Pienillä arvoilla kummallakaan ei ole yhteyttä muutosten määrään, mutta suurilla arvoilla on tunnistettavissa lineaarinen yhteys. Suuret arvot kielivät, että funktioon on syytä kiinnittää huomiota, mutta pelkästään suuren kompleksisuuden välttämiseksi ei kannata jakaa funktiota pienempiin funktioihin. Kognitiivisen kompleksisuuden on tutkittu olevan jokseenkin yhteydessä ymmärrettävyyteen (Muñoz Barón, Wyrich ja Wagner 2020) ja tämän pro gradu -tutkielman perusteella yhteys on odotettava, mutta on odotettavaa, että yhteys McCabe-kompleksisuuteen on samanlainen.

Hyödyllisimmiltä metriikoilta tämän pro gradu -tutkielman perusteella vaikuttavat fan-out ja Halstadin E. Kummankin yhteys tiedoston muutosten määrään on superlineaarinen eli tiedostojen muutosten odotettua kokonaismäärää voi pienentää jakamalla tiedostoja eri osiin niin, että fan-out ja Halsteadin E pienenevät. Fan-out-mitta on validoitu joissakin aiemmissa tutkimuksissakin, mutta Halsteadin E-mittaa ei kirjallisuuskartoituksen perusteella ole juuri tutkittu. Syy on ehkä, että E:n mittaamiseen vaadittua V*-mittaa ei pysty yleensä yksiselitteisesti määritelemään. Myöskään multimetricin dokumentaatio ei kuvaa, miten Halsteadin

E määritellään, mutta yhteys sen mittaaman E-mitan ja muutosten määrän välillä on niin vahva, että E-mitassa vaikuttaa olevan perää.

Koska sekä SonarQuben aineiston keruussa käytetty ilmaisversio että PyPI-projektit ovat avointa lähdekoodia, niissä käytetyt tarkat tavat mitata koodia selviävät lähdekoodia tutkimalla. Jatkotutkimus voisi esimerkiksi selvittää, miten multimetric tarkalleen määritteli operaattorin ja operandin Pythonille ja miten se mittasi Halsteadin E-metriikan. Fan-out-mitta puolestaan multimetricin mittauksissa vaikutti tarkoittavan pelkkää import-lauseen sisältävien rivien määrää, mutta tarkka vastaus selviäisi lähdekoodia tutkimalla.

Pelkkä Git-lokista löytyvien commitien määrä on melko yksipuolinen tapa mitata koodin muutostarvetta tai ongelmallisuutta ylipäätään. Ensinnäkin skripti 5.11 osoittautui puutteeliseksi. Tiedostokohtaisia commiteja laskeva git log -komento ei huomioi tiedostojen siirtämisiä eikä uudelleennimeämisiä, vaikka dokumentaation perusteella ne olisi huomioitavissa –follow-parametrillä (Git-yhteisö, n.d.). Toisekseen vaikka funktio- ja luokkakohtaisia commiteja laskevat git log -komennot huomioivat ne periaatteessa, siltäkin jäi usein huomaamatta tutkittavan koodirivivälin aiemmin toisessa tiedostossa ollut historia. Kolmanneksi etenkin isoja kokonaisuuksia toteutettaessa commitit on usein yhdistetty yhdeksi käyttämällä squash merge -yhdistämistä yhdistettäessä kokonaisuus TIMin Git-päähaaraan. Toisaalta puutteiden aiheuttamasta kohinasta huolimatta selviä yhteyksiä metriikoiden ja commitien määrän välillä on havaittavissa, joten jokseenkin validoidut metriikat vaikuttavat sitäkin validimmilta. Lisäksi tutkittaessa ääriesimerkkejä manuaalisesti vaikutti, että skripti jätti huomiotta olennaisia commiteja eniten, jos laskettu commitien määrä oli mitattuun metriikkaan nähden pieni.

Toinen tapa mitata ohjelmakoodin ongelmallisuutta olisi haastatella TIMin kehitysyhteisöä. Toisaalta käsitykset ongelmallisuudesta ovat hyvin subjektiivisia. Jos jokaiselta kehittäjältä kysyttäisiin mielipidettä jokaisesta funktiosta, luokasta ja tiedostosta, vastaamiseen menisi huomattavasti aikaa eikä luultavasti suurimmalla osalla kehittäjistä ole kokemusta kuin tietystä osasta TIMiä. Toisaalta Git-lokiin tallentuvat kehittäjien itse ilmoittamat nimet ja sähköpostiosoitteet, joiden perusteella kyselyt voi kohdentaa. Eräs tapa olisi myös kysyä mielipidettä vain ääriesimerkeistä metriikoiden tai commitien määrien perusteella.

Vaikka staattisten koodimetriikoiden validiutta on tutkittu vuosikymmeniä, tiettyjen ohjelmistojen kontekstissa sitä ei vaikuta olevan juuri tutkittu. Tämän pro gradu -tutkielma jokseenkin tukee aiempaa tutkimusta valideiksi havaituista metriikoista, joskin metodien koheesion puutteen validius vaikuttaa TIMissä melko heikolta aiempaan tutkimukseen verrattuna. Yhteys metriikan ja muiden tietyn ohjelmiston piirteiden välillä voi kuitenkin kertoa yhtä hyvin metriikasta kuin ohjelmistosta.

Lähteet

Alves, Vander, Nan Niu, Carina Alves ja George Valença. 2010. "Requirements engineering for software product lines: A systematic literature review". *Information and Software Technology* 52 (8): 806–820.

Arvanitou, Elvira Maria, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Matthias Galster ja Paris Avgeriou. 2017. "A mapping study on design-time quality attributes and metrics". *Journal of Systems and Software* 127:52–77.

Bansiya, Jagdish, ja Carl G. Davis. 2002. "A hierarchical model for object-oriented design quality assessment". *IEEE Transactions on software engineering* 28 (1): 4–17.

Batchelder, Ned. 2023. *mccabe - PyPI.* https://pypi.org/project/mccabe/. Haettu 4.10.2023.

Bucci, Giacomo, ja Paolo Nesi. 1995. "Using TOOMS/TROL for specifying a cellular phone". Teoksessa *Proceedings Seventh Euromicro Workshop on Real-Time Systems,* 49–56. IEEE.

Campbell, G. Ann. 2021. *Cognitive Complexity - a new way of measuring understandability.* 1.5. SonarSource S.A., HuhtikuuHuhtikuu.

Campwood Software. 2023a. *Campwood Software, publisher of SourceMonitor.* https://www.campwoods
Haettu 29.9.2023.

———. 2023b. *SourceMonitor.* https://www.derpaul.net/SourceMonitor/. Haettu 29.9.2023.

Cant, SN, D Ross Jeffery ja Brian Henderson-Sellers. 1995. "A conceptual model of cognitive complexity of elements of the programming process". *Information and Software Technology* 37 (7): 351–362.

Chidamber, Shyam R, ja Chris F Kemerer. 1994. "A metrics suite for object oriented design". *IEEE Transactions on software engineering* 20 (6): 476–493.

CoderGears. 2023a. *CppDepend - Boost Your C and C++ Code Quality.* https://www.cppdepend.com/. Haettu 22.9.2023.

CoderGears. 2023b. *CppDepend Pricing: Get your static analysis tool Today!* https://www.cppdepend.co Haettu 27.9.2023.

Coskun, Erman, ja Martha Grabowski. 2001. “An interdisciplinary model of complexity in embedded intelligent real-time systems”. *Information and Software Technology* 43 (9): 527–537.

Duran, Rodrigo, Juha Sorva ja Sofia Leite. 2018. “Towards an analysis of program complexity from a cognitive perspective”. Teoksessa *Proceedings of the 2018 ACM Conference on International Computing Education Research,* 21–30.

Foundation, Eclipse. 2023. *Search | Eclipse Plugins, Bundles and Products - Eclipse Marketplace.* https://marketplace.eclipse.org/search/site. Haettu 29.9.2023.

Foundation, Python Software. 2023a. *enum — Support for enumerations — Python 3.12.0 documentation.* https://docs.python.org/3/library/enum.html. Haettu 6.12.2023.

———. 2023b. *Help - PyPI.* https://pypi.org/help/packages. Haettu 5.10.2023.

———. 2023c. *PyPI - The Python Package Index.* https://pypi.org/. Haettu 13.10.2023.

Git-yhteisö. n.d. *Git - git-log Documentation.* https://git-scm.com/docs/git-log.

Gray, David, David Bowes, Neil Davey, Yi Sun ja Bruce Christianson. 2009. “Using the Support Vector Machine as a Classification Method for Software Defect Prediction with Static Code Metrics”.

Henry, Sallie, ja Dennis Kafura. 1981. “Software structure metrics based on information flow”. *IEEE transactions on Software Engineering,* numero 5, 510–518.

Jabangwe, Ronald, Jürgen Börstler, Darja Šmite ja Claes Wohlin. 2015. “Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review”. *Empirical Software Engineering* 20:640–693.

JYU. 2023a. *GitHub - TIM-JYU/TIM: TIM (The Interactive Material) is an open-source cloud-based platform for creating interactive learning documents.* https://github.com/TIM-JYU/TIM/. Haettu 4.8.2023-20.10.2023.

JYU. 2023b. *Johdatus TIMin kehitykseen – TIM.* https://tim.jyu.fi/view/tim/TIMin-kehitys/Johdatus-TIMin-kehitykseen#Fktkq1eN4prB. Haettu 7.8.2023.

———. 2023c. *Palveludokumentaatio - TIM.* https://tim.jyu.fi/view/tim/TIMin-kehitys/palveludokumentaatio. Haettu 11.9.2023.

———. 2023d. *The Interactive Material.* https://github.com/orgs/TIM-JYU/repositories. Haettu 20.10.2023.

Kitchenham, Barbara. 2004. "Procedures for performing systematic reviews". *Keele, UK, Keele University* 33 (2004): 1–26.

Lebedev, Ilya. 2023. *cognitive-complexity - PyPI.* https://pypi.org/project/cognitive-complexity/. Haettu 30.10.2023.

Lincke, Rüdiger, Jonas Lundberg ja Welf Löwe. 2008. "Comparing software metrics tools". Teoksessa *Proceedings of the 2008 international symposium on Software testing and analysis,* 131–142.

McCabe, Thomas J. 1976. "A complexity measure". *IEEE Transactions on software Engineering,* numero 4, 308–320.

Menzies, Tim, Alex Dekhtyar, Justin Distefano ja Jeremy Greenwald. 2007. "Problems with Precision: A Response to"comments on'data mining static code attributes to learn defect predictors'"". *IEEE Transactions on Software Engineering* 33 (9): 637–640.

Muñoz Barón, Marvin, Marvin Wyrich ja Stefan Wagner. 2020. "An empirical validation of cognitive complexity as a measure of source code understandability". Teoksessa *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM),* 1–12.

Nilson, Mayra, Vard Antinyan ja Lucas Gren. 2019. "Do internal software quality tools measure validated metrics?" Teoksessa *Product-Focused Software Process Improvement: 20th International Conference, PROFES 2019, Barcelona, Spain, November 27–29, 2019, Proceedings 20,* 637–648. Springer.

S.A., SonarSource. 2023a. *Metric Definition | Sonar.* https://docs.sonarsource.com/sonarqube/latest/user-guide/metric-definitions/. Haettu 13.10.2023.

S.A., SonarSource. 2023b. *SonarLint | Eclipse Plugins, Bundles and Products - Eclipse Marketplace.* https://marketplace.eclipse.org/content/sonarlint. Haettu 29.9.2023.

———. 2023c. *SonarQube Free Open Source Community Edition | Sonar.* https://www.sonarsource.com source-editions/sonarqube-community-edition/. Haettu 29.9.2023.

Scalabrino, Simone, Gabriele Bavota, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk ja Rocco Oliveto. 2019. "Automatically assessing code understandability". *IEEE Transactions on Software Engineering* 47 (3): 595–613.

Schnappinger, Markus, Arnaud Fietzke ja Alexander Pretschner. 2021. "Human-level ordinal maintainability prediction based on static code metrics". Teoksessa *Evaluation and Assessment in Software Engineering,* 160–169.

SciTools. 2023a. *Pricing : Understand by SciTools.* https://scitools.com/pricing. Haettu 27.10.2023.

———. 2023b. *Supported Languages: SciTools Support.* https://support.scitools.com/support/solutions/a supported-languages. Haettu 27.10.2023.

———. 2023c. *Understand Software Metrics.* https://documentation.scitools.com/pdf/metricsdoc.pdf. Haettu 27.10.2023.

Shen, Vincent Yun, Samuel D. Conte ja Hubert E. Dunsmore. 1983. "Software science revisited: A critical analysis of the theory and its empirical support". *IEEE Transactions on Software Engineering,* numero 2, 155–165.

Software, Brainwy. 2023a. *PyDev - Python IDE for Eclipse | Eclipse Plugins, Bundles and Products - Eclipse Marketplace.* https://marketplace.eclipse.org/content/pydev-python-ide-eclipse. Haettu 29.9.2023.

———. 2023b. *PyDev Code Analysis.* https://www.pydev.org/manual$_{a}dv_{c}ode_{a}nalysis.html$. Haettu 29.9.2023.

QA-systems. 2023. *Static Analysis amp; Verification Tools for Embedded Software | QA Systems.* https://www.qa-systems.com/. Haettu 22.9.2023.

TIMin koodikäytänteet. 2023. Tekninen raportti. Jyväskylän yliopisto.

Tiwari, Ketki. 2020. "Study and Assessment of Reverse Engineering Tool".

Wachowski, Michal. 2023a. *inheritance-explorer - PyPI.* https://pypi.org/project/lcom/. Haettu 5.10.2023.

———. 2023b. *lcom - PyPI.* https://pypi.org/project/lcom/. Haettu 5.10.2023.

Wagner, Stefan. 2013. "Software product quality control".

Walton, Lance. 2023. *Eclipse Metrics Plugin - State Of Flow.* https://eclipse-metrics.sourceforge.net/. Haettu 29.9.2023.

Weihmann, Konrad. 2023. *multimetric - PyPI.* https://pypi.org/project/multimetric/. Haettu 4.10.2023.

Yue, Tao, Lionel C Briand ja Yvan Labiche. 2015. "aToucan: an automated framework to derive UML analysis models from use case models". *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24 (3): 1–52.