Testattavuus ja ohjelmistoarkkitehtuuri

Kristian Wahlroos

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 9. toukokuuta 2016

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Kristian Wahlroos

Työn nimi — Arbetets titel — Title

Testattavuus ja ohjelmistoarkkitehtuuri

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	9. toukokuuta 2016	24

Tiivistelmä — Referat — Abstract

Tässä tutkielmassa tarkastellaan ohjelmiston arkkitehtuurin ja testattavuuden suhdetta. Arkkitehtuurisella tasolla tarkasteltavaksi otetaan ohjelmistojärjestelmän rakenne, jota on mahdollista visualisoida erilaisin keinoin. Arkkitehtuurin kuvauksesta pyritään määrittelemään tekijöitä, joita voidaan arvioida sekä metrisinä että laadullisina ominaisuuksina.

Tavoitteena on selvittää, minkä tekijöiden avulla järjestelmän testattavuutta olisi mahdollista tarkastella visuaalisella tasolla. Tämän lisäksi tarkastelu kiinnittyy erilaisiin keinoihin, joiden avulla metriset ja laadulliset ominaisuudet testattavuuden kannalta tulevat mahdollisimman hyvin ilmi. Tutkimuksen tulos voi täten toimia hyvänä ohjenuorana ohjelmoijille ja alustana koodianalyysi-työkaluille.

Tässä tutkielmassa on käytetty pääosin kahdensuuntaista lumipallomenetelmää. Kirjallisuuskatsauksen avulla on ensin etsitty aiheesta merkittäviä tutkimuksia, joiden lähdeluettelon kautta on etsitty paljon viitattuja tutkimuksia. Tämän jälkeen lähteitä on käytetty käänteisesti eli etsitty niitä teoksia, jotka ovat viitanneet löydettyihin teoksiin.

Tutkielman tuloksena on saatu muodostettua seitsemän päätekijää, jotka löytyvät sekä ohjelmistojärjestelmän abstraktimmalta tasolta että konkreettisemmalta luokkatasolta. Osa näistä muodostetuista ominaisuuksista on metrillisiä ominaisuuksia, osa laadullisia. Kaikkia ominaisuuksia on kuitenkin mahdollista tarkastella ohjelmistojärjestelmän rakenteen visuaalisesta kuvauksesta, jonka abstraktiotaso voi vaihdella riippuen siitä, halutaanko jotain tiettyä asiaa painottaa enemmän. Seitsemän päätekijää on kerätty kootusti taulukkoon tämän tutkielman loppuun.

ACM Computing Classification System (CCS):
Software and its engineering → System description languages
Software and its engineering → Software defect analysis

Avainsanat — Nyckelord — Keywords
ohjelmistoarkkitehtuuri, testattavuus, metriikka, näkymä

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Ohjelmistoarkkitehtuuri on ohjelmistojen ymmärtämisen kannalta tärkeä konsepti, josta jokaisen ohjelmoijan pitäisi olla tietoinen, sillä jokaisella ohjelmistojärjestelmällä on oma arkkitehtuurinsa. Järjestelmän arkkitehtuurin voidaan nähdä muodostuvan niistä hallitsevista ominaisuuksista, jotka ovat konkretisoituneet sen elementteihin ja suhteisiin (ISO/IEC/IEEE, 2011, s. 2). Suunnittelematon arkkitehtuuri eli niin sanottu *Big Ball of Mud* [1] saattaa pahimmassa tapauksessa olla yksi epäonnistuneen projektin tunnusmerkkejä, kun taas hyvin muodostettu arkkitehtuuri auttaa eri sidosryhmien välisessä kommunikaatiossa ja tekee tuotettavasta ohjelmistojärjestelmästä helposti lähestyttävän. Onnistunut arkkitehtuuri myös helpottaa sekä ohjelmiston testausta että validointia ja on yksi suurimmista onnistumistekijöistä projektin jatkon kannalta.

Ohjelmistojen ylläpidettävyys lasketaan yhdeksi ei-toiminnalliseksi laatuvaatimukseksi, ja siihen kuuluu vahvasti ohjelmistojen testattavuus (ISO/IEC, 2010), joka taas vaatimuksena on lähempänä ohjelmoijaa kuin asiakasta. Testausta tapahtuu ohjelmistojärjestelmän elinaikana useita kertoja, joista aikaisimmat jo projektin alkumetreillä, kun halutaan evaluoida järjestelmän toiminnallisuutta. Testattavuus taas määrittelee, kuinka hyvin järjestelmään kohdistuva testaus löytää mahdolliset virheet ja kuinka helppo on erilaisia testitapauksia luoda järjestelmää varten. Arkkitehtuurin ja testattavuuden yhteys on kuitenkin hieman häilyvä, vaikka monelle kehittäjälle hyvät käytänteet ovatkin tuttuja esimerkiksi oliopohjaisten järjestelmien arkkitehtuurista. Suuri osa opituista käytänteistä kuitenkin ottaa kantaa vain lähdekooditasolla tapahtuviin ongelmakohtiin, jolloin arkkitehtuurinen puoli saattaa jäädä kokonaan huomioimatta. Tämä tutkielma pyrkii selkeyttämään ohjelmistojärjestelmän rakenteen ja testattavuuden suhdetta sekä sitä, miten testattavuus tulee ilmi ohjelmistoarkkitehtuurissa, kun ohjelmistoarkkitehtuuri nähdään ohjelmistojärjestelmän rakenteena. Tutkielma edellyttää lukijalta tuntemusta olio-ohjelmoinnin paradigmasta, tietoa siitä miten oliopohjaisia järjestelmiä on mahdollista testata sekä ymmärrystä käsitteistä *yksikkötestaus* ja *integraatiotestaus*.

1.1 Tutkimuskysymys

Tutkielmassa pyritään selvittämään, miten testattavuus näkyy ohjelmistojen arkkitehtuurisella tasolla. Itse kysymys voidaan jakaa vielä tarkentaviin kysymyksiin: mitä testattavuus on ohjelmistojärjestelmissä ja mitkä tekijät siihen vaikuttavat, miten arkkitehtuuria on mahdollista kuvata ja miten ohjelmistojärjestelmän arkkitehtuurista luotu näkymä kertoo jotain järjestelmän testattavuudesta.

[1] Lisätietoa *Big Ball of Mud*-mallista: `http://www.laputan.org/mud/`

1.2 Tutkimusmenetelmä

Tutkimuskysymykseen vastataan kirjallisuuskatsauksen avulla ja tutkimalla, mitkä seikat tuovat testattavuutta ohjelmistojärjestelmään. Näiden perusteella pyritään tunnistamaan kriittisiä arkkitehtuurisia ominaisuuksia, joiden avulla testattavuuden tasoa on mahdollista tarkastella järjestelmässä. Testattavuuden havaitsemiseen etsitään näkymää nykyisistä menetelmistä kuvata arkkitehtuuria. Näitä menetelmiä ovat pääasiassa erilaisten näkymien ja näkökulmien joukko, joiden avulla on mahdollista tuoda ohjelmistojärjestelmästä esiin joitain sen erityispiirteitä.

Itse tutkielma on rajattu ainoastaan oliopohjaiseen arkkitehtuuriin ja sen erilaisiin kuvaamistapoihin. Testaustavoista on mainittu yksikkö- ja integraatiotestaus, koska ne kiteyttävät testausprosessin. Arkkitehtuurin mallintamisesta on jätetty ADL:t (architecture description languages) pois, koska ne eivät tarjoa samanlaista visuaalista näkymää arkkitehtuuriin, kuin pelkästään piirtämiseen keskittyvät. Myöskään erilaisia arkkitehtuurisia tyylejä ei vertailla tutkimuksessa testattavuuden kannalta, vaan tutkimus keskittyy nimenomaan ohjelmistojärjestelmän konkreettisen rakenteen arvioimiseen testattavuuden kannalta.

1.3 Tutkimuksen rakenne

Luvussa 2 käydään läpi ohjelmistoarkkitehtuuria käsitteenä ja luodaan pohja eri kuvaustavoille ohjelmistoarkkitehtuuria varten. Samalla kuvataan, miten ja mistä näkymä voidaan luoda ohjelmistojärjestelmälle. Luku 3 tarkentaa testattavuutta laadullisena tekijänä ja pyrkii määrittelemään, mitä testattavuus on ohjelmistojärjestelmissä. Luku 4 määrittelee arkkitehtuurin ja testattavuuden välistä suhdetta ja sen lopussa on muodostettu kokoava taulukko tutkielman aikana löydetyistä testattavuuteen vaikuttavista tekijöistä. Viimeinen luku 5 esittää kirjoittajan omat pohdinnat ja ehdottaa tutkimuksia tulevaisuuteen.

2 Ohjelmistoarkkitehtuuri

Arkkitehtuuri ohjelmistojärjestelmissä on käsite, joka ei ole kovin yksiselitteinen, eikä siitä ole yksimielistä määritelmää (Solms, 2012, s. 363). Ohjelmistoarkkitehtuurin voidaan kuitenkin sanoa kuvaavan vähintään aina abstraktilla tasolla toteutettavaa ohjelmistojärjestelmää niin, että sitä tarkastelevan tahon on mahdollista keskittyä vain suppeaan alueeseen kerrallaan. Arkkitehtuurin kuvaaminen tapahtuu usein visuaalisella tavalla, koska visuaalisesta mallista saadaan nopeasti ja helposti ulos tarvittava informaatio. Visuaalisen mallin avulla on myös mahdollista nähdä ohjelmistojärjestelmästä sellaista lähdekoodiin piilotettua informaatiota, joka muuten jäisi sen kirjoittaneen kehittäjän omiksi ajatuksiksi.

2.1 Ohjelmistojen arkkitehtuuri käsitteenä

Ohjelmistoarkkitehtuuri voidaan nähdä karkeasti neljästä eri näkökulmasta (Gorton, 2011, s. 2-7): ohjelmiston rakenteen kuvaajana, kuvaajana ohjelmiston komponenttien välisille suhteille, ei-toiminnalliset (non-functional) vaatimukset huomioon ottavana mallina ja yleisenä abstraktiona.

Toisesta näkökulmasta tarkasteltuna arkkitehtuuri voidaan jakaa kolmeen eri luokkaan (Solms, 2012): korkean tason abstraktioon ohjelmistojärjestelmästä, rakenteita sekä ulkoisia ominaisuuksia korostavaan määritelmään ja käsitteistöön sekä rajoitteisiin, joiden puitteissa ohjelmistojärjestelmää kehitetään. Näistä kaksi ensimmäistä määritelmää vastaavat samoja kuin Gortonissa (2011) esitellyt, ja niitä voidaan mallintaa erilaisten näkymien ja UML-kaavioiden avulla seuraavalla tavalla.

- Korkean tason abstraktioita mallinnetaan erilaisten näkymien kautta tai käyttämällä IEEE:n määrittelemiä käytänteitä
- rakenteita korostavassa mallinnuksessa voidaan käyttää UML-kaavioita ja niiden tuomia keinoja abstrahoida järjestelmää.

Näiden kahden määrittelyn ohjelmistoarkkitehtuuri-käsitteen välillä voidaan nähdä yhteisenä piirteenä tarve kuvata ohjelmistojärjestelmän rakenteellisuutta ja kommunikaatiota. Kaikki nämä edellä mainitut on vielä saatava upotettua ohjelmistojärjestelmästä luotavaan arkkitehtuuriseen malliin niin, että ne vastaavat muun muassa seuraaviin kysymyksiin (Rozanski & Woods, 2011, s. 31 - 33): mitkä ovat arkkitehtuurin toiminnalliset elementit; miten nämä elementit kommunikoivat keskenään ja ulkomaailman kanssa; mitä tietoa käsitellään, talletetaan ja esitetään; mitä fyysisiä ja ohjelmallisia elementtejä tarvitaan tukemaan näiden tueksi.

Arkkitehtuurista luotava malli ei kuitenkaan saisi olla monoliittinen malli, koska arkkitehtuuria ei ole mahdollista kuvata vain yhden mallin avulla (Rozanski & Woods, 2011). Tämä johtuu siitä, että monoliittinen malli paisuu liian suureksi, jolloin sitä on vaikea kenenkään osapuolen enää ymmärtää ja siitä on vaikeaa löytää arkkitehtuurin tärkeimmät ominaisuudet. Monoliittinen malli on usein myös puutteellinen, jäljessä eikä vastaa enää nykyistä ohjelmistojärjestelmää.

2.2 Arkkitehtuurin kuvaus

Ratkaisu monoliittiseen malliin on jakaa arkkitehtuurin kuvaus useisiin toisiinsa liittyviin näkymiin (views), jotka esittävät jokainen yhden näkökulman (viewpoint) järjestelmän arkkitehtuuriin keräämällä yhteen sekä toiminnalliset piirteet että laadulliset ominaisuudet (Ran, 1998; Rozanski & Woods, 2011, s. 33-34; Gorton, 2011, s. 8-9).

Useiden erilaisten näkymien avulla on helpompi kommunikoida eri sidosryhmien välillä, ja ne mahdollistavat järjestelmän tarkastelun vain niiltä osin,

jotka tietty sidosryhmä kokee tärkeiksi. Yleisesti voidaan sanoa, että erilaiset näkymät auttavat jakamaan arkkitehtuurin vastuita pienempiin palasiin (Galster, 2011, s. 2). Vastuiden ja näkymien jakaminen taas auttaa jokaista ohjelmistoarkkitehtuuriin liittyvää prosessia tehden ohjelmistoarkkitehtuurin tarkastelusta, suunnittelusta ja toteutuksesta modulaarisempaa.

2.2.1 Näkymät

Arkkitehtuuristen näkymien tehtävänä on kuvata tietylle asialle merkitykselliset näkökulmat arkkitehtuurista. Yksi määritelmistä arkkitehtuuriselle näkymälle on Krutchenin 4+1 -näkymämalli (4+1 View Model) (Kruchten, 1995).

4+1 -mallissa arkkitehtuurin kuvaus muodostetaan neljän näkymän ja yhden tekstuaalisen kuvauksen avulla:

- looginen, joka kuvailee järjestelmän rakennetta ja suhteita
- prosessinen, joka keskittyy suorituksen kuvaamiseen
- fyysinen, joka esittää miten komponentit kuvautuvat fyysiselle laitteistolle
- kehityksellinen, joka kuvailee järjestelmän sisäistä toteutusta
- skenaariollinen, jonka avulla näkymät liitetään yhteen.

Muita näkymämalleja arkkitehtuuriin on useita, joista jokainen kuitenkin tuo esille vähintään jollain tavalla ohjelmistojärjestelmän rakennetta ja yhteyksiä. Tästä esimerkkinä on *Views and Beyond* -malli (Clements et al., 2002). Siinä arkkitehtuurinen malli kuvataan kolmella eri näkymällä:

- moduuli, joka on rakenteellinen näkymä järjestelmään
- komponentit ja konnektorit, jotka kuvailevat järjestelmän toiminnallista puolta ja yhteyksiä
- allokaatio, joka kuvaa miten prosessit kuvautuvat laitteistotasolla.

Näiden kahden esitellyn mallin lisäksi on olemassa joukko eri tarkoituksiin sopivia näkymämalleja, joilla on omat vahvuutensa ja heikkoutensa. Edellä mainituista kahdesta mallista kuitenkin nähdään, että niissä pyritään nostamaan esille järjestelmän rakennetta ja rakenteessa ilmeneviä suhteita, joita muun muassa komponenttien sekä moduulien välillä on.

2.2.2 Näkökulmat

Näkökulmat liittyvät vahvasti arkkitehtuurisiin näkymiin, koska näkymien avulla halutaaan tuoda jokin näkökulma esille (Rozanski & Woods, 2011, s. 36-42). Näkökulman voidaan määritellä olevan kokoelma malleja (patterns), templaatteja sekä konventioita, joiden avulla näkymä pystytään rakentamaan. Näkökulmien joukko taas voidaan jakaa seitsemään eri osaan: kontekstuaalinen, funktionaalinen, informaatiollinen, samanaikaisuudellinen, tuotannollinen, käyttöönotollinen ja operationaalinen. Näistä keskitytään tämän tutkielman myöhemmässä vaiheessa hieman tuotannolliseen sekä funktionaaliseen näkökulmaan. Tuotannollinen näkökulma on tärkeä siksi, että se ottaa huomioon niiden sidosryhmien asiat, jotka ovat mukana järjestelmän kehityksessä, testauksessa, ylläpidossa ja parantamisessa. Funktionaalinen näkökulma taas painottaa muun muassa järjestelmän suorituksenaikaisia komponentteja ja niiden rajapintoja. Näiden avulla halutaan tuoda esille näkökulma testattavuuden analysoimiseen, jota esimerkiksi kehittäjät voisivat hoydyntää.

IEEE:n virallinen määritelmä näkokulmalle on (ISO/IEC/IEEE, 2011, s. 6) *A concern can be framed by more than one viewpoint. A view is governed by its viewpoint: the viewpoint establishes the conventions for constructing, interpreting and analyzing the view to address concerns framed by that viewpoint. Viewpoint conventions can include languages, notations, model kinds, design rules, and/or modelling methods, analysis techniques and other operations on views.* Se on määritelmä, jota tullaan käyttämään myös tässä tutkielmassa.

2.3 Näkymän luominen

Ohjelmistoarkkitehtuuri toimii siltana ohjelmakoodin ja ohjelmalle asetettujen vaatimusten välillä (Garlan, 2000, s. 94), joten on luonnollista, että näkymä luodaan näiden pohjalta. Sitä luodessa on otettava huomioon, kenelle näkymä on tarkoitettu, minkälaisia asioita halutaan näkymän painottavan ja millä abstraktiotasolla näkymän tulisi olla.

Luotavan näkymän perimmäinen tarkoitus voi esimerkiksi olla, että mahdollisimman paljon halutaan nähdä arkkitehtuurisia ominaisuuksia katsomatta lähdekoodia. Syy tähän voi olla esimerkiksi se, ettei lähdekoodi ole saatavilla tai että sen läpikäyminen kokonaan ajatuksen kanssa on kehittäjälle liian monimutkaista ja aikaa vievää. Yleisesti koetaan myös, että toisen kirjoittamaa ohjelmakoodia on vaikea ymmärtää[2]. Tällaiseen näkymän tarpeeseen sopivat esimerkiksi UML:n avulla luodut kaaviot järjestelmän rakenteeesta, joiden avulla erilaisten suhteiden ja rakenteellisten ominaisuuksien tarkastelu helpottuu huomattavasti.

[2]Syitä miksi toisen ohjelmakoodia on vaikea lukea: `http://www.preposterousuniverse.com/blog/2013/04/24/why-is-code-hard-to-understand/`

2.4 UML

UML eli *Unified Modeling Language* on yksi visuaalinen formalisoitu mallinnuskieli näkymien luomiseen (Object Management Group, 2015). Sen avulla ohjelmistoa voidaan mallintaa kolmelta eri näkökulmalta: kehitystä täydentävät mallit, käyttäytymisen kuvaamiseen tarkoitetut mallit ja rakennetta kuvaavat mallit. Käytännössä sen avulla pystytään kuvaamaan esimerkiksi jokainen sekä 4+1 -mallin että *Views and Beyond* -mallin näkymä käyttäen hyväksi UML:n 13 erilaista kaaviota. Arkkitehtuurin kuvauksen kannalta relevanteimmat kaaviot ovat luokkakaavio ja komponenttikaavio, koska niiden avulla saadaan kuvattua erittäin tarkasti ja monipuolisesti järjestelmän rakenteellisia ominaisuuksia ja rajoitteita. Luokkakaaviossa huomio keskittyy konkreettisten luokkien ja niiden välisten suhteiden kuvaamiseen, kun taas komponenttikaavio kuvaa korkeammalla tasolla järjestelmää kuvaavia komponentteja, joita yhdistävät erilaiset rajapinnat.

UML sopii ominaisuuksiensa takia erittäin hyvin oliojärjestelmien kuvaamiseen tarjoamalla tuen muun muassa periytymisen ja rajapintojen kuvaamiselle. Tämän takia se on otettu tässä tutkielmassa erityisasemaan järjestelmän ja sen rakenteen kuvaamisessa.

3 Testattavuus laadullisena tekijänä

Testattavuus luetaan kuuluvaksi ylläpidettävyyden alle (ISO/IEC, 2010) ja on täten yksi ohjelmistojärjestelmän laadullisista tekijöistä. Laatuvaatimuksena se ei kuitenkaan ole aina kovin näkyvä esimerkiksi asiakkaalle, mutta ohjelmistokehittäjälle se tulee esille useassakin kontekstissa. Vaikeasti testattava järjestelmä on kehittäjälle vaikea ylläpitää ja ylläpidettävyys on tärkeä ominaisuus myös asiakkaalle, koska se tuo järjestelmään vakautta.

Ohjelmistojärjestelmään kohdistuva testaus voidaan jakaa moneen eri luokkaan, joista kaksi esimerkkiä ovat yksikkö -ja integraatiotestaus. Yksikkötestauksessa varmistetaan järjestelmän komponenttien kuten luokkien toiminnallisuutta itsenäisesti, kun taas integraatiotestauksessa testaus kohdistuu järjestelmän rajapintoihin ja komponenttien yhteistoiminnallisuuteen. Testauksen voidaan siis nähdä löytävän itse viat, mutta testattavuus taas paikat, joissa viat voivat sijaita (Voas & Miller, 1995, s. 19). Testattavuus tukee täten testausta ja testausprosessia, ja sen avulla on mahdollista määritellä myös todennäköisyyttä sille, että järjestelmässä piilevät viat löydetään (Voas & Miller, 1992).

3.1 Testattavuuden arviointi

Testattavuuden arvioimisessa voidaan käyttää hyödyksi kolmea eri parametria (Freedman, 1991): havaittavuus (observability), hallittavuus (controllability) ja vaativuus, joka tarvitaan suorittamaan tietty testi. Havaittavuus

määrittelee, kuinka mahdollista testien tuloksia on tarkastella, ja hallittavuus sitä, kuinka paljon testauksen kohteena olevan komponentin tilaan voidaan vaikuttaa.

Havaittavuus voidaan myös nähdä arvona, joka kertoo, kuinka helppo on määrittää, vaikuttavatko tietyt syötteet testituloksiin ja hallittavuus vaativuutena tuottaa jokin tietty tulos syötteestä (Freedman, 1991, s. 554). Hallittavuuden parametrisoimiseen on mahdollista käyttää *domain-to-range* -arvoa eli DRR-arvoa. Sen avulla voidaan havaita tiedon katoamista komponenttien sisällä (Voas & Miller, 1995), ja se pystytään muodostamaan laskemalla tulosteen (output) suhteellinen osuus syötteen (input) koosta.

3.2 Testattavuuden parantaminen

Hyvä testattavuus auttaa järjestelmän ylläpidossa, validoinnissa ja parantaa laatua (Voas & Miller, 1995, s. 20), joten se on tärkeä ominaisuus jokaisessa ohjelmistojärjestelmässä. Tarkastelu hyvään testattavuuteen voidaan kohdistaa niihin tekijöihin, jotka yleisesti luovat hyvää testattavuutta, jotta saataisiin muodostettua kuva ilman takertumista yksityiskohtiin.

Koska ohjelmistojärjestelmä koostuu sen rakenteen muodostavista komponenteista, tutkiminen voidaan keskittää ensin niihin. Testattavan komponentin ominaisuuksiksi voidaan määritellä seuraavia piirteitä (Freedman, 1991): komponentista luotavat testitapaukset pysyvät pieninä ja ovat helposti luotavissa; komponentin testitapaukset eivät ala toistamaan itseään; komponentin testauksen aikana havaitut ongelmat on helppo jäljittää tiettyihin komponentteihin. Näitä tukemaan voidaan jo ohjelmistojärjestelmän alkuvaiheessa kiinnittää huomiota tekijöihin, jotka saadaan DRR-arvoja analysoimalla (Voas & Miller, 1992)

- ymmärtää toiminnan eriyttämisen mahdollisuus
- tunnistaa moduulit, joiden DRR-arvo on korkea, ja tehdä ne mahdollisimman pieniksi ja yksinkertaisiksi
- tunnistaa rajapinta korkean DRR-arvon moduuleille siten, että rajapinta paljastaa tarpeeksi tietoa komponenttien sisäisestä tilasta
- eriyttää korkean DRR-arvon komponentit toisistaan (Voas & Miller, 1995, s. 23).

Hyvää testattavuutta on mahdollista saavuttaa pelkästään analysoimalla komponentteja yksittäisinä tekijöinä. Huomio voidaan kiinnittää sellaisiin seikkoihin kuten kuinka paljon komponentit tukevat itse testausprosessia ja kuinka paljon komponentit ovat riippuvaisia toisistaan. Suuria DRR-arvon komponentteja pitäisi yrittää välttää ja eriyttää toiminnallisuus niin, että mahdollisia ongelmakohtia pystyttäisiin hallitsemaan ja havaitsemaan.

3.3 Testausta heikentävät ominaisuudet

Tietyt ominaisuudet vaikuttavat yleisesti negatiivisesti testattavuuteen ohjelmistojärjestelmissä. Tiedon katoaminen on yksi näistä (Voas & Miller, 1995), ja se voidaan jakaa implisiittiseen ja eksplisiittiseen katoamiseen. Implisiittisessä katoamisessa useat eri parametrit samaan kohteeseen tuottavat saman testituloksen, ja eksplisiittisessä tiedon katoamisessa tietoa piilotetaan ulkoiselta näkymältä, eli tieto jää tarkastelemattomaksi ominaisuudeksi komponentin sisälle.

Tiedon piilottaminen on hyvin yleinen tapa olio-ohjelmoinnissa, mutta liiallinen tiedon piilotus tuottaa vaikeuksia testauksessa, koska testauksen hallittavuus heikkenee. Ongelmat syötteiden ja tulosten tarkastelussa tiedon katoamisen vuoksi johtavat helposti tarpeettomiin testeihin, joita on vaikea ymmärtää, ja vaikeasti testattava komponentti sisältää ulos- ja sisääntuloparametrien epäjohdonmukaisuuksia (Voas & Miller, 1995). Toisin sanoen vaikeasti testattava komponentti tuottaa eri tuloksia eri ajoilla tai ei ikinä tuota haluttua tulosta, koska sen hallittavuus ja havaittavuus ovat ongelmallisia.

3.4 Testattavuuden mittaaminen

Erilaisia testattavuuteen luokkatasolla vaikuttavia mittareita on tutkittu paljon, koska niiden käyttäminen tuo konkretiaa analysoimiseen ja niiden pohjalta on mahdollista tehdä esimerkiksi staattista koodianalyysia suorittavia erillisiä ohjelmia[3].

Yksikkötestauksen näkökulmasta Java-pohjaisille ohjelmistojärjestelmille on määritetty metriikoita tutkimuksessa, jossa mittareina käytettiin testitapausten kasvua ja pituutta (Bruntink & van Deursen, 2004). Hieman samanlainen jaottelu tehtiin eräässä toisessa myöhemmässä tutkimuksessa. Siinä määriteltiin yleisesti tiettyjä testaukseen vaikuttavia mittareita, joiden suuri arvo korreloi vahvasti huonon testattavuuden kanssa (Dubey & Rana, 2011). Nämä määritetyt tekijät ovat negatiivisia asioita testauksen kannalta, koska esimerkiksi suuri metodien määrä on merkki luokan liiallisesta kompleksisuudesta ja se johtaa testauksen monimutkaistumiseen. Suuri kytkennän määrä vaikeuttaa luokan itsenäistä testaamista, koska luokka käyttää hyvin paljon hyväkseen muiden luokkien toiminnallisuutta. Suuri periytymisaste tuo taas luokalle mahdollisesti paljon perittyjä metodeja, jotka on toteutettu jossain muualla. Tämä voi johtaa helposti siihen, että luokka on riippuvainen erittäin paljon muiden luokkien toteutuksista ja luokan testaaminen vaatii käytännössä koko luokkahierarkian läpikäymisen.

Neljää korkeamman tason metriikkaa on käytetty laskemaan testattavuutta ohjelmistojärjestelmissä, jotka korreloivat edellisten tutkimusten met-

[3]Esimerkiksi JArchitect-ohjelman käyttämät metriikat: `http://www.jarchitect.com/Metrics#MetricsOnApplication`

riikoiden kanssa. Niitä voidaan pitää keskeisinä olio-ohjelmoinnin suunnitteluperiaatteina (Khan & Mustafa, 2009), ja edellämainituissa tutkimuksissa käytetyistä mittareista voidaankin muodostaa seuraava taulukko, jossa jokaisen mittarin pääidea on kiteytetty ja suhteutettu neljään korkean tason yleismetriikkaan.

Taulukko 3.1: Mittarit pelkistettyinä ja niiden suhteet Khanin ja Mustafan esittämään neljään eri korkean tason metriikkaan. Ehdotettu-sarake kertoo, mistä lähteestä mittarin ideat ovat.

Mittari	**Korreloi**	**Ehdotettu**
Ulkoiset viittaukset	Kytkentä, Koheesio	Bruntink & van Deursen (2004)
Koko (rivit + metodit)	Enkapsulaatio	Bruntink & van Deursen (2004); Dubey & Rana (2011)
Luokkahierarkien aktivoituminen	Enkapsulaatio	Dubey & Rana (2011)
Kytkennän määrä	Kytkentä	Dubey & Rana (2011)
Periytymisaste	Uudelleenkäyttö	Dubey & Rana (2011)

Taulukkoon 3.1 on kerätty mittareiden pääideat kaikista kolmesta lähteestä, ja siitä voidaan nähdä luokkien näkökulmasta muutama iso tekijä testattavuudessa: koko, itsenäinen toimiminen, riippuvuudet muista luokista ja periytyminen. Mittareista osa on kuitenkin ominaisuuksiltaan sellaisia, että niitä voidaan mitata suurimmaksi osaksi vain lähdekoodin avulla ja ne ottavat kantaa testattavuuteen vain yhden luokan näkökulmasta. Tämä vaikeuttaa kokonaiskuvan hahmottamista, koska lähdekoodin kautta tulee liikaa irrelevanttia tietoa vaikeuttaen tiedon analysointia. Suuremman kokonaisuuden hahmottaminen vaatii siis muita keinoja avukseen.

4 Testattavuus arkkitehtuurisella tasolla

Arkkitehtuurisella tasolla testattavuutta voidaan tutkia esimerkiksi komponenttitasolla, jota tukevat hyvin UML-komponenttikaaviot. Tämä sopii rakenteen arvioimiseen hyvin, koska testattavuus tulee ilmi korkealla tasolla muun muassa integraatiotestauksen aikana, kun sekä komponenttien rajapintojen toimivuus että komponenttien toimivuus pitää validoida (Eickelmann & Richardson, 1996, s. 65). Toinen tapa tutkia testattavuutta arkkitehtuurisella tasolla on UML-luokkakaavioiden avulla, joiden apuna voidaan käyttää hyväksi esimerkiksi niitä aliluvun 3.4 metriikoita, jotka eivät tarvitse lähdekoodia tuekseen.

Yksittäisten komponenttien, eli luokista koostuvien kokonaisuuksien testattavuutta, on analysoitu erilaisten mallien avulla ja näistä eräs on komponenttien pentagrammimalli (Gao & Shih, 2005). Sen avulla jokaista kompo-

nenttia tarkastellaan viideltä eri näkökulmalta: havaittavuus, hallittavuus, jäljitettävyys (traceability), tuki testivalmiudelle (test support capability) ja ymmärrettävyys (understandability). Näistä kaksi ensimmäistä ovat ideoiltaan samat kuin aliluvussa 3.1 ja loput vastaavat muun muassa kysymyksiin: kuinka helppoa on tarkastella komponentin tilaa ulkopuolisena; kuinka hyvin komponentti tukee erilaisia testausstrategioita; kuinka hyvin komponentin tarkoitus ymmärretään, että testejä pystytään luomaan.

On siis hyvä tarkastella testattavuutta arkkitehtuurisella tasolla kahdelta eri näkökulmalta: metriikoista koostuva analyysi ja yksittäisten komponenttien laadullinen tarkastelu. Analyysi ilman metriikoita keskitetään enemmän kokonaisuuksien hahmottamiseen ja ongelmakohtien löytämiseen laadullisten ominaisuuksien pohjalta, kun taas metriikoiden kanssa analyysi on staattista ja painottaa enemmän määrällistä tutkimusta.

4.1 Hyvä testattavuus arkkitehtuurissa

Hyvää testattavuutta arkkitehtuureissa on vaikea määrittää, koska eri järjestelmillä on tiettyyn tarkoitukseen sopiva arkkitehtuuri. Tietyt arkkitehtuuriset tyylit tukevat kuitenkin tiettyjä testausstrategioita (Eickelmann & Richardson, 1996), joten on selvää että oikein valittu arkkitehtuurinen tyyli vaikuttaa positiivisesti testattavuuteen. Paljon mielekkäämpää on kuitenkin tarkastella ominaisuuksia testattavalle arkkitehtuurille, jotta voitaisiin luetella erinäisiä testattavuuteen vaikuttavia tekijöitä.

Oliopohjaisissa ohjelmistojärjestelmissä koetaan vähintään vastuiden jakamisen olevan lähtökohta hyvälle testattavuudelle, koska se parantaa koko järjestelmän havaittavuutta ja hallittavuutta (Binder, 1994). Suoraan ohjelmistojärjestelmän rakenteelliselta tasolta voidaan huomio keskittää esimerkiksi kapseloinnin, polymorfismin, perinnän ja kompleksisuuden tarkasteluun.

Järjestelmää suunniteltaessa testauksen varalle, on siihen olemassa erilaisia laadullisia ja metrisiä ominaisuuksia, joihin huomio pitäisi kiinnittää koko ohjelmistoprosessin ajan (Joshi & Sardana, 2014). Näiden metriikoiden avulla löydettyjen ongelmakohtien eliminoimiseen voidaan käyttää hyväksi järjestelmän modulaarista rakennetta, joka pitäisi myös olla vallitseva ominaisuus järjestelmässä koko sen elinkaaren ajan. Toinen apukeino on UML ja sen muokkaaminen stereotyyppien avulla. UML-kaavioiden muokkaus voi tapahtua esimerkiksi luomalla uusia malleja nykyisten pohjalta tai käyttämällä UML:n geneerisiä laajennoskohtia, joiden avulla näkymää saadaan muokattua haluttuun suuntaan.

Taulukko 4.1: Hyvään testattavuuteen rakenteellisella tasolla vaikuttavat ominaisuudet jaoteltuna kahteen eri luokkaan: laadulliset ja metriset.

Analyysimentelmä	**Ominaisuus**	**Lähde**
Laadullinen	Testausominaisuus, jäljitettävyys, vaatimusten tarkkuus, antimallit[a], havaittavuus, hallittavuus	Binder (1994); Joshi & Sardana (2014)
Metrinen	Sykliset riippuvuudet, kytkennän taso, koheesion taso	Joshi & Sardana (2014)

[a]Toistuva ratkaisu johonkin ongelmaan, mikä vaikuttaa kuitenkin negatiivisesti laatuun.

Taulukossa 4.1 on koottu yhteen ominaisuudet, jotka ottavat kantaa laadullisiin ja mitattaviin ominaisuuksiin. Niistä nähdään, että on tärkeää tarkastella sekä komponenttien ulkoisia että sisäisiä ominaisuuksia, jotta voitaisiin muodostaa hyvä kuva testattavuuden tasosta. Suoraan ohjelmistojärjestelmän rakenteen näkökulmasta esiin nousevat riippuvuudet, kytkentä ja koheesio.

4.2 Arkkitehtuurin testattavuuden mittaaminen

Ylläpidettävyys on vaikuttava piirre testattavuuteen, joten on luonnollista tarkastella myös sen kautta järjestelmän testattavuutta. Ylläpidettävyyden näkökulmasta on tunnistettu erilaisia arkkitehtuurisen tason mittareita arkkitehtuurisille elementeille ja niiden avulla voidaan mitatata järjestelmän ominaisuuksia, kuten kytkennän ja koheesion tasoa (Bengtsson, 1998).

Taulukko 4.2: Mittarit, joiden avulla ylläpidettävyyttä voidaan mitata. Tarkkuuden taso on skaalattu arkkitehtuurin näkökulmasta matalasta korkeaan. Matala tarkoittaa, että kuvaus on lähellä ohjelmakoodia ja korkea, että mittaria pystytään käyttämään myös abstraktimmalla tasolla.

Mittari	**Tarkkuuden taso**
Metodien määrä elementin rajapinnassa	Matala
Elementin metodien ja parametrien määrä	Matala
Saatavilla olevat metodit muualta	Matala
Lähtevien viestien määrä	Matala/Korkea
Paikkamerkkien (placeholder) määrä	Korkea
Elementtityyppien määrä, joita tarkasteltava elementti toteuttaa	Korkea
Elementtien määrä, jotka toteuttavat tarkasteltavan elementin tyypin	Korkea

Taulukkoon 4.2 kootuista mittareista nähdään, että mittarit jakautuvat karkeasti kahteen eri tarkkuuden tasoon: matalaan ja korkeaan. Syy siihen on,

että arkkitehtuurisen kuvauksen ollessa abstraktio ohjelmakoodista on luonnollista, että eri tarkkuuden tasoilla on mahdollista liikkua. Tunnistetuiden mittareiden ominaisuuksista taas nähdään samankaltaisuutta aliluvun 3.4 määriteltyjen mittareiden kanssa, mutta tässä tunnistetut mittarit tarkastelevat korkeampaa tasoa. Kytkentä ja koheesio, joita kummatkin mittarisarjat osittain mittaavat, määriteltiin myös tavoiksi mitata järjestelmän testattavuutta samaisessa aliluvussa. Kytkentää voidaan tutkia analysoimalla ohjelmistojärjestelmän arkkitehtuuristen elementtien välisiä suhteita ja koheesiota tutkimalla kuinka itsenäisiä ne ovat ovat.

4.2.1 UML-kaaviot testattavuuden mittaamisessa

UML-luokkakaavio toimii hyvänä pohjana, kun halutaan mitata luokkien välisiä suhteita ja riippuvuuksia (Baudry, Traon & Sunye, 2002) ja sen avulla voidaan määritellä testattavuuden antimalli (testability anti-pattern), joka on luokkakaavioiden avulla esille tuleva ilmiö (Baudry et al., 2003). Antimallissa kaksi suurta tekijää vaikuttavat testattavuuteen negatiivisesti: luokkien välinen suuri interaktio ja itsekäyttö, jossa luokka on riippuvainen rekursiivisesti itsestään. Syitä näiden kahden esiintymiselle voidaan lukea olevan: luokkien välinen suuri kytkentätaso, riippuvuuksien syklimäisyys ja luokkien välinen suuri kommunikaatiotaso. Yhdessä nämä tekijät vaikeuttavat paljon vastuiden jakamista ja kasvattavat tarvittavien testitapausten määrää (Baudry et al., 2003). Antimallin osatekijöiden löytyminen arkkitehtuurista on siis vihje siitä, että arkkitehtuuri saattaa sisältää testausta vaikeuttavaa rakennetta.

Toinen mahdollinen lähestymistapa on tarkastella järjestelmää UML:n avulla luotavalla erityisellä riippuvuusgraafilla (Baudry, Traon & Sunye, 2002), jonka avulla nähdään komponenttien välistä riippuvuuksien hajontaa. On myös mahdollista tarkastella testauksen kannalta kriittisten riippuvuuksien poistoa analysoimalla luokkakaaviosta saatavia ominaisuuksia (Jungmayr, 2002). Tällöin riippuvuudeksi lasketaan mikä tahansa komponenttien välinen interaktio, jossa komponentti joutuu käyttämään toisen komponentin tarjoamia palveluita, jotka ovat usein metodeja, suorittaakseen toiminnallisuuttaan.

Riippuvuuksiin liittyviä tekijöitä voidaan havainnollistaa seuraavan taulukon avulla, jossa jokaiselle testautta vaikeuttavalle ominaisuudelle on annettu myös tapa, jolla ominaisuutta on mahdollista tarkastella.

Taulukko 4.3: Tekijöitä ongelmallisten riippuvuuksien syntymiselle testattavuuden kannalta.

Ominaisuus	**Tapa havaita**	**Lähde**
Kompleksiset perintäsuhteet	Graafimalli	Baudry, Traon & Sunye (2002)
Abstraktien luokkien liikakäyttö rajapintojen sijasta	Graafimalli ja luokkakaavio	Baudry, Traon & Sunye (2002)
Metriikat • Riippuvuuksien lukumäärä per komponentti • Komponenttien määrä riippuvuussykleissä • Feedback-riippuvuuksien määrä[a]	Graafimalli ja luokkakaavio	Jungmayr (2002)

[a]Feedback-riippuvuus on syklinen riippuvuussuhde komponenttien välillä, josta poistamalla jokin riippuvuus saadaan sykli purettua täysin.

Taulukossa 4.3 olevista tekijöistä voidaan nähdä, että riippuvuuksien tarkastelu testattavuuden kannalta on mahdollista UML-kaavion ja siitä luotavan graafimallin avulla. Varsinkin sykliset riippuvuudet tulevat ilmi testattavuutta heikentävänä tekijänä, joiden tarkastelu kooditasolla on hyvin vaikeaa, koska kooditasolla keskitytään vain yhden luokan näkökulmaan ja kokonaisuus voi tällöin jäädä hahmottamatta.

UML on toiminut työkaluna löytämään yhteydellisiä ominaisuuksia arkkitehtuurissa, joita tarkastellessa voidaan sanoa tarkasteltavan kytkennän tasoa, kompleksisuutta ja riippuvuuksia. Komponenttien sekä yhteyksien tarkastelu oli myös luvussa 2.1 esitelty eräs näkemys ohjelmistoarkkitehtuuriin ja on järjestelmän todellisen testattavuuden määrittelemisen kannalta parempi, jos testattavuutta voidaan tarkastella myös korkealla tasolla. Liian tarkka kuvaus järjestelmästä vaikeuttaa todellisten ongelmien näkemistä, koska se vie aikaa ymmärtää ja keskittyy liian pikkutarkkoihin asioihin, jotka eivät kaikki välttämättä ole relevantteja testattavuuden analysoinnin kannalta. Ylätason kuvauksella on etuna myös se, että sen avulla pystytään löytämään helposti usein esiintyviä ongelmallisia tilanteita, joiden tunnistaminen auttaa järjestelmän testattavuuden hallinnassa.

4.2.2 UML uusien näkymien tukena

Standardi UML-kaavio ei kuitenkaan itsessään riitä näyttämään kaikkia esiin tulleita testattavuuden kannalta tärkeitä ominaisuuksia, mutta sen tueksi on luotu useita menetelmiä, jotka hyödyntävät esimerkiksi jo valmista UML-luokkakaaviota. Eräs näistä on graafimalli eli *class dependency graph model* (Baudry, Traon & Sunye, 2002), joka myös aliluvussa 4.2.1 mainittiin. Sen

avulla on mahdollista nähdä muun muassa luokkien välistä suurta interaktiota, itsekäyttöä, interaktioiden kompleksisuutta, pitkiä luokkahierarkisia haaraumia ja kompleksisia periytymisiä rajapintatasoilla. Kaikista näistä on mahdollista luoda erilaisia metriikoita ja saada laskettua kytkennän, perinnän ja yleisen kompleksisuuden tasoa testattavuuden näkökulmalta.

Tämä tuottaa kuitenkin ongelmia, koska jotkut tulokset saattavat olla vääriä niiden sisältäessä virheellistä esiintymää (false-positive results). Tähän ongelmaan on ehdotettu UML-kaavion jatkamista stereotyyppien avulla. Uusia kytköksiä riippuvuuksille ehdotetaan neljä kappaletta (Baudry et al., 2003, s. 4)

- *create*, jos luokka A luo luokan B
- *use*, jos luokka A voi kutsua kaikkia B:n metodeja, muttei luo sitä
- *use_consult*, jos A:n kutsumat metodit eivät muokkaa B:n sisäistä tilaa
- *use_def*, jos yksikin A:n kutsuma metodi muokkaa B:n sisäistä tilaa

Kyseisten stereotyyppien avulla saadaan luokkakaavioissa paremmin esille riippuvuudet ja niiden tyypit virheellisten tuloksien välttämiseksi. On esimerkiksi usein melko normaalia oliojärjestelmissä, että yksi luokka saattaa luoda useita riippuvuuksia muihin luokkiin, mutta tämä on tehty tarkoituksella eikä vahingossa kehittäjien toimesta. Näin on esimerkiksi tehdas-mallin[4] (factory pattern) käytössä, jossa erikseen määritelty tehdas-olio on riippuvainen useiden eri luokkien synnystä. Se vähentää globaalisti riippuvuuksien määrää, mutta itse tehdas-oliot tuottaisivat metriikoiden mukaan paljon virheellisiä riippuvuusongelmia, jos jokainen kaaviossa esiintyvä riippuvuus olisi samanarvoinen.

4.3 Arkkitehtuuriset hajut ja testattavuus

Arkkitehtuurisiin hajuihin on alettu kinnittämään yhä enemmän huomiota niiden helposti lähestyttävien konkreettisten ominaisuuksiensa vuoksi (de Andrade, Almeida & Crnkovic, 2014). Ne ovat sukua tutummalle käsitteelle koodihaju, mutta tulevat esille arkkitehtuurisella tasolla, kun koodihajut tulevat esille luokkatasolla ja vaativat melkein aina lähdekoodin tuekseen. Arkkitehtuuriset hajut ovat pääosin suunnittelupäätöksiä, jotka negatiivisesti vaikuttavat järjestelmän elinkaareen ja muun muassa testattavuuten sekä ylläpidettävyyteen. Niitä ei voi pitää välttämättä virheinä, mutta ne vaikuttavat vähintään aina laatuun huonontavalla tavalla (de Andrade, Almeida & Crnkovic, 2014). Syitä arkkitehtuurisille hajuille voidaan nähdä olevan muun muassa: suunnittelumalli väärässä kontekstissa; suunnitteluabstrahoinnin (design abstraction) käyttö niin, että sillä on ei-toivottuja vaikutuksia; suunnitteluabstrahoinnin käyttö väärällä tarkkuuden tasolla.

[4]Lisätietoa tehdas-mallista: `http://www.oodesign.com/factory-pattern.html`

Arkkitehtuuristen hajujen tutkimiseen voidaan käyttää analyyttistä lähestymistapaa, koska arkkitehtuuristen hajujen lähtökohtana toimii usein joku virheellinen arkkitehtuurinen päätös. On kuitenkin helpompi tutkia visuaalista informaatiota, jossa voidaan hyödyntää seuraavia metriikoita (Bertran, 2011).

- ei-haluttujen asiakas-komponenttien (client component) määrä
- komponentin vastuiden määrä
- komponentin konnektoreiden määrä
- komponentin erilaisten konnektoreiden määrä
- komponenttien välinen riippuvuuksien hajauma.

Metriikoista nähdään, että ne mittaavat paljon koheesiota, kytkentää, riippuvuuksia ja vastuita. Näiden metriikoiden avulla sekä näkymällä moduulitasoon, olisi esimerkiksi hyvin mahdollistaa löytää yksi virhealttiillisin arkkitehtuurista löytyvä ongelmakohta *Cross-Module Cycle* (Mo et al., 2015, s. 57). *Cross-Module Cycle* on olemassa arkkitehtuurissa silloin, kun siinä on syklinen riippuvuussuhde moduulitasolla. Se vaikeuttaa testattavuutta siten, että testauksen aikana yhden moduulin testaaminen vaatii käytännössä myös kaikkien muiden moduulien testauksen, jolloin testitapausten eriyttäminen vaikeutuu huomattavasti. Sykliset riippuvuudet todettiin olevan myös aliluvun 4.1 yksi testattavuuteen vaikuttava tekijä. Muita arkkitehtuurisia hajuja, jotka suoraan vaikuttavat testattavuuteen, voidaan tunnistaa olevan ainakin kaksi lisää (Garcia et al., 2009): *Connector Envy* ja *Scattered Parasitic Functionality*.

Connector Envy on tilanne, jossa komponentti toteuttaa konnektorille kuuluvaa toiminnallisuutta eli toteuttaa esimerkiksi joko kommunikointia, koordinointia tai tiedon konversiota. Nämä toiminnallisuudet tulisi jättää konnektorin toteutettavaksi, koska komponentin toiminnallisuus ja interaktiot eivät ole enää erillään testattavia ominaisuuksia. Tilanteena se johtaa siihen, että ei pystytä enää varmasti sanomaan, että mikä kohta komponentissa oli virheellinen, jos se testauksen aikana hajoaa. *Scattered Parasitic Functionality* taas syntyy, jos useammalla komponentilla on sama korkean tason vastuu ja joidenkin näiden komponenttien täytyy toteuttaa myöskin joku oma ortogonaalinen vastuu. Tämä estää komponenttien järkevän yhdistämisen ilman, että uuteen komponettiin tulee parasiitteja eli ylimääräisiä vastuita. Testattavuus kärsii tästä, koska esimerkiksi integraatiotestauksen aikana tapahtunut virhe vaikuttaa laajalla alueella ja vaikeuttaa vian alkuperän määrittämistä.

Testaukseen vaikuttavista arkkitehtuurisista hajusta voidaan huomata, että ne pyrkivät löytämään sykliset riippuvuudet, huonon koheesion ja riippuvuuksien jakautumisen turhan laajalle. Samat asiat jotka vaikuttavat

luokkatasolla testattavuuteen vaikuttavat arkkitehtuurisella tasolla, mutta hieman eri näkökulmasta. Arkkitehtuurisella tasolla tarkasteltavana voi olla usean eri luokan kokonaisuudet eli komponentit tai arkkitehtuuriset elementit, kun luokkatasolla tarkastelun alla on yksittäiset luokat ja jopa yksittäiset metodit.

4.3.1 Koodihajujen ja arkkitehtuuristen hajujen suhde

Arkkitehtuurin ja ohjelmakoodin suhde on luonnollisesti hyvin vahva ja monissa metriikoissakin on nähty samankaltaisuuksia. Tätä suhdetta selittämään on tuotu käsite hybridihajut, jotka ovat sekoitus koodihajuja ja arkkitehtuurisia hajuja, tuoden uuden näkökulman arkkitehtuuristen hajujen löytämiseksi myös testattavuuden kannalta. Koodipoikkeama (code anomaly) luetaan hybridihajuksi, jos sen vaikuttamat luokat ovat vastuussa arkkitehtuurisista elementeistä ja näissä arkkitehtuurisissa elementeissä havaitaan ongelmia (Vale et al., 2014). Hybridihajujen tunnistaminen on siis eräs keino helpottaa arkkitehtuuristen hajujen löytämistä. Löytämällä koodihajujen pohjalta arkkitehtuuriin vaikuttavia ongelmallisia luokkia, voidaan vastaavasti arkkitehtuuriselta tasolla tunnistaa muodostuneita arkkitehtuurisia hajuja. Arkkitehtuuristen hajujen löytäminen saattaa taas antaa tärkeää tietoa pinnan alla piilevistä koodihajuista, joten kumpikin suunta täydentää hyvin toisiaan.

On tutkittu, että tietyt kooditason ongelmat ovat indikaattoreita arkkitehtuurisista hajuista aspektiorientoituneissa järjestelmissä (Macia et al., 2011). Seuraavaan taulukkoon on koottu nämä ongelmat yhteen ja suhteutettu ne testattavuuten vaikuttaviin arkkitehtuurisiin hajuihin.

Taulukko 4.4: Arkkitehtuurinen haju ja syy sen esiintymiselle kooditasolla.

Arkkitehtuurinen haju	**Syy kooditasolla**
SPF	Liikaa vastuita
SPF, Connector Envy	Liikaa suoria vastuita
SPF	Liikaa riippuvuuksia
SPF	Samankaltaista toiminnallisuutta

Vaikka taulukon 4.4 koodihajut eivät suoraan koske oliopohjaisia järjestelmiä, niistä voidaan silti nähdä, että ongelmat tulevat liiallisista vastuista, liiallisista riippuvuuksista ja samankaltaisesta toiminnallisuudesta. Nämä ongelmat ovat nähtävissä tavallisissakin oliojärjestelmissä, joista on mahdollista määrittää muutamia arkkitehtuuriherkkiä koodipoikkeamia (Macia et al., 2013).

Seuraavassa taulukossa on kerätty oliojärjestelmissä esiintyvät ongelmat ja niiden suorat yhteydet testattavuuteen vaikuttaviin arkkitehtuurisiin hajuihin (Macia et al., 2013).

Taulukko 4.5: Olioperäisten koodihajujen ja arkkitehtuuristen hajujen suhde, sekä niiden syyt.

Koodihaju	**Arkkitehtuurinen haju**	**Syy**
God Class[a]	*SPF, Connector Envy*	Toiminnallisuus liian keskittynyttä, liialliset vastuut
Misplaced Class[b]	*Cyclic Dependencies, SPF*	Riippuvuuksia enemmän komponentin ulkopuolelle kuin sisäpuolelle

[a]Luokka, joka tekee suuren osan järjestelmän toiminnallisuudesta.
[b]Luokka, joka on väärässä moduulissa/komponentissa.

Taulukon 4.5 suhteista nähdään hieman samanlaisuutta, kuin mitä taulukko 4.4 näytti syinä aspektiorientoituneille koodihajuille. Suurimmat ongelmat olioperäisissäkin hajuissa tulivat liiallisista riippuvuuksista ja huonosta vastuiden jaosta, jotka saattavat johtaa yksittäisten komponenttien paisumiseen. Yksittäisten komponenttien paisuminen taas luo helposti huonoa kompositiota ja toistuvaa toiminnallisuutta, josta äärimmäisessä tapauksessa järjestelmässä on vain muutama komponentti, joiden harteilla on iso osa toiminnallisuuden toteuttamisesta. Ongelmat kooditasolla täten helposti realisoituvat arkkitehtuurisen tason ongelmiksi, jolloin pienet ongelmat saattavat muodostua suuriksi ja vaikeasti ratkaistaviksi ongelmiksi.

Tarkastelu voidaan keskittää myös aiheisiin kuten mistä koodihajut johtuvat tai mitkä tekijät koodihajuissa vaikuttavat arkkitehtuuristen hajujen esiintymiseen järjestelmissä. Näiden asioiden tarkastelun avuksi on luotu erityisen näkymä koodihajujen tunnistamiseksi ja sen avulla voidaan tunnistaa muun muassa kytkentään ja koheesioon vaikuttavia koodihajuja (Fontana, Ferme & Zanoni, 2015).

Taulukko 4.6: Suoraan kytkentään ja koheesioon suhteessa olevat koodihajut.

Koodihaju	**Vastuussa**	**Vaikuttaa**
God Class	Suuret luokat	Koheesio & Kytkentä
Long Method	Pitkät metodit	Koheesio & Kytkentä
Dispersed Coupling	Laajalle hajautunut kytkentä	Koheesio & Kytkentä
Shotgun Surgery	Replikoituvasta muutoksen vaatimasta kohdealueesta	Koheesio

Taulukon 4.6 tekijöistä nähdään, että ongelmat syntyvät, jos jollakin luokalla on seuraavia ominaisuuksia: liian suuri koko; riippuvuuksia paljon muihin luokkiin; toiminnallisuus hajautunut niin laajalle, että sen muuttaminen vaatii usean muutoksen eri kohteisiin. Näitä samoja ominaisuuksia on mahdollista tarkastella arkkitehtuurisella tasolla niin, että luokkien sijaan tarkastelu kohdistuu komponentteihin.

On edellisten tarkasteluiden nojalla siis selvää, että hyvä kytkennän ja koheesion taso ovat tavoteltavia ominaisuuksia sekä kooditasolla että ark-

kitehtuuritasolla. Merkkittävä tekijöitä arkkitehtuuristen hajujen synnyssä olivat nimittäin vastuut, riippuvuudet, koheesio ja kytkentä. Testattavuuden kannalta ohjelmistojärjestelmän rakennetta voidaan arvioida esiteltyjen arkkitehtuurisiin hajuihin vaikuttavien tekijöiden näkökulmasta ja käyttämällä hyväksi luvussa 4.2 esiteltyjä mittareita.

4.4 Havaitut testattavuuteen vaikuttavat tekijät

Seuraavalla sivulla olevan taulukkoon 4.7 on kerätty kootusti tutkielman aikana löydetyt tekijät, jotka testattavuuteen vaikuttavat rakenteellisella tasolla eli tasolla, jossa rakennetta pystytään analysoimaan ja arvioimaan erilaisten näkökulmien avulla. Taulukkoon on pyritty keräämään vain keskeisiä asioita, joiden tarkastelu onnistuu sekä arkkitehtuurisella tasolla, että kooditasolla. Kooditasolla tarkastelu voidaan aloittaa tutkimalla löydettyjä ominaisuuksia yksittäisille luokille, jotka ovat sidoksissa vahvasti tärkeisiin arkkitehturaalisiin elementteihin, kun taas arkkitehtuurisella tasolla paino voidaan kiinnittää laajempiin kokonaisuuksiin, kuten komponettien ja niiden ominaisuuksien tarkasteluun.

On kuitenkin huomattava, että ominaisuudet jakautuvat varsinkin järjestelmän tasolla pelkästään laadullisiin ominaisuuksiin, joiden mittaaminen suoraan ei ole edes kovin mahdollista. Tällöin analyysin avuksi on otettava esimerkiksi dokumentaatiota järjestelmästä, komponenttien suunnittelusta vastuussa olevien ihmisiä, koko komponentin toteuttavaa ryhmää tai yleisesti koottua ryhmää, jonka kanssa komponentteja ja rakennetta tarkastellaan yhdessä. Näin pystytään muodostamaan hyvä kuva pelkästään järjestelmän testattavuuden tasosta ilman metrisiä mittareita, jotka vaativat jotain formalismia taustalleen virheellisten tuloksien minimoimiseksi.

Kaikki muut ominaisuudet ovat kuitenkin sekä laadullisia että mitattavia ominaisuuksia. Metrisinä ne voidaan nähdä siten, että monessa jokin määrällinen ominaisuus korostuu. Esimerkiksi mitä enemmän riippuvuuksia yhdellä luokalla on, sitä enemmän siinä piilee mittarien mukaan ongelmia testattavuuden kannalta. Totuus voi kuitenkin olla toinen ja tämän takia jokainen mittari pitäisikin nähdä enemmän indikaattorina mahdollisille ongelmille. Oikeat ongelmien syyt löytyvät mittareiden osoittamien ongelmakohtien tarkemmalla analyysillä. Tässä analyysissä voidaan miettiä esimerkiksi mitkä tekijät ohjelmistojärjestelmässä ovat luoneet ongelmallisen tilanteen ja löytyykö ongelmaan ratkaisua.

Olio-ohjelmoinnista on määritelty klassisia metriikoita (Chidamber & Kemerer, 1994) ja taulukosta löytyvät ominaisuudet heijastelevat vahvasti näiden metriikoiden ideoita. Testattavuuden analysointi on täten mahdollista samalla, kun arvioidaan järjestelmän muita ei-toiminnallisia vaatimuksia.

Taulukko 4.7: Testattavuuteen vaikuttavat tekijät arkkitehtuurisella tasolla. Tarkennus-sarake antaa lisätietoa menetelmästä ja Menetelmä-sarake kuvaa miten niitä voidaan havaita. Lähde-sarake kertoo mikä tutkimus on tutkinut aihetta ja mahdollisesti esittänyt jonkun menetelmistä.

Tekijä	**Tarkennus**	**Menetelmä**	**Lähde**
Järjestelmän ominaisuudet	Komponenttien testattavuus, vastuut, vaatimukset, havaittavuus, hallittavuus, jäljitettävyys	Analysointi, pentagrammimalli	Baudry et al. (2003); Binder (1994); Gao & Shih (2005); Joshi & Sardana (2014)
Kompleksisuus	Kompleksinen toteutus, kompleksiset perintäsuhteet	Metriikat, UML, Graafimalli	Baudry, Traon & Sunye (2002); Baudry et al. (2003); Binder (1994); Dubey & Rana (2011)
Perintä	Ongelmalliset ja liialliset perintäsuhteet	Metriikat, UML, Graafimalli	Baudry, Traon & Sunye (2002); Dubey & Rana (2011)
Kytkentä	Itsekäyttö, suuret kytkentätasot	Metriikat, UML, Graafimalli, hajut	Dubey & Rana (2011); Joshi & Sardana (2014)
Riippuvuudet	Syklimäisyys, määrä, antimalli	UML, graafimalli, hajut	Baudry, Traon & Sunye (2002); Joshi & Sardana (2014); Jungmayr (2002)
Koheesio	Toiminta huonosti keskitettyä	Metriikat, hajut, analysointi	Garcia et al. (2009); Joshi & Sardana (2014)
Vastuut	Vastuut jakautuneet huonosti	Hajut, UML	Garcia et al. (2009)

5 Pohdinta

Testattavuuteen löydetyt tekijät ovat osoittautuneet mielestäni hyvin keskeisiksi oliojärjestelmien ominaisuuksiksi, jotka heijastelevat olio-ohjelmoinnin hyviä käytänteitä. Taulukossa 4.7 olevien ominaisuuksien huomioon ottaminen ja noudattaminen nostaa ohjelmistojen laatua myös muullakin kuin vain testattavuuden saralla. Esimerkiksi järjestelmän modulaarisuuden parantaminen vaikuttaa sekä muunneltavuuteen että testattavuuteen.

Luvussa 4 esiin tulleet metriikat testattavuudesta ovat hieman vaikeakäyttöisiä oikeissa projekteissa, koska usein järjestelmää ei mallinneta täydellisesti ja saattaa olla, ettei järjestelmän visuaalinen kuvaus noudattele täysin, tai edes etäisesti, UML-standardia. On myös vaikeaa määritellä, mitä kaikkea pystytään metrisesti löytämään ilman laadullista analyysiä. Ilman metriikoiden osoittamien ongelmakohtien analyysia saatetaan helposti joutua tilanteeseen, jossa metriikka on sokea todellisille ongelmille tai hälyttää vääristä virheistä.

Aliluvussa 4.3 esille nousseet arkkitehtuuriset hajut ja riippuvuusgraafeista saatava data aliluvussa 4.2.1 ovat kuitenkin hyvinkin potentiaalisia keinoja ohjelmistokehittäjän tueksi löytämään ongelmakohtia projekteista. Näiden automatisointi olisi mielestäni hyvin läpimurtavaa, koska se helpottaisi paljon kehittäjän työtä. Arkkitehtuurista analyysia suorittavia työkaluja ei tällä hetkellä kovinkaan paljoa ole ja harva niistä edes ottaa huomioon esimerkiksi arkkitehtuurisia hajuja, ja keskittyy enemmän koodihajuihin ja niiden löytämiseen[5]. Tämä painottuneisuus johtuu todennäköisesti siitä syystä, että arkkitehtuurisia hajuja on vaikea tuoda ilmi koneellisesti staattisen analyysin keinoin, koska ohjelmistojärjestelmän rakenteen taustalla on jokin ihmisen tekemä piilevä päätös, jota ei voida havaita välttämättä kooditasolta ollenkaan.

Testattavuuteen vaikuttavia arkkitehtuurisia hajuja pitäisi tutkia lisää ja muodostaa niistä yhtä kattava katalogi kuin koodihajuista on muodostettu[6]. Näihin arkkitehtuurisiin hajuihin voitaisiin yhdistää testattavuuteen vaikuttavat hybridihajut, jolloin olisi mahdollista löytää arkkitehtuurisia ongelmakohtia kooditasolta alkaen. Tästä aiheesta tehtyjä tutkimuksia on kuitenkin melko vähän, ja vielä harvempi ottaa mukaan edes testattavuutta, mikä on harmi koska testaus on tärkeä osa jokaisen ohjelmiston elinkaarta.

Tulevaisuuden tutkimuksissa voitaisiinkin painottaa enemmän testattavuutta, koska nyt niitä löytyy vaihtelevasti kyseisestä aiheesta, ja ne painottavat eri asioita. Testattavuutta laatutekijänä pitäisi tutkia enemmän ja varsinkin rakenteen ja tätä kautta eri arkkitehtuuristen tyylien vaikutusta testattavuuteen.

[5]Esimerkkinä analyysityökalu SonarQube: `http://www.sonarqube.org`

[6]Koodihajut katalogina: `https://refactoring.guru/catalog`

6 Yhteenveto

Testattavuus näkyy ohjelmistojen arkkitehtuurissa niinä laadullisina ja metrisinä rakenteellisina ominaisuuksina, jotka korostavat testattavuutta. Laadulliset ominaisuudet voivat olla jokin komponentin konkreettinen ominaisuus, joka helpottaa testausprosessia huomattavasti, tai se voi olla komponenttien vastuiden ja vaatimusten ymmärtäminen niin, että testausta on helpompi tehdä. Järjestelmän testattaviksi ominaisuuksiksi voidaan määritellä sen havaittavuuden, hallittavuuden ja jäljitettävyyden tasot. Nämä pätevät myös yksittäisiin komponentteihin ja niiden tarjoamiin palveluihin, jotka nähdään usein metodeina ja rajapintoina.

Metrisinä tekijöinä voidaan nähdä rakenteen kompleksisuus, perinnän taso, kytkennän määrä, riippuvuuksien laatu ja määrä, koheesion taso ja vastuiden jakautuminen. Kaikkia näitä on mahdollista tarkastella UML:n luokka- ja komponenttikaavion kaavion sekä riippuvuusgraafin avulla. Hieman toisenlaista analyysiä voidaan tehdä etsimällä visuaalisesta näkymästä testattavuuteen vaikuttavia arkkitehtuurisia hajuja (*Cross-Module Cycle, Connector Ency, Scattered Parasitic Functionality*) ja niihin vaikuttavia hybridihajuja.

Lähteet

Baudry, B., Y. Le Traon & G. Sunye. 2002. Testability analysis of a UML class diagram. Teoksessa *Eighth IEEE Symposium on Software Metrics, 2002. Proceedings.* IEEE. Sivut 54–63.

Baudry, Benoit, Yves Le Traon, Gerson Sunyé & Jean-Marc Jézéquel. 2003. Measuring and improving design patterns testability. Teoksessa *Software Metrics Symposium, 2003. Proceedings. Ninth International.* IEEE. Sivut 50–59.

Bengtsson, PerOlof. 1998. Towards maintainability metrics on software architecture: An adaptation of object-oriented metrics. Teoksessa *First Nordic Workshop on Software Architecture, Ronneby.* Sivut 87–91.

Bertran, Isela Macia. 2011. Detecting architecturally-relevant code smells in evolving software systems. Teoksessa *Proceedings of the 33rd International Conference on Software Engineering.* ACM. Sivut 1090–1093.

Binder, Robert V. 1994. "Design for Testability in Object-oriented Systems." *Communications of the ACM.* 37(9):87–101.

Bruntink, M. & A. van Deursen. 2004. Predicting class testability using object-oriented metrics. Teoksessa *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on.* IEEE. Sivut 136–145.

Chidamber, S. R. & C. F. Kemerer. 1994. "A metrics suite for object oriented design." *IEEE Transactions on Software Engineering* 20(6):476–493.

Clements, Paul, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers & Reed Little. 2002. *Documenting Software Architectures: Views and Beyond.* Pearson Education.

de Andrade, Hugo Sica, Eduardo Almeida & Ivica Crnkovic. 2014. Architectural Bad Smells in Software Product Lines: An Exploratory Study. Teoksessa *Proceedings of the WICSA 2014 Companion Volume.* WICSA '14 Companion. ACM. Sivut 12:1–12:6.

Dubey, Sanjay Kumar & Ajay Rana. 2011. "Assessment of Maintainability Metrics for Object-oriented Software System." *ACM SIGSOFT Software Engineering Notes* 36(5):1–7.

Eickelmann, Nancy S. & Debra J. Richardson. 1996. What Makes One Software Architecture More Testable Than Another? Teoksessa *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops.* ISAW '96. ACM. Sivut 65–67.

Fontana, Francesca Arcelli, Vincenzo Ferme & Marco Zanoni. 2015. Towards Assessing Software Architecture Quality by Exploiting Code Smell Relations. IEEE. Sivut 1–7.

Freedman, Roy S. 1991. "Testability of Software Components." *IEEE Transactions on Software Engineering* 17(6):553–564.

Galster, Matthias. 2011. Dependencies, Traceability and Consistency in Software Architecture: Towards a View-based Perspective. Teoksessa *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*. ECSA '11. ACM. Sivut 1:1–1:4.

Gao, Jerry & Ming-Chih Shih. 2005. A component testability model for verification and measurement. Teoksessa *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*. Vol. 2 IEEE. Sivut 211–218.

Garcia, J., D. Popescu, G. Edwards & N. Medvidovic. 2009. Identifying Architectural Bad Smells. Teoksessa *13th European Conference on Software Maintenance and Reengineering, 2009. CSMR '09*. IEEE. Sivut 255–258.

Garlan, David. 2000. Software Architecture: A Roadmap. Teoksessa *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. ACM. Sivut 91–101.

Gorton, Ian. 2011. Understanding Software Architecture. Teoksessa *Essential Software Architecture*. Springer Berlin Heidelberg. Sivut 1–15.

ISO/IEC. 2010. ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Tekninen raportti.
URL: *http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733*

ISO/IEC/IEEE. 2011. "ISO/IEC/IEEE Systems and software engineering – Architecture description." *ISO/IEC/IEEE 42010:2011*. Sivut 1–46.

Joshi, Madhura & Neetu Sardana. 2014. Design and code time testability analysis for object oriented systems. Teoksessa *Computing for Sustainable Global Development (INDIACom), 2014 International Conference on*. IEEE. Sivut 590–592.

Jungmayr, S. 2002. Identifying test-critical dependencies. Teoksessa *Software Maintenance, 2002. Proceedings. International Conference on*. IEEE. Sivut 404–413.

Khan, R. A. & K. Mustafa. 2009. "Metric Based Testability Model for Object Oriented Design (MTMOOD)." *SIGSOFT Softw. Eng. Notes* 34(2):1–6.

Kruchten, Philippe. 1995. Architectural Blueprints—The “4+1” View Model of Software Architecture. IEEE. Sivut 42–50.

Macia, I., A. Garcia, C. Chavez & A. von Staa. 2013. Enhancing the Detection of Code Anomalies with Architecture-Sensitive Strategies. IEEE. Sivut 177–186.

Macia, Isela, Alessandro Garcia, Arndt von Staa, Joshua Garcia & Nenad Medvidovic. 2011. On the Impact of Aspect-Oriented Code Smells on Architecture Modularity: An Exploratory Study. IEEE. Sivut 41–50.

Mo, R., Y. Cai, R. Kazman & L. Xiao. 2015. Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells. Teoksessa *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*. Sivut 51–60.

Object Management Group. 2015. “OMG Unified Modeling Language TM (OMG UML), version 2.5.”.
URL: *http://www.omg.org/spec/UML/2.5/PDF/*

Ran, Alexander. 1998. Architectural Structures and Views. Teoksessa *Proceedings of the Third International Workshop on Software Architecture*. ISAW ’98. ACM. Sivut 117–120.

Rozanski, Nick & Ein Woods. 2011. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. 2 ed. Addison-Wesley Professional.

Solms, Fritz. 2012. What is Software Architecture? Teoksessa *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*. SAICSIT ’12. ACM. Sivut 363–373.

Vale, Gustavo, Eduardo Figueiredo, Ramon Abilio & Heitor Costa. 2014. Bad Smells in Software Product Lines: A Systematic Review. IEEE. Sivut 84–94.

Voas, Jeffrey M. & Keith W. Miller. 1992. Improving the software development process using testability research. Teoksessa *Software Reliability Engineering, 1992. Proceedings., Third International Symposium on*. IEEE. Sivut 114–121.

Voas, Jeffrey M. & Keith W. Miller. 1995. “Software Testability: The New Verification.” *IEEE Software* 12(3):17–28.