Testattavuuden ilmeneminen ohjelmistoarkkitehtuurissa

Kristian Wahlroos

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 9. maaliskuuta 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

Testattavuuden ilmeneminen ohjelmistoarkkitehtuurissa

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	9. maaliskuuta 2016	15

Tiivistelmä — Referat — Abstract

tiivistelmä (100-200 sanaa; kenelle, miksi, millaisessa ympäristössä; tutkimuskysymys; tulokset; impakti)

Avainsanat — Nyckelord — Keywords
ohjelmistoarkkitehtuuri, testattavuus, 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 (IEEE, 2000, s. 4). Suunnittelematon arkkitehtuuri eli niin sanottu *Big Ball of Mud* saattaa pahimmassa tapauksessa olla yksi epäonnistuneen projektin tunnusmerkkejä, kun taas hyvin muodostettu arkkitehtuuri auttaa eri sidosryhmien välisessä kommunikaatiossa, tekee tuotettavasta ohjelmistojärjestelmästä helposti lähestyttävän, helpottaa ohjelmiston testausta sekä validointia ja on yksi suurimmista onnistumistekijöistä projektin jatkon kannalta.

Ohjelmistojen ylläpidettävyys lasketaan yhdeksi ei-toiminnalliseksi laatuvaatimukseksi ja siihen kuuluu vahvasti ohjelmistojen testattavuus, 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, että kuinka helppoa järjestelmää on ylipäätänsä testata. Kuitenkin arkkitehtuurin ja testattavuuden yhteys on hieman häilyvä, vaikka arkkitehtuuri koskee jokaista kehittäjää. Tämä tutkimus pyrkii selventämään arkkitehtuurin ja testattavuuden suhdetta ja sitä, miten testattavuus tulee ilmi ohjelmistoarkkitehtuurissa.

1.1 Tutkimuskysymys

Tutkimus vastaa kysymykseen *miten testattavuus näkyy ohjelmistojen arkkitehtuurissa.* Itse kysymys voidaan jakaa vielä tarkentaviin kysymyksiin: *mitä on testattavuus ohjelmistojärjestelmissä, miten arkkitehtuuria voidaan kuvata, miten arkkitehtuurista luotu näkymä kertoo jotain järjestelmän testattavuudesta.*

1.2 Tutkimusmenetelmä

Tutkimuskysymykseen vastataan kirjallisuuskatsauksen avulla ja tutkimalla, että mitkä arkkitehtuuriset päätökset ja elementit ovat hyvän testattavuuden tekijöitä. Hyvän testattavuuden havaitsemiseen etsitään näkymää nykyisistä menetelmistä kuvata arkkitehtuuria. Näitä menetelmiä ovat pääasiassa erilaisten näkymien ja näkökulmien joukko, joista tässä tutkimuksessa pyritään etsimään kehittäjän näkökulmasta kaikista relevantein kokoelma.

1.3 Rajaus

Tutkimuksessa keskitytään pääosin ainoastaan oliopohjaisen arkkitehtuurin kuvaamiseen ja sen yleisimpiin käytänteisiin. Testaustavoista on keskitytty yksikkö -ja hyväksymistesteihin. Arkkitehtuurin mallintamisesta on jätetty ADL:t (architecture description languages) pois.

1.4 Tutkimuksen rakenne

Kappaleessa 2 käydään läpi ohjelmistoarkkitehtuuria käsitteenä ja luodaan pohja eri kuvaustavoille ohjelmistoarkkitehtuuria varten. Samalla kuvataan, että miten ja mistä näkymä voidaan luoda ohjelmistojärjestelmälle. Kappale 3 keskittyy ohjelmistolle asetettaviin laadullisiin vaatimuksiin, joista kappale 4 tarkentaa testattavuutta. Kappale 5 etsii arkkitehtuurin ja testattavuuden välistä suhdetta ja viimeinen kappale keskittyy testattavuuden parantamiseen arkkitehtuurin avulla.

2 Ohjelmistoarkkitehtuuri

Arkkitehtuuri ohjelmistoissa on käsite, josta moni kehittäjä on tietoinen mutta joka ei ole kovin yksiselitteinen eikä siitä ole yksimielisiä määritelmää (Solms, 2012, s. 363). Se kuvaa kuitenkin vähintään aina abstraktilla tasolla toteutettavaa ohjelmistojärjestelmää niin, että on mahdollista keskittyä vain suppeaan alueeseen jättämällä pois eri sidosryhmien kannalta epärelevantteja asioita. Kuvaus tapahtuu usein visuaalisella tavalla, koska visuaalisesta mallista on nopea ja helppo saada ulos tarvittava informaatio.

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, mallina joka ottaa huomioon ei-toiminnalliset (non-functional) vaatimukset ja yleisenä abstraktiona.

Rakenteen kuvaajana arkkitehtuuri määrittelee ohjelmistojärjestelmän sisäistä rakennetta, joka koostuu useista komponenteista sekä moduuleista, jotka yhdistävät ja ryhmittelet komponentteja. Erilaiset moduulit tarjoavat aina tietyn toiminnallisuuden ja tämän kautta erilaiset vastuut on jaettu ohjelmistojärjestelmän sisällä loogiisiin kokoelmiin.

Komponenttien välisen kommunikaation mallintaminen tapahtuu, kun ohjelmistojärjestelmää jaetaan erilaisiin komponentteihin. Tavoitteena on kertoa mallintamalla, että mitkä komponentit tai moduulit ovat vuorovaikutuksessa toistensa kanssa ja millä tavoin. Yleisin kommunikaatiotapa on esimerkiksi suorat funktiokutsut komponenttien välillä, joiden mallinnus selventää esimerkiksi informaation kulkua järjestelmässä.

Ei-toiminnaliset vaatimukset tulevat kunnolla esille vasta arkkitehtuurin avulla ja arkkitehtuurin mallinnuksen avulla voidaan määrittää miten ohjelma suorittaa sille määrättyä tehtävää sen sijasta, että mallinnettaisiin ainoastaan mitä ohjelma tekee.

Arkkitehtuurin avulla on mahdollista abstrahoida sidosryhmille ohjelmistojärjestelmää helpommin lähestyttäväksi, jolloin kommunikaatio eri osapuolten välillä helpottuu. Abstrahoinnin avulla pystytään yksinkertaistamaan

järjestelmän konkreettista toteutusta ja suorittamaan arkkitehtuurista erittelyä (architectural decomposition), jossa muodostetaan epärelevanteista komponenteista mustia laatikoita (black boxes). Mustien laatikoiden ideana on piilottaa komponenttien sisäistä toteutusta eri abstraktiotasoilla, jolloin arkkitehtuurista voidaan muodostaa eri tarkkuustason malleja.

Hieman samanlainen jaottelu on määritelty myös Solmsin tutkimuksessa (Solms, 2012, s. 368-369). Siinä ohjelmistoarkkitehtuuri-määritelmä jaetaan kolmeen eri luokkaan: 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 (Gorton, 2011, s. 2-7), mutta tarkentaen kumpaakin omat kuvaustavat. Korkean tason abstraktioita mallinnetaan erilaisten näkymien kautta tai käyttämällä IEEE:n määrittelemiä käytänneitä (IEEE, 2000, s. 4-5). Rakenteita korostettaessa mallinnuksessa käytetään usein UML-kaavioita, koska UML määrittelee ohjelmiston arkkitehtuuria osina, joita pystytään rekursiivisesti tarkentamaan haluttaessa. Tarkennuksen avulla osat pystytään kuvaamaan kommunikoivan keskenään erilaisten rajapintojen kautta, osia yhdistäviä suhteita pystytään tarkastelemaan ja erilaisia rajoitteita pystytään luomaan osien välille.

Kolmas määritelmä on paljon laajempi näkemys ohjelmistoarkkitehtuurista, koska se kuvaa ohjelmistojärjestelmän käsitteistön ja ominaisuudet siinä ympäristössä, jossa ne ilmeentyvät elementtien, suhteiden ja suunnittelun periaatteiden kautta. Peruskäsitteistö tarjoaa sen käsitteistön, jonka avulla sovelluslogiikka voidaan määritellä. Esimerkiksi määrittelemällä, että jokin järjestelmä on tyypiltään palvelin, kun taas ominaisuudet liittyvät ohjelmistojärjestelmän laadullisiin ominaisuuksiin. Periaatteet voidaan nähdä järjestelmän keskeisinä suunnittelurajoitteina (core design constraints), joiden kokonaisuus muodostaa järjestelmän arkkitehtuurisen tyylin. Ohjelmistojärjestelmän arkkitehtuurin tyyli voi esimerkiksi olla väylät ja suodattimet (pipes and filters).

Näistä kahdesta eri jaottelusta ohjelmistoarkkitehtuuriin voidaan nähdä merkittävinä tekijöinä tarpeen kuvata ohjelmiston rakenteellisuutta, komponenttien välistä kommunikaatiota ja laadullisia vaatimuksia. Kaikki nämä edellämainitut ominaisuudet on saatava upotettua ohjelmistojärjestelmästä luotavaan arkkitehtuuriseen malliin niin, että ne vastaavat muunmuassa seuraaviin kysymyksiin (Rozanski and 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 näitä elementtejä. Kuitenkaan arkkitehtuurista luotava malli ei saisi olla monoliittinen malli, joka pyrkii kuvamaan kaiken yhdessä mallissa, koska arkkitehtuuria ei ole mahdollista kuvata vain yhden mallin avulla Rozanski

and Woods (2011). Tämä johtuu siitä, että monoliittinen malli on erittäin vaikea ymmärtää jokaiselle osapuolelle, siitä on vaikeaa löytää arkkitehtuurin tärkeimmät ominaisuudet (features) ja se on usein puutteelinen, jäljessä eikä vastaa enää nykyistä ohjelmistojärjestelmää. Ratkaisu tähän on jakaa malli useisiin toisiinsa liittyviin näkymiin (views), jotka esittävät jokainen yhden näkökulman (viewpoint) järjestelmän arkkitehtuuriin keräämällä yhteen toiminnalliset piirteet sekä laadulliset ominaisuudet (Rozanski and Woods, 2011, s. 33-34; Gorton, 2011, s. 8-9; Ran, 1998, s. 117). Tämän avulla voidaan tarkastella saavuttaako järjestelmä sille asetetut tavoitteet.

2.2 Arkkitehtuurin kuvaus

Arkkitehtuurin kuvaaminen sallii kommunikoinnin eri sidosryhmien välille ja mahdollistaa järjestelmän tarkastelemisen vain niiltä osin, jotka tiettyä sidosryhmää kiinnostaa (Razavizadeh et al., 2009, s. 329; Brøndum and Zhu, 2010, s. 60). Kuvaaminen tapahtuu näkymien avulla, jotka auttavat erottelemaan (separation of concerns) arkkitehtuuria pienempiin palasiin (Galster, 2011, s. 2). Itse näkymät luodaan taas käyttämällä ennaltamäärättyjä näkökulmia, jotka edustavat tiettyä tai useampaa sidosryhmälle tärkeää asiaa.

Hyvä kommunikointi sidosryhmien kanssa on aina tärkeää, joista ääripäinä ovat kehittäjät ja asiakkaat. Kehittäjät haluavat usein todella tarkkaa usean eri tason ja näkökulman abstraktia kuvausta luotavasta järjestelmästä (Ran, 1998, s. 120) tai ymmärtääkseen jo valmiiksi luotua järjestelmää Razavizadeh et al. (2009), kun taas asiakkaat haluavat nähdä järjestelmän toiminnallisuuden juuri heitä koskevien asioiden kannalta ja usein mahdollisimman korkealla tasolla.

2.2.1 Näkymät

Arkkitehtuuristen näkymien tehtävänä on kuvata ne näkökulmat arkkitehtuurista, jotka ovat merkityksellisiä itse asialle, jota näkymä haluaa painottaa (Rozanski and Woods, 2011, s. 34; May, 2005, s. 15). Yksi tunnetuimmista määritelmistä arkkitehtuuriselle näkymälle on Krutchenin 4+1 näkymämalli (4+1 View Model) (Gorton, 2011, s.7-8).

4+1 -mallissa arkkitehtuuri kuvataan neljän näkymän avulla: looginen, prosessi, fyysinen ja kehitys. Looginen näkymä kuvailee esimerkiksi luokkakaavioiden avulla ohjelmistojärjestelmän elementtejä ja niiden välisiä suhteita tarkentaen järjestelmän rakennetta, prosessinäkymä keskittyy ajonaikaisen suorituksen kuvaamiseen tarkentamalla muunmuassa miten samanaikaisuuden hallinta tapahtuu järjestelmässä, fyysinen näkymä keskittyy siihen miten järjestelmän eri komponentit kuvautuvat fyysiselle laitteistolle jossa komponenttia ajetaan ja lopuksi kehitysnäkymä (development view) keskittyy järjestelmän sisäiseen toteutukseen tarkemmalla tasolla kuvailemalla sisäkkäisia pakkauksia tai luokkahierarkiaa. Jokainen näkymä voidaan liittää

osaksi toista näkymää skenaarioiden avulla, jotka heijastelevat järjestelmälle asetettuja vaatimuksia, jolloin skenaariot voidaan nähdä ikäänkuin liimana muuten erillisille näkymille.

Muita näkymämalleja arkkitehtuuriin on useita, joista jokainen kuitenkin tuo esille vähintään jollain tavalla ohjelmistojärjestelmän rakennetta ja yhteyksiä. Tästä esimerkkinä tunnettu *Views and Beyond* -malli (Gorton, 2011, s.8). Siinä arkkitehtuurinen malli kuvataan kolmella eri näkymällä: moduuli, joka on järjestelmän rakenteellinen näkymä kuvaten muunmuassa luokkia, pakkauksia, moduulien eriyttämistä (decomposition) sekä periytymistä; komponentti ja konnektori, joka kuvailee järjestelmän toiminnallista puolta sekä miten komponentit ovat yhteydessä toisiinsa; allokaatio, joka kuvailee miten prosessit kuvautuvat laitteistotasolla ja miten ne kommunikoivat keskenään.

Näiden kahden esitellyn mallin lisäksi on olemassa joukko eri tarkoituksiin sopivia näkymämalleja, joilla on omat vahvuutensa sekä heikkoutensa. Yhteensä viittä eri mallia vertailtiin Mayn tutkimuksessa (May, 2005), jossa kartoitettiin mahdollisimman laajaa näkymien muodostamaa näkökulmien joukkoa, jotka kattaisivat mahdollisimman laajalti ohjelmistoarkkitehtuurin aluetta (domain). Vertailtavat arkkitehtuurit olivat 4+1-malli, SEI:n (Software Engineering Institute) *Views and Beyond*-malli, ISO:n (International Organization for Standardization) *Reference Model of Open Distributed Processing*, Siemens'n *Four View Model* ja *Rational Architecture Description Specification*. Malleja vertailtiin tutkimalla, että kuinka niiden tarjoamat näkökulmat painottavat kolmeen eri kategoriaan liittyviä asioita: sidosryhmät, laadulliset tekijät ja rakenteelliset ominaisuudet. Joukko, joka painotti mahdollisimman laajasti jokaisen osa-alueen ominaisuuksia muodostettiin lopulta seuraavista osista: *Views and Beyond* -mallin kaikki kolme näkymää ja *Rational ADS*:n vaatimukset-näkökulma (requirements viewpoint). Syy miksi juuri nämä valittiin oli se, että *Views and Beyond* sopii erittäin hyvin mallin pohjaksi, koska se kuvaa järjestelmän rakennetta kattavasti, sen näkökulmat ovat itsenäisiä ja se kokonaisuudessaan limittyy hyvin 4+1-mallin sekä Siemens'n mallin kanssa. Kuitenkaan se ei sisällä tarvittavia näkymiä kuvaamaan eri sidosryhmille tärkeitä asioita, kuten esimerkiksi loppukäyttäjiä varten oleva näkymä puuttuu kokonaan. Tämän takia mukaan on otettu *Rational ADS*, jonka vaatimukset-näkökulma koostuu neljästä eri näkymästä: ei-toiminnalliset vaatimukset, alue (domain), käyttötapaus ja käyttökokemus.

2.2.2 Näkökulmat

Rozanskin ja ym. (Rozanski and Woods, 2011, s. 36-42) määrittelevät, että näkökulmat (viewpoints) liittyvät vahvasti arkkitehtuurisiin näkymiin. Näkökulma voidaan määritellä olevan kokoelma malleja (patterns), templaatteja sekä konventioita, joiden avulla näkymä pystytään rakentamaan. Näkökulmien joukko voidaan taas jakaa seitsemään eri osaan: kontekstinen, funktionaalinen, informaatiollinen, samanaikaisuudellinen, kehittämisellinen,

käyttöönotollinen ja operationaalinen. Näistä keskitytään tämän tutkimuksen myöhemmässä vaiheessa kehittämisen sekä funktionaalisuuden näkökulmaan. Kehittämisen näkökulmaan siksi, koska se ottaa huomieen niiden sidosryhmien asiat, jotka ovat mukana järjestelmän kehityksessä, testauksessa, ylläpidossa ja parantamisessa. Funktionaalinen näkökulma taas painottaa muunmuassa järjestelmän suorituksenaikaisia komponentteja sekä niiden rajapintoja.

IEEE:n virallinen määritelmä näkokulmalle on *a specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis* (IEEE, 2000, s. 4). Se on määritelmä, jota tullaan käyttämään myös tässä tutkimuksessa.

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 myös otettava huomioon kenelle näkymä on tarkoitettu, minkälaisia asioita halutaan näkymän painottavan ja millä abstraktiotasolla näkymän tulisi olla.

Esimerkiksi ohjelmakoodin kautta luotava UML-malli saattaa olla näkymä projektin rakenteeseen uusia ohjelmoijia varten, jotta voitaisiin helposti esittää kyseisen ohjelmistojärjestelmän päärajapinnat. Tällaisen mallin päätarkoitus on kuvata ohjelmistojärjestelmän rakennetta, mutta abstrahoida pois kaikki turhan tarkat kuvaukset komponenteista, joilla ei ole juuri sillä hetkellä merkitystä.

Jos järjestelmää ei ole vielä olemassa lainkaan, voidaan järjestelmän arkkitehtuuria lähteä ensin rakentamaan iteratiivisesti kolmen eri vaiheen kautta (Gorton, 2011, s. 97-115): määritellään arkkitehtuuriset vaatimukset; luodaan rakenne, joka tukee näitä vaatimuksia; testataan, että rakenne tukee vaatimuksia. Arkkitehtuurille asetetut vaatimukset syntyvät toiminnallisten ja sidosryhmien asettamien vaatimusten tuloksena, koska ne yhdessä luovat tärkeimmät vaatimukset ja rajoitukset ohjelmistojärjestelmälle. Esimerkkinä järjestelmälle asetettavista vaatimuksista saattaa olla, että sen on pystyttävä tukemaan useita eri käyttöliittymiä. Tämä vaatimus voidaan taas ryhmittää tiettyyn laatuvaatimukseen, kuten useat muutkin järjestelmälle asetetut vaatimukset (Ran, 1998, s. 117). Tässä tapauksessa vaatimus ryhmitettäisiin muokattavuus-laatuvaatimuksen alle. Seuraavassa vaiheessa valitaan tätä laatuvaatimusta tukeva tyyli (architectural style), jota koko järjestelmä joutuu noudattamaan. Esimerkiksi N-tason -malli tukee muokattavuutta hyvin (Gorton, 2011, s. 103). Tämän jälkeen järjestelmän keskeisimmät komponentit suunnitellaan ja sijoitetaan ne oikeisiin alueisiin, jotka valittu arkkitehtuurinen tyyli tuo tullessaan. Komponenttien väliset suhteet mallinnetaan pitämällä vastuut erillään, suhteet mahdollisimman vähäisinä sekä

käyttämällä hyödyksi piilottamista (architectural decomposition). Lopuksi pystytään luomaan skenaarioita ja testaamaan niiden avulla, että onko luotu arkkitehtuuri sopiva. Komponenttien määrittely mahdollistaa arkkitehtuurin rakenteen kuvaamisen ja siten järjestelmästä on mahdollista luoda erilaisia näkymiä erilaisille sidosryhmille. Rakennetta pystytään kuvaamaan N-tason -mallin avulla erittäin korkealla tasolla, jos halutaan tuoda esille juurikin näkökulma muokattavuuteen, mutta myös matalemmalla tasolla, kun ohjelmistokehittäjä haluaa ymmärtää miten komponentit kommunikoivat keskenään.

3 Ohjelmistojen laadulliset tekijät

Miten voidaan jakaa? Miten tuodaan esille?

3.1 Laatuvaatimukset

Keräys? Muodostus? Ryhmittely?

3.1.1 Toiminnalliset

3.1.2 Ei-toiminnalliset vaatimukset

3.2 Tradeoffit

3.3 Arviointi

4 Testattavuus laadullisena tekijänä

Testattavuus luetaan kuuluvan ylläpidettävyyden alle (ISO, 2011) ja on täten yksi laatuvaatimuuksista. Kuitenkaan se laatuvaatimuksena ei esimerkiksi asiakkaalle ole edes aina kovin näkyvä, mutta ohjelmistokehittäjälle se tulee esille useassakin kontekstissa. Vaikeasti testattava järjestelmä on myös vaikeasti ylläpidettävä, ja ylläpidettävä järjestelmä on sekä asiakkaalle, että kehittäjällä tärkeä ominaisuus.

Ohjelmistojärjestelmään kohdistuva testaus voidaan jakaa moneen eri luokkaan, joista kaksi yleisintä ovat yksikkö -ja integraatiotestaus. Yksikkötestauksessa varmistetaan järjestelmän komponenttien toiminnallisuutta itsenäisesti, kun taas integraatiotestauksessa testaus kohdistuu järjestelmän rajapintoihin ja komponenttien yhteistoiminnallisuuteen. Testauksen voidaan nähdä siis löytän itse viat, mutta testattavuus paikat, joissa viat voivat sijaita (Voas and Miller, 1995, s. 19). Testattavuus tukee täten testausta ja testausprosessia, koska sen avulla voidaan määritellä mikä on todennäköisyys sille, että järjestelmässä olevat viat löydetään (Voas and Miller, N.d., s. 114).

Testattavuuden arvioimisessa voidaan käyttää hyödyksi kolmea eri parametria (Baudry et al., N.d., s. 2): havaittavuus (observability), hallittavuus (controllability) ja vaativuus, joka tarvitaan suorittamaan tietty testi. Havaittavuus määrittelee miten mahdollista testien tuloksia on tarkastella ja hallittavuus sitä, kuinka paljon testattavaan komponentin tilaan voidaan vaikuttaa. Havaittavuus voidaan myös nähdä arvona, joka kertoo kuinka helppo on määrittää, että vaikuttavatko tietyt syötteet testituloksiin ja hallittavuus sitä, että kuinka helppo on tuottaa jokin tietty tulos syötteestä (Freedman, 1991, s. 554).

4.1 Testattavuuden merkitys

Huono testattavuus voi tarkoittaa järjestelmän kannalta ylläpidettävyyden huononemista ja täten aiheuttaa jatkuvaa ongelmaa kehityksen kannalta, koska testaus on usein tapahtuvaa toimintaa, jonka tarkoituksena on löytää vikoja järjestelmästä. Myöhäisessä vaiheessa löydetty vika saattaa tulla erittäin kalliiksi ja viedä pois useita työtunteja, jotka olisi pystytty käyttämään johonkin tuottavampaan tekemiseen, jos vaikeasti testattava järjestelmä ei olisi ongelman löytämistä hidastanut.

4.2 Huono testattavuus

Tietyt ominaisuudet vaikuttavat yleisesti ohjelmistojärjestelmissä negatiivisesti testattavuuteen. Tiedon katoaminen on yksi näistä (Voas and Miller, 1995, s. 20-22) ja se voidaan jakaa implisiittisen ja eksplisiittisen tiedon katoamiseen. Implisiittisessä katoamisessa useat eri parametrit samaan kohteeseen tuottavat myös saman testituloksen ja eksplisiittisessä tiedon katoamisessa

tietoa piilotetaan ulkoiselta näkymältä. Jälkimmäinen on hyvin yleinen tapa olio-ohjelmoinnissa ja liiallinen tiedon piilotus tuottaa vaikeuksia testauksessa, koska hallittavuus heikkenee.

Ongelmat syötteiden ja tulosten tarkastelussa johtavat helposti tarpeettomiin testeihin, joita on vaikea ymmärtää (Freedman, 1991, s. 554) ja vaikeasti testattavan komponentin ominaisuuksiksi voidaankin määritellä sen sisältävän ulos- ja sisääntulo parametrien epäjohdonmukaisuuksia. Toisinsanoen vaikeasti testattava komponentti siis tuottaa eri tuloksia eri ajoilla tai ei ikinä tuota haluttua tulosta.

4.3 Hyvä testattavuus

Hyvä testattavuus on järjestelmän laadulle hyvä ominaisuus (Voas and Miller, 1995, s. 20).

Isolating modules with high DDR (Voas and Miller, 1995, s. 23).

Jo suunnitteluvaiheessa voidaan kiinnittää huomiota testattavuuteen ja näin parantaa itse testausprosessia. Decomposition, moduulien korkeat DRR (domain/range -ratio), korkean DRR:n omaavat moduulit mah. pieniksi ja eriytetyiksi, hyvä rajapinta joka paljastaa tarpeeksi moduulin sisäistä tilaa (Voas and Miller, N.d., s. 117).

Testattavan komponentin ominaisuudet: testitapaukset ovat pieniä ja helposti luotavia, testitapaukset eivät ole itseään toistavia, ongelmat on helppo jäljittää tiettyihin komponentteihin (Freedman, 1991, s. 554)

4.4 Testattavuuden mittaaminen

Erilaisia testattavuuteen vaikuttavia mittareita on tutkittu paljon ja oliopohjaisista järjestelmistä voidaan havaita seuraavat testaukseen vaikuttavat tekijät, joiden suuri arvo korreloi vahvasti huonoon testattavuuteen (Dubey and Rana, 2011, s. 5):

- metodien määrä luokassa
- metodien määrä luokkahierarkiassa, jotka suoritetaan, kun luokan metodi suoritetaan
- luokkien määrä, johon luokalla on kytkentää
- periytymisaste

Nämä tekijät ovat negatiivisia asioita testauksen kannalta, koska suuri metodien määrä on merkki luokan liiallisesta kompleksisuudesta ja se johtaa testauksen monimutkaistumiseen. Suuri kytkennän määrä vaikeuttaa luokan itsenäistä testaamista ja suuri periytymisaste tuo luokalle mahdollisesti paljon perittyjä metodeja, jotka on toteutettu jossain muualla, jolloin luokka on riippuvainen erittäin paljon muiden luokkien toteutuksista.

Lisää testattavuuden mittareita yksikkötestauksen näkökulmasta on määritelty (Bruntink and van Deursen, 2004, s. 9). Siinä testitapausten kasvuun ja pituuteen vaikuttivat luokan:

- toisten luokkien määrä, joita luokka kutsuu tai jonka kenttiin se viittaa
- koko (rivien määrä)
- omien metodien määrä ja muiden luokkien metodien määrä, joita omat metodit käyttävät

On kuitenkin huomattavaa, että nämä ovat mittareita, joita voidaan suurimmaksi osaksi mitata vain lähdekoodin avulla. Osaa pystytään kuitenkin mittamaan myös UML:n luokkakaavion avulla ja se toimii hyvänä pohjana, kun halutaan mitata luokkien välisiä suhteita ja riippuvuuksia (Baudry, Traon and Sunye, N.d., s.). (Baudry et al., N.d., s. 3) määritteleekin testattavuuden anti-mallin (testability anti-pattern), jossa testattavuuteen negatiivisesti vaikuttavat kaksi suurta tekijää: luokkien välinen suuri interaktio ja itsekäyttö, jossa luokka on riippuvainen rekursiivisesti itsestään. Syitä näiden kahden esiintymiselle voi olla luokkien välinen suuri kytkentätaso, riippuvuuksien syklimäisyys ja komponenttien välinen suuri kommunikaatiotaso. Nämä tekijät vaikeuttavat paljon vastuiden jakamista ja kasvattavat tarvittavien testitapausten määrää.

UML-kaaviosta on mahdollista nähdä erilaisia asioita, kuin suoraan lähdekoodista. UML:n luokkakaavio toimiessa abstraktiivisena näkymänä suoraan lähdekoodiin, se tuo hieman korkeamman tason näkymän järjestelmään, mutta ilman liiallista tiedon piilottamista. Asioita, joita voidaan havaita UML:n avulla ja jotka vaikuttavat testattavuuteen negatiivisesti, esittää (Baudry, Traon and Sunye, N.d.) välttäväksi:

- Kompleksiset perintäsuhteet
- Abstraktien luokkien käyttö rajapintojen sijasta
-

4.5 Design for testability

Järjestelmä voidaan suunnitella tukemaan hyvää testattavuutta (Voas and Miller, 1995, s. 20).

5 Arkkitehtuurin vaikutus testattavuuteen

Testattavuus tulee esille arkkitehtuurisella tasolla hieman erilailla, kun ohjelmakooditasolla. Arkkitehtuurisella tasolla testattavuus vaikuttaa laajemmin ja sitä tutkitaan komponenttitasolla, jota tukee erittäin hyvin UML-kaaviot,

joiden avulla ohjelmistojärjestelmän rakenne saadaan kuvattua.

Mikä tekee yhdestä arkkitehtuurista testattavamman, kuin toisen? -> Tyylien käyttö parantaa testattavuutta, (Eickelmann and Richardson, 1996)

Testattavuus tulee ilmi korkealla tasolla integraatiotestauksessa (Eickelmann and Richardson, 1996) -> komponenttien rajapintojen toimivuus, miten komponetit antavat datan ja kontrollin eteenpäin.

Kolmentyyppisiä kriittisiä moduuleita: tehtävä, turvallisuus ja pääsykriittiset moduulit (Eickelmann and Richardson, 1996). Arkkitehtuurisellakin tasolla nämä pitäisi pitää erillään muista moduuleista.

5.0.1 Metrics for maintainability

Ylläpidettävyyttä voidaan mitata arkkitehtuurisella tasolla analysoimalla näkymää, josta nähdään esimerkiksi UML-rakennekaavion avulla järjestelmän rakennetta. Sitä voidaan mitata arkkitehtuurisista elementeistä seuraavien mittarien avulla (Bengtsson, 1998, s. 3): metodien määrä elementin rajapinnassa; paikkamerkkien (placeholder) määrä arkkitehtuurisille elementeille, joiden avulla elementti parametrisoidaan; elementtien välisten viestien määrä; elementtityyppien määrä, joita tarkasteltava elementti toteuttaa (implement); elementtien määrä, mitkä toteuttavat tarkasteltavan elementin tyypin; saatavilla olevien metodien lukumäärä muista elementeistä, jotka ovat yhteydessä tarkasteltavaan elementtiin; elementin metodien ja parametrien määrä.

Mittaaminen on mahdollista tehdä ilman lähdekoodia, jos näkymä on jo valmiiksi luotu. Näkymän avulla voidaan tuoda ongelmakohtia esille ohjelmistojärjestelmässä, koska moni metriikoista määrittää järjestelmän sisäistä riippuvuusastetta ja mitä suurempi aste on, sitä suurempi on mahdollisuus, ettei järjestelmä oli kovin ylläpidettävä.

5.1 Hyvä testattavuus arkkitehtuurissa?

Testattavuuden parantaminen erillisen patternin avulla (Coelho, Kulesza and Von Staa, 2005).

Tietyt arkkitehtuuriset tyylit sopivat tiettyihin testaustapoihin (critical module, sandwich, top-down, bottom-up, thread, big bang) (Eickelmann and Richardson, 1996). Kolmeen ensimmäiseen liittyvät vahvasti stubit ja test driverit.

-> tyylin tunnistaminen?

5.2 Sunnittelumallien vaikutus testaukseen

Mitä on? Miten vaikuttaa rakenteeseen? Mitä hyötyä? (Baudry et al., N.d.).

Composition-patternin analyysi -> kansio sisältää useita tiedostoja ja kansioita -> sykli. Voidaan helpottaa lisäämällä rajoite (constraint) tai rajapinta mielummin kuin abstraktiluokka.

Vaikeita patterneita testata: Mediator (similar to Observer) & Visitor.

5.3 UML

UML + graph model to compute all possible anti-patterns that affect testability (Baudry, Traon and Sunye, N.d.).

UML:n parantaminen uusilla notaatioilla (Baudry et al., N.d., s. 4). **create** luokka A luo luokan B, **use** luokka A voi kutsua mitä tahansa, paitsi B:n konstruktoria. **use_consult** kutsumat metodit, jotka eivät koskaan muokkaa B:n sisäistä tilaa, **use_def** -> jos vähintään yksi metodi muokkaa B:n tilaa.

5.4 Arkkitehtuuriset hajut

Arkkitehtuuriset suunnittelupäätökset, jotka negatiivisesti vaikuttavat järjestelmän elinkaareen ja muunmuassa testattavuuteen sekä ylläpidettävyyteen. Hajut eivät ole virheitä vaan vaikuttavat negatiivisesti laatuun (de Andrade, Almeida and Crnkovic, N.d., s. 1). Arkkitehtuuriset hajut eroavat koodihajuista abstraktiotasolla. Syitä arkkitehtuurisille hajuille on (de Andrade, Almeida and Crnkovic, N.d., s. 2): suunnittelumalli väärässä kontekstissa; design abstractions käyttö niin, että niillä on ei-haluttuja vaikutuksia; design abstractions käyttö väärällä rakeisuuden (granularity) tasolla.

Testattavuus (Garcia et al., 2009) yleisesti (Bertran, 2011).

(Mo et al., 2015) arkkitehtuuriset hajut "hotspotit" 5kpl + kaikista altin virheille oli (Mo et al., 2015, s. 57) Unstable Interface & Cross-Module Cycle.

Software Product Line Arch. smells -> (de Andrade, Almeida and Crnkovic, N.d.). Maintainability. Case: Notepad SPL. Komponenttimalli + arkkitehtuurin malli. **1** testability connector envy: Combining product construction capabilities and connector responsibilities represent a reduction in testability because application functionalities and interaction functionalities cannot be seperately tested. **2**: Ambiguous interfaces 1kpl. **3**: Feature Concentration -> paljon toiminnallisuutta yhteen komponenttiin sidottuna -> vaikeuttaa maintainability.

Viisi hajua (4 + 1 for spl) (Garcia et al., N.d.): Connector envy (komponenteilla liikaa toiminnallisuutte suhteissa niiden konnektoreihin), Scattered Parasitic Functionality (korkean tason asia jaettu usealle eri komponentille -> väh. yhdellä komponentilla useita vastuita, komponentit tässä riippuvaisia toisistaan), Ambiguous Interfaces (komponentti tarjoaa vain yhden geneerisen sisääntulopisteen [entry-point]), Extraneous Adjacent Connector (kaksi eri tyypin konnektoria yhdistää kaksi komponenttia toisiinsa), Feature Concentration (SPL).

6 Pohdinta

7 Yhteenveto

"puoli sivua"

Yleensä hieman johdantoa lyhyempi.

Muistuttaa mieleen tutkimuskysymyksen, mainitsee tärkeimmät tulokset ja niiden perusteet. Keskittyy impaktiin ja esimerkiksi suosituksiin. Ei vain summeeraa luku kerrallaan aikaisempaa tekstiä.

Lähteet

Baudry, B., Y. Le Traon and G. Sunye. N.d. Testability analysis of a UML class diagram. In *Eighth IEEE Symposium on Software Metrics, 2002. Proceedings.* pp. 54–63.

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

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

Bertran, Isela Macia. 2011. Detecting architecturally-relevant code smells in evolving software systems. In *Proceedings of the 33rd International Conference on Software Engineering.* ACM pp. 1090–1093.
URL: *http://dl.acm.org/citation.cfm?id=1986003*

Brøndum, John and Liming Zhu. 2010. Towards an Architectural Viewpoint for Systems of Software Intensive Systems. In *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge.* SHARK '10 New York, NY, USA: ACM pp. 60–63.
URL: *http://doi.acm.org/10.1145/1833335.1833344*

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

Coelho, Roberta, Uirá Kulesza and Arndt Von Staa. 2005. Improving architecture testability with patterns. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* ACM pp. 114–115.
URL: *http://dl.acm.org/citation.cfm?id=1094890*

de Andrade, Hugo Sica, Eduardo Almeida and Ivica Crnkovic. N.d. Architectural bad smells in software product lines: an exploratory study. ACM Press pp. 1–6.
URL: *http://dl.acm.org/citation.cfm?doid=2578128.2578237*

Dubey, Sanjay Kumar and Ajay Rana. 2011. "Assessment of Maintainability Metrics for Object-oriented Software System." *SIGSOFT Softw. Eng. Notes* 36(5):1–7.
URL: *http://doi.acm.org/10.1145/2020976.2020983*

Eickelmann, Nancy S. and Debra J. Richardson. 1996. What Makes One Software Architecture More Testable Than Another? In *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 New York, NY, USA: ACM pp. 65–67.
URL: *http://doi.acm.org/10.1145/243327.243602*

Freedman, Roy S. 1991. "Testability of Software Components." *IEEE Trans. Softw. Eng.* 17(6):553–564.
URL: *http://dx.doi.org/10.1109/32.87281*

Galster, Matthias. 2011. Dependencies, Traceability and Consistency in Software Architecture: Towards a View-based Perspective. In *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*. ECSA '11 New York, NY, USA: ACM pp. 1:1–1:4.
URL: *http://doi.acm.org/10.1145/2031759.2031761*

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

Garcia, Joshua, Daniel Popescu, George Edwards and Nenad Medvidovic. N.d. Toward a catalogue of architectural bad smells. In *Architectures for adaptive software systems*. Springer pp. 146–162.

Garlan, David. 2000. Software Architecture: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00 New York, NY, USA: ACM pp. 91–101.
URL: *http://doi.acm.org/10.1145/336512.336537*

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

IEEE. 2000. "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems." pp. i–23.

ISO. 2011. “ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.”.

May, Nicholas. 2005. A survey of software architecture viewpoint models. In *Proceedings of the Sixth Australasian Workshop on Software and System Architectures*. Citeseer pp. 13–24.

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

Ran, Alexander. 1998. Architectural Structures and Views. In *Proceedings of the Third International Workshop on Software Architecture*. ISAW ’98 New York, NY, USA: ACM pp. 117–120.
URL: *http://doi.acm.org/10.1145/288408.288438*

Razavizadeh, A., H. Verjus, S. Cimpan and S. Ducasse. 2009. Multiple viewpoints architecture extraction. In *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*. pp. 329–332.

Rozanski, Nick and 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? In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*. SAICSIT ’12 ACM pp. 363–373.

Voas, Jeffrey M. and Keith W. Miller. 1995. “Software Testability: The New Verification.” *IEEE Softw.* 12(3):17–28.
URL: *http://dx.doi.org/10.1109/52.382180*

Voas, Jeffrey M. and Keith W. Miller. N.d. Improving the software development process using testability research. In *Software Reliability Engineering, 1992. Proceedings., Third International Symposium on*. IEEE pp. 114–121.