Santeri Mikkonen

Testausarkkitehtuuri ohjelmistotestauksessa

Tietotekniikan pro gradu -tutkielma

21. toukokuuta 2024

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Santeri Mikkonen

Yhteystiedot: santeri.m.mikkonen@student.jyu.fi

Ohjaaja: Tommi Mikkonen

Työn nimi: Testausarkkitehtuuri ohjelmistotestauksessa

Title in English: Test architecture in software testing

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 52+0

Tiivistelmä: Ohjelmistojen ja järjestelmien kasvaessa monimutkaisemmiksi ohjelmistotestauksen merkitys ohjelmistokehityksessä korostuu. Testauksen on oltava hallittua ja luotettavaa, ja testausarkkitehtuuri pyrkii vastaamaan tähän määrittelemällä testausjärjestelmän, testausympäristöt, testausautomaation sekä arkkitehtoniset kuvaukset ja näkökulmat.

Tutkielmassa suoritettiin systemaattinen kirjallisuuskatsaus, jossa tarkasteltiin, mitä tieteellistä kirjallisuutta testausarkkitehtuurista on olemassa ja miten testausarkkitehtuuri on määritelty ja voidaan määritellä tieteellisen kirjallisuuden perusteella. Tutkielmaan valittiin yhteensä 22 tutkimusta. Tutkielman tuloksena testausarkkitehtuurikirjallisuus tunnistettiin ja testausarkkitehtuurin käsitteet ja osat määriteltiin.

Avainsanat: testausarkkitehtuuri, ohjelmistotestaus, ohjelmistoarkkitehtuuri, systemaattinen kirjallisuuskatsaus

Abstract: As softwares and systems become more complex, the importance of software testing in software development is emphasized. Testing has to be controlled and reliable and test architecture seeks to address this by defining test system, test environments, test automation, architectural descriptions and views.

Systematic literature review was conducted on this study to review what scientific literature exists of test architecture and how test architecture has and can be defined based on scien-

tific literature. Total of 22 studies were selected for the study. As a result, test architecture literature was identified and test architecture and of its parts were defined.

Keywords: test architecture, software testing, software architecture, systematic literature review

Kuviot

Taulukot

Sisällys

1 Johdanto

Ohjelmistojen ja tietojärjestelmien monimutkaistuessa ja niiden koon kasvaessa ohjelmistotestauksen merkitys on noussut yhä kriittisemmäksi. Ohjelmistotestauksen tarkoituksena on varmistaa ohjelmiston toimivuus, luotettavuus ja laatu. Tehokkaan ja onnistuneen testauksen takaamiseksi testauksessa voidaan hyödyntää testausarkkitehtuuria ohjaamassa testausta ja siihen liittyviä toimintoja.

Tässä tutkielmassa määritellään testausarkkitehtuurin käsite ja sen osat systemaattisen kirjallisuuskatsauksen avulla sekä käyttäen apuna ohjelmistoarkkitehtuuria ja sen käsitettä. Tutkielmassa pyritään myös kartoittamaan testausarkkitehtuurin rooleja käsitellen testausarkkitehdin vastuita ja tehtäviä.

Tutkielman aihe on tärkeä, koska testausarkkitehtuuri ei ole kovin vakiintunut käsite alalla, eikä sille ole määritelty vielä universaaleja standardeja. Testausarkkitehtuurille löytyy vain puutteellisia määritelmiä ja määritelmistä ei ole selkeää yhteisymmärrystä (León-Carrillo 2023). Tieteellisessä kirjallisuudessa ja yhteisössä on ilmaistu tarve jatkotutkimuksille (Jon D. Hagar 2017) testausarkkitehtuurin käsitteen määrittämiseksi.

Testausarkkitehtuuri ohjaa ohjelmistotestausta määrittelemällä testausjärjestelmän, testausympäristöt, testausautomaation sekä arkkitehtoniset kuvaukset ja näkökulmat. Testausarkkitehtuurin kuvaukset auttavat sidosryhmien välistä kommunikointia (Garlan 2008). Tässä tutkielmassa testausarkkitehtuuria tarkastellaan ohjelmistotestauksen näkökulmasta.

Luvussa 2 käsitellään ohjelmistotestausta, sen merkitystä ohjelmistokehityksessä, Meyerin (2008) testaukseen liittyviä periaatteita ja testauksen tasoja. Luvussa 3 käsitellään ohjelmistoarkkitehtuuria esittämällä sen määritelmä, rooli ohjelmistokehityksessä, arkkitehtuurikuvauksen käsitteen, Kruchtenin (1995) 4+1-malli ja Brownin (2024) C4-malli. Luvussa 4 esitetään tämän tutkielman tutkimusmenetelmä – systemaattinen kirjallisuuskatsaus – ja sen vaiheet Kitchenham ym. (2007) ohjeiden mukaan sisältäen katsauksen suunnittelun ja suorittamisen. Luvussa 4 esitetään myös tutkimusaiheen motivointi kuvaamalla arkkitehtuurin mahdollisia hyötyjä ohjelmistotestauksessa. Luvussa 5 esitetään systemaattisen kirjallisuuskatsauksen tulokset ja laadullisesti syntetisoidaan tulokset määritellen testausarkkitehtuurin

käsite ja sen osat. Luvussa 6 pohditaan tutkielman validiteettia ja reliabiliteettia, mahdollisia rajoitteita sekä jatkotutkimuskohteita. Tutkielman lopussa luvussa 7 esitetään tutkielman yhteenveto.

2 Ohjelmistotestaus

Tässä luvussa tarkastellaan ohjelmistotestausta ja sen tehtäviä. Lisäksi luvussa esitetään ohjelmistotestauksessa käytettävät periaatteet. Luvun lopuksi esitetään ohjelmistotestauksen neljä tasoa: yksikkötestaus, integraatiotestaus, järjestelmätestaus ja hyväksymistestaus.

2.1 Tehtävät ja merkitys

Ohjelmistotestauksella tarkoitetaan prosesseja ja toimia, joilla tarkaillaan toimiiko ohjelmisto suunnitellulla tavalla verraten eroja ohjelmiston käyttäytymisen ja odotettujen tulosten välillä sekä tunnistetaan ja korjataan mahdollisia ohjelmiston virheitä (Bertolino 2007), (Anand ja Uddin 2019). Tämän lisäksi ohjelmistotestauksen tavoitteena on varmistaa, että ohjelmisto toimii hallitusti ja luotettavasti.

Ohjelmistotestauksen päämääränä on myös validoida ja verifioida ohjelmisto. Validoinnilla varmistetaan, että ohjelmisto vastaa sidosryhmien tarpeita. Validointi vastaa kysymykseen ”kehitämmekö oikean tuotteen?”. Verifioinnin voidaan määritellä olevan prosessi, jossa tarkastellaan täyttääkö ohjelmisto sille asetetut vaatimukset varmistaen, että ohjelmisto toteuttaa tietyn toiminnon oikein. Verifiointi vastaa kysymykseen ”kehitämmekö tuotteen oikein?” (Reddy ja Prasad 2016).

Ohjelmistotestauksen merkitys on korostunut yhä enemmän, kun ohjelmistoista ja tietojärjestelmistä on tullut entistä monimutkaisempia ja laajempia. Ohjelmistotestauksen kustannusten osuus ohjelmistokehityksen kokonaiskustannuksista arvioidaan olevan 30-60 prosentin luokkaa (Brar ja Kaur 2015). Ohjelmistotestaukseen investointi kannattaa, koska häiriöt tuotannossa näissä järjestelmissä voivat olla katastrofaalisia ja johtaa suuriin lisäkustannuksiin (Valle-Gómez ym. 2019). Ohjelmistotestauksen epäonnistumisen vaikutukset ovat vielä tuhoisampia kriittisissä järjestelmissä, kuten ydinvoimaloissa ja rautateiden ohjausinfrastruktuureissa, missä häiriöt voivat johtaa pahimmillaan ihmishenkien menetyksiin (Liu ja Mei 2014).

2.2 Periaatteet

Meyer (2008) määrittelee artikkelissaan seitsemän ohjelmistotestaukseen liittyvää periaatetta. Periaatteet tarjoavat ohjeet testauksen suorittamiseen ohjelmistokehityksen eri testausvaiheissa.

Ensimmäisen periaatteen mukaan ohjelmiston testaamisessa pitää pyrkiä löytämään virheitä (engl. ”To test a program is to try to make it fail.”). Tämä lähestymistapa erottaa testauksen virheenkorjauksesta ja selkeyttää testausta (Meyer 2008).

Toisen periaatteen mukaan testit eivät korvaa spesifikaatioita (engl. ”Tests are no substitute for specifications.”). Meyer (2008) tarkentaa vielä, että testit ovat instansseja, joista puuttuu abstraktio, jonka vain spesifikaatiot voivat tarjota. Testejä voidaan myös luoda automaattisesti spesifikaatioiden pohjalta, mutta spesifikaatioita ei voida luoda testien pohjalta ilman testaajan väliintuloa. Lisäksi ilman spesifikaatioita testit eivät kata kaikkia kriittisiä tapauksia (Meyer 2008).

Kolmas periaate liittyy regressiotestaukseen. Sen mukaan jokaisesta ohjelmiston suorittamisessa havaitusta virheestä pitää luoda testitapaus osaksi testisarjaa (engl. ”Any failed execution must yield a test case, to remain a permanent part of the project’s test suite.”) (Meyer 2008).

Neljäs periaate ohjeistaa, että testien epäonnistumisen tai onnistumisen määrittäminen pitää olla automatisoitu (engl. ”Determining success or failure of tests must be an automatic process.”). Määrittämisestä vastaa oraakkeli. Oraakkeli vertaa odotettua tulosta havainnon tulokseen (Meyer 2008).

Viidennen periaatteen mukaan tehokkaan testausprosessin pitää sisältää sekä manuaalisesti että automaattisesti luotuja testitapauksia (engl. ”An effective testing process must include both manually and automatically produced test cases.”). Testauksen lähestymistavat täydentävät toisiaan. Manuaalisilla testeillä voidaan käsitellä helposti ohjelmiston ongelma-alueita, joihin automaattisten testien luominen on hankalaa. Automaattiset testit puolestaan soveltuvat hyvin rajatilanteiden kattamiseen (Meyer 2008).

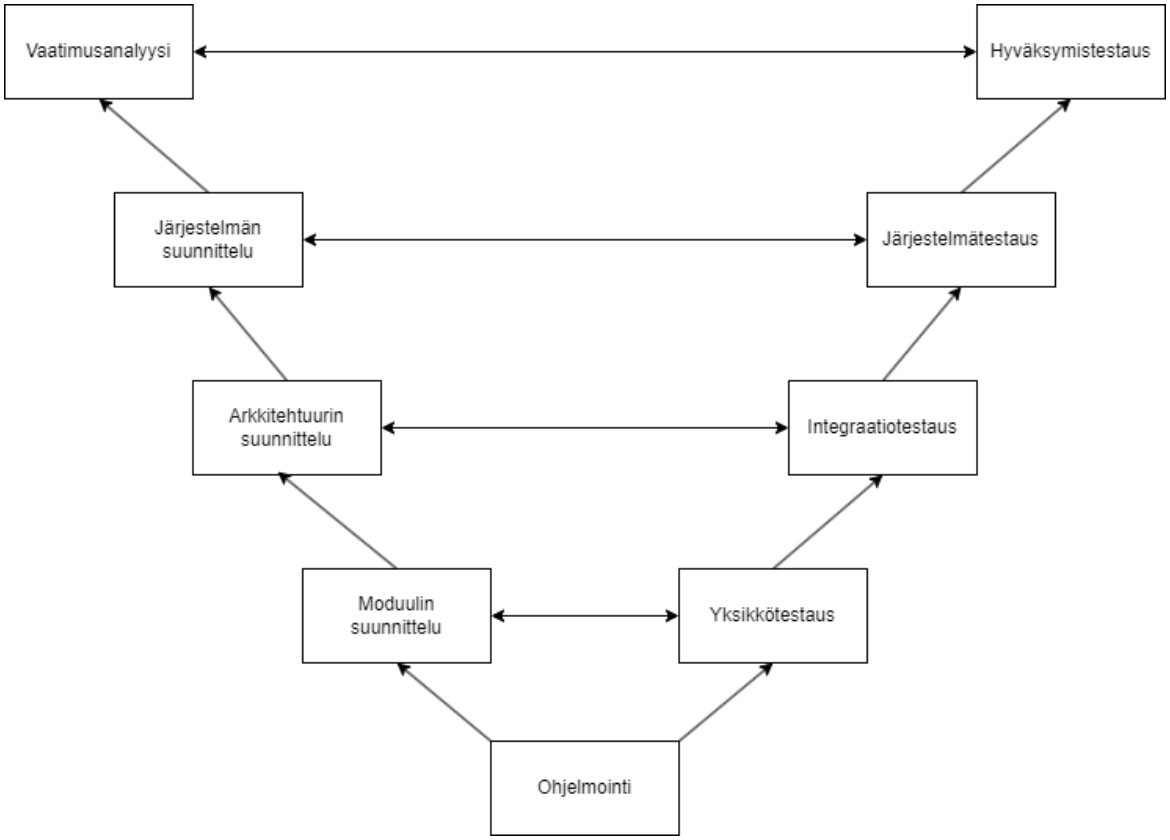
Kuudes periaate ohjeistaa arvioimaan testausstrategian objektiivisesti käyttäen eksplisiitti-

siä kriteerejä (engl. ”Evaluate any testing strategy, however attractive in principle, through objective assessment using explicit criteria in a reproducible testing process.”) (Meyer 2008).

Seitsemännen ja viimeisen Meyerin ohjelmistotestauksen periaatteen mukaan testausstrategian tärkein ominaisuus on löydettyjen virheiden suhde aikaan nähden (engl. ”A testing strategy’s most important property is the number of faults it uncovers as a function of time.”) (Meyer 2008). Meyer (2008) mainitsee tämän funktion auttavan testauksen lopettamisen määrittelyssä.

2.3 Tasot

Ohjelmistotestauksen tasot kuvaavat testausprosessin eri vaiheita ohjelmistokehityksessä. Ohjelmistotestauksen tasoja on neljä: yksikkötestaus, integraatiotestaus, järjestelmätestaus ja hyväksyntätestaus. Testauksen tasot on kuvattu V-mallissa, testauksen oikeassa haarassa. V-malli nähdään vesiputousmallin laajennuksena ja sen tarkoituksena on parantaa ohjelmistokehityksen tehokkuutta ja vaikuttavuutta sekä korostaa ohjelmistokehityksen vaiheen ja sitä vastaavan testausprosessin vaiheen välistä suhdetta. Testauksen kustannukset kasvavat ja virheiden korjaaminen hankaloituu mitä ylempänä V-mallissa ollaan (Mathur ja Malik 2010). V-malli on esitetty alla kuviossa 1.



Kuvio 1. V-malli

2.3.1 Yksikkötestaus

Whittaker (2000) määrittelee yksikkötestauksen testaavan yksittäisiä ohjelmiston moduuleja tai moduulien joukkoja eristyksessä muusta järjestelmästä. Runeson (2006) tarkentaa vielä edellistä määritelmää kuvaamalla yksikkötestauksen testaavan pienintä erillistä moduulia järjestelmässä. Runeson (2006) mainitsee samalla myös, että alalla on eriäviä mielipiteitä yksikkötestauksen käsitteen tarkennuksen tarpeellisuudesta. Yksikkötestien kirjoittamisesta vastaa usein itse ohjelmakoodin ohjelmoija (Brar ja Kaur 2015). Yksikkötestaus on kuvattu V-mallin alaosassa ja se on testauksen taso, joka tulee suorittaa ohjelmistokehityksen ensimmäisessä vaiheessa.

Yksikkötestauksen tavoitteena on varmistaa, että jokainen järjestelmän osa toimii odotetulla tavalla eristyksessä muusta järjestelmästä. Yksikkötestit parantavat Klammer ja Kern (2015) mukaan ohjelmiston testattavuutta ja ilman yksikkötestejä testien lisääminen jälkikäteen voi aiheuttaa suuria lisäkustannuksia, kun ohjelmistokoodia joudutaan refaktoroimaan samalla. Yksikkötestien puuttumista voidaan pahimmassa tapauksessa paikata lisäämällä manuaalista järjestelmätestausta, joka on kallista (Klammer ja Kern 2015).

Yksi lähestymistavoista yksikkötestien kirjoittamiseen on Test Driven Development (TDD). Siinä yksikkötestit kirjoitetaan ennen ohjelmakoodin kirjoittamista, jonka jälkeen testit ajetaan epäonnistuneesti. Vasta tämän jälkeen ohjelmakoodin tehdään tarpeelliset muokkaukset ja refaktorointi niin, että seuraavalla suorituskerralla testit menevät läpi (Dookhun ja Nagowah 2019). TDD on ketterä lähestymistapa kirjoittaa yksikkötestejä, mutta sen tutkitut hyödyt ohjelmistotestauksessa ovat Karac ja Turhan (2018) mukaan ristiriitaisia.

2.3.2 Integraatiotestaus

Integraatiotestauksessa testataan ohjelmiston erillisten yksikkötestattujen moduulien vuorovaikutusta toistensa kanssa osana järjestelmää. Testauksessa otetaan huomioon ulkoiset moduulit ja niiden vaikutukset (Brar ja Kaur 2015). Testauksen kohteena voi olla esimerkiksi toiminto, jonka monta eri ohjelmiston tai järjestelmän moduulia toteuttaa. Integraatiotestaus on V-mallissa toisiksi alin testausprosessin vaihe ja se on yhteydessä ohjelmiston arkkitehtuurin suunnitteluun.

Integraatiotestien tavoitteena on löytää virheitä moduulien välisissä vuorovaikutustilanteissa, joita ei löytyisi vain moduuleita erikseen testaamalla. Tämän lisäksi integraatiotestauksella taataan, että ohjelmiston osat toimivat odotetulla tavalla niitä integroitaessa (Holling ym. 2016), (Brar ja Kaur 2015).

Brar ja Kaur (2015) listaavat neljä integraatiotestauksen suoritustapaa: top down, bottom up, big bang ja sandwich. Top down -menetelmässä korkeamman tason moduulit testataan ensin, jonka jälkeen liikutaan alaspäin testaamaan alemman tason moduuleita. Jos alemman tason moduulia ei ole olemassa, sen tilalle luodaan ”moduulitynkä” tarjoamaan halutun rajapinnan datavirta. Bottom up -menetelmässä etenemissuunta on nimensä mukaisesti päinvastainen. Big bang -menetelmällä ei ole sekvenssistä kulkua, vaan siinä moduulit testataan erikseen ja yhdistetään samalla kertaa yhteen, jonka jälkeen integraation tulos testataan. Lopuksi sandwich -menetelmässä yhdistetään bottom up ja top down -menetelmiä, jolloin testaaminen ja integroiminen suoritetaan molempiin suuntiin (Brar ja Kaur 2015).

2.3.3 Järjestelmätestaus

Järjestelmän toimivuus ja vaatimustenmukaisuus arvioidaan testaamalla järjestelmää kokonaisuutena järjestelmän testaussuunnitelman perusteella (Lukas ja Lukas 2014). Järjestelmätestauksen avulla voidaan tunnistaa ja korjata mahdolliset puutteet ja virheet järjestelmässä ennen hyväksyntätestausta. Järjestelmätestauksesta käytetään usein termiä end-to-end testing (E2E). Järjestelmätestaus on V-mallissa toisiksi ylin testausprosessin vaihe ja se on yhteydessä järjestelmän suunnitteluun.

2.3.4 Hyväksymistestaus

“ISTQB Glossary” (2024) määrittää hyväksymistestauksen testitasoksi, jonka tarkoituksena on määrittää onko tarkasteltava järjestelmä hyväksyttävissä. Täten hyväksymistestauksessa todennetaan, että asiakkaalle kehitetty järjestelmä sisältää asiakkaan kanssa sovitut toiminnot ja toimii sovitulla tavalla (Miller ja Collins 2001). Asiakas ei kuitenkaan aina tiedä tarkalleen mitä hän tarvitsee järjestelmältä, jonka takia joudutaan turvautumaan sopimuksen sisältöön juristin kanssa (Brannigan 1985). Hyväksymistestaus on V-mallissa ylin testaus-

prosessin vaihe ja se on yhteydessä vaatimusanalyysiin.

3 Ohjelmistoarkkitehtuuri

Tässä luvussa käsitellään ohjelmistoarkkitehtuurin määritelmää, sen roolia ohjelmistokehityksessä ja arkkitehtuurikuvauksia. Lisäksi luvussa esitellään Kruchtenin (1995) 4+1-malli ohjelmistojen arkkitehtuurin kuvaamiseen.

3.1 Määritelmä

IEEE 1471 standardissa ohjelmistoarkkitehtuuri on määritelty olevan ”järjestelmän perusorganisaatio, joka sisältää sen osat, niiden suhteet toisiinsa ja ympäristöön sekä sen suunnittelua ja kehitystä ohjaavat periaatteet” (Maier, Emery ja Hilliard 2001). Määritelmän mukaan ohjelmistoarkkitehtuuri ei ole pelkästään järjestelmän komponenttien yleinen rakenne. Garlan (2008) huomauttaa, että vaikka ohjelmistoarkkitehtuurin käsitteelle löytyy useita eri määritelmiä, kaikille määritelmille yhteistä on käsitys, että järjestelmän arkkitehtuuri kuvaa sen rakennetta käyttäen yhtä tai useampaa näkymää. Garlan (2008) esittää ohjelmistoarkkitehtuurin toimivan siltana vaatimusten ja ohjelmakoodin välillä. Sen avulla järjestelmän kehittäjät pystyvät arvioimaan järjestelmän kykyä täyttää sille asetut vaatimukset.

Solms (2012) on tunnistanut analyysissään ohjelmistoarkkitehtuurin elementit

- peruskäsitteinä ja rajoituksina, joiden sisällä toimintoja tarjoavat sovelluskomponentit ovat määritelty,
- joukkona arkkitehtonisia komponentteja, jotka käsittelevät ja ratkaisevat teknisiä huolenaiheita,
- integraatiokanavina ympäristölle ja sisäisenä integraatioinfrastruktuurina arkkitehtonisille ja ohjelmiston komponenteille ja
- joukkona arkkitehtonisia strategioita, joiden avulla vastataan ohjelmistojärjestelmän laatuvaatimuksiin.

3.2 Roolit

Ohjelmistoarkkitehti voidaan määritellä Kruchten (2008) mukaan henkilöksi, joka tekee suunnitteluvalintoja, jotka he validoivat ja dokumentoivat erilaisissa arkkitehtuuriin liittyvissä artifakteissa. McBride (2004) kuvailevat ohjelmistoarkkitehtejä teknisesti päteviksi järjestelmätason ajattelijoiksi, jotka ovat mukana suunnitteluprosessissa toteuttamassa suunnittelut käytäntöön johtaen monenlaiset sidosryhmät menestykseen.

Ohjelmistoarkkitehdin rooli ohjelmistokehityksessä nähdään laajempana kuin pelkästään ohjelmiston arkkitehtuurin suunnittelijana ja kehittäjänä. Kruchten (2008) mukaan ohjelmistoarkkitehdillä on merkittävä rooli eri tiimien välisessä koordinoinnissa ja projektin suunnittelutoiminnassa. Lisäksi ohjelmistoarkkitehti on mukana teknisten riskien arvioinnissa ja niiden ratkaisujen suunnittelussa sekä järjestelmän arkkitehtonisen eheyden ylläpitämisessä (Kruchten 2008). Tästä syystä ohjelmistoarkkitehtien tehtäviä tai vastuualueita ei ole tarkasti määritelty. Tämä korostuu enemmän vielä ketterässä ohjelmistokehityksessä, missä arkkitehtuuri rakentuu ja muuttuu projektin edetessä (Premraj ym. 2011).

McBride (2004) on esittänyt kuusi periaatteellista ohjetta ohjelmistoarkkitehdeille. Ensimmäinen ohje neuvoo ohjelmistoarkkitehtejä lieventämään monimutkaisuutta strategisilla ja taktisilla keinoilla. Strategiset keinot pitävät sisällään projektinhallinnan, riskianalyysin, tehokkaan kommunikoinnin ja sidosryhmien koulutus asiaan liittyvistä teknologioista. Taktiset keinot sisältävät tehokkaan vaatimusten suunnittelun, järjestelmän jakamisen kerroksiin sekä rajapintojen määrittämisen (McBride 2004).

Toisen ohjeen mukaan ohjelmistoarkkitehdin pitää hallita toiminnallisia vaatimuksia. Ohjelmistokehityksen alussa ohjelmistoarkkitehdin pitää muodostaa ymmärrys aiheen ongelma-alueesta, jota hyödyntäen ohjelmistoarkkitehti pystyy määrittämään mikä on mahdollista huomioiden aikarajoitteet sekä käytettävissä olevat resurssit ja teknologiat (McBride 2004).

McBriden (2004) kolmas ohje neuvoo ohjelmistoarkkitehtejä kommunikoimaan tehokkaasti. Tehokasta kommunikointia tarvitaan, koska ohjelmistoarkkitehti on vuorovaikutuksessa eri sidosryhmien kanssa, joiden toimialueen tietämys ja ohjelmistokehitykseen liittyvä osaaminen vaihtelevat. Näkemykset ohjelmistoon liittyvistä ratkaisuista ohjelmistoarkkitehti voi kommunikoida sidosryhmille käyttäen apuna arkkitehtoonisia näkymiä tai abstraktioita (Mc-

Bride 2004).

Neljännen ohjeen mukaan ohjelmistoarkkitehdin tulee omaksua johtajuus. McBride (2004) kuvaa johtajuuden olevan yksi ohjelmistoarkkitehtien määrittelevistä ominaisuuksista. Johtajuuteen kuuluu järjestelmätason suunnittelun ja teknisen suunnan määrittäminen, yhteistyökyky eri tiimien kanssa ja kyky tehdä tehokkaita päätöksiä (McBride 2004).

Viidennessä ohjeessa ohjelmistoarkkitehtiä kehoitetaan kiinnittää huomiota ei-toiminnallisiin vaatimuksiin. Alussa kerättyjen vaatimusten lisäksi ohjelmistoarkkitehdin pitää tunnistaa muut ei-toiminnalliset vaatimukset ja jakaa niiden käsittely ohjelmistokehityksen eri tiimien välille (McBride 2004).

Kuudennessa ja viimeisessä ohjeessaan McBride (2004) ohjeistaa ohjelmistoarkkitehtejä omaamaan kattavan ”työkalupakin” ja osaamisen. Ohjelmistoarkkitehdin tulee osata käyttää malleja ja idiomeja sekä osata alan parhaat käytännöt. Ohjelmistoarkkitehdin pitää myös hyödyntää ohjelmistokehyksiä aina kun on tarkoituksenmukaista (McBride 2004).

3.3 Arkkitehtuurikuvaus

Arkkitehtuurikuvauksella tarkoitetaan dokumentteja, jotka kuvaavat järjestelmän arkkitehtuuria (Maier, Emery ja Hilliard 2001). Arkkitehtuurikuvaus koostuu näkymistä (engl. view), jotka ovat kuvauksia järjestelmän osista tietystä näkokohdasta, jotka täyttävät toiminnalliset ja ei-toiminnalliset vaatimukset (Omrani ja Ebrahimi 2013), (Maier, Emery ja Hilliard 2001). Näkökulma (engl. viewpoint) määrittelee yhtenäisen esitysmuodon koko järjestelmälle ja sen alijärjestelmille. Lisäksi se määrittelee tavoitteet, kohdeyleisön, näkymäluokan sisällön sekä huolenaiheet, joita luokan näkymät käsittelevät (Omrani ja Ebrahimi 2013). Huolenaiheet voivat liittyä järjestelmän toiminnallisuuteen, suorituskykyyn, tietoturvaan ja toteutettavuuteen. Kaikki järjestelmästä tunnistetut huolenaiheet pitää käsitellä arkkitehtuurikuvauksessa vähintään yhdellä näkymällä. Arkkitehtuurin kuvauksessa on annettava perustelut tärkeimpien arkkitehtonisten päätösten tekemiselle sisältäen mahdolliset kompromissit, muut vaihtoehdot, jotka jäivät valintojen ulkopuolelle ja analyysin päätöksen taustoista (Maier, Emery ja Hilliard 2001).

3.3.1 4+1-malli

4+1-malli on näkymämalli, jota käytetään ohjelmistointensiivisen järjestelmän arkkitehtuurin kuvauksen järjestämiseen (Kruchten 1995). Kruchten (1995) esittää ohjelmistoarkkitehtuurin kuvaamiselle viisi näkökulmaa: looginen näkymä (engl. logical view), kehitysnäkymä (engl. development view), prosessinäkymä (engl. process view), fyysinen näkymä (engl. physical view) ja skenaarionäkymä (engl. scenario view). 4+1-malli on Kruchtenin (1995) mukaan yleisluonteinen, eikä aseta rajoja notaatiolle tai työkaluille. Kuviossa 2 on esitetty Kruchtenin 4+1-malli ja sen viisi näkökulmaa.

Looginen näkymä kuvaa toiminnallisuuden jakautumista järjestelmän eri komponenttien kesken ja se edustaa järjestelmän toiminnallisia vaatimuksia. Toiminnot eritellään ja abstrahoidaan käyttäen olioluokkia ja olioita (Kruchten 1995), (Tang, Han ja Chen 2004).

Kehitysnäkymä kuvaa ohjelmistomoduulien organisointia ohjelmistokehitysympäristössä esittäen niiden suhteet. Kehitysnäkymää käytetään pohjana ohjelmistonhallinnassa sekä ohjelmiston uudelleenkäytön, siirrettävyyden ja turvallisuuden pohdinnassa (Kruchten 1995).

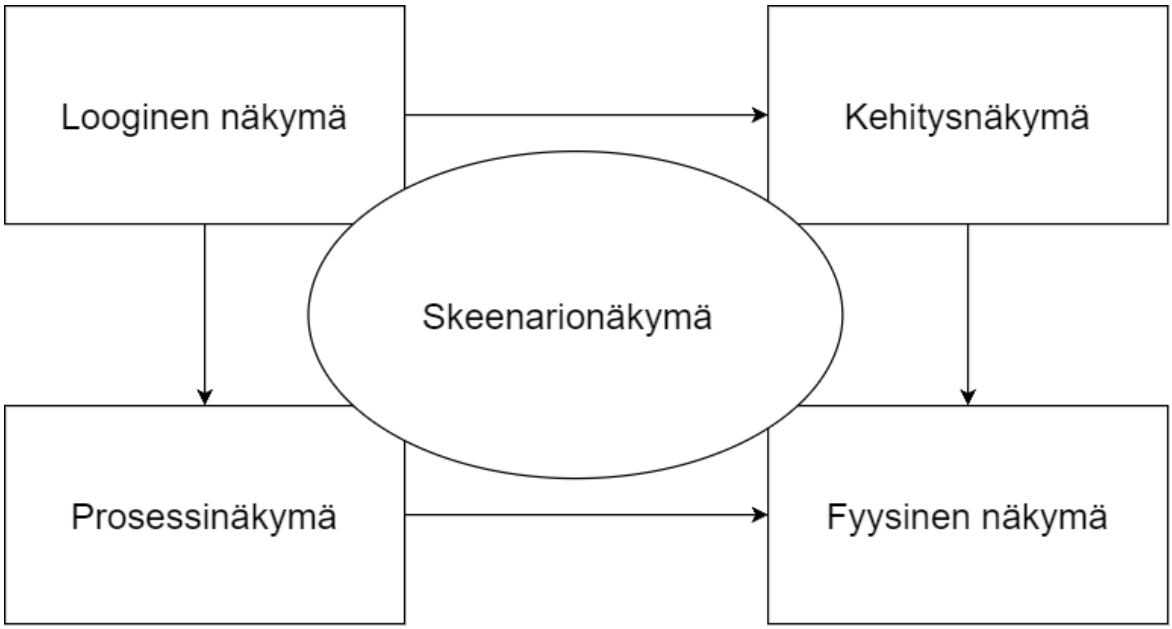
Prosessinäkymä helpottaa ohjelmiston jakamista joukkoon itsenäisiä prosesseja ja kuvaa näiden prosessien välisiä suhteita. Prosessinäkymä ottaa huomioon myös ohjelmiston eitoiminnalliset vaatimukset, kuten luotettavuus, siirrettävyys, skaalautuvuus ja saatavuus (Kruchten 1995), (Meng ym. 2010).

Fyysinen näkymä yhdistää ohjelmiston laitteistosolmuihin ja kuvaa laitteistojen välistä kommunikointia fyysisellä tasolla. Fyysinen näkymä voi kuvata eri laitteistojen konfiguraatioita eri ohjelmistokehityksen käyttötarkoituksiin (Kruchten 1995).

Viides näkymä – skenaarionäkymä – kuvaa ohjelmiston arkkitehtuuria käyttötapauksien kautta. Skenaarionäkymän rooli on löytää arkkitehtoonisia elementtejä arkkitehtuurin suunnittelun aikana ja validoida arkkitehtuuri, kun sen suunnittelu päättyy (Kruchten 1995), (Tang, Han ja Chen 2004).

Kruchtenin (1995) mukaan kaikkien arkkitehtuurikuvauksien ei tarvitse käyttää kaikkia viittä näkymää, vaan arkkitehtuurikuvaukseen pitää valita näkymät, jotka tarvitsevat kuvauksen tarkasteltavassa ohjelmistossa. Esimerkiksi fyysisen näkymän sisällyttäminen arkkitehtuuri-

kuvauksessa on turhaa, jos järjestelmässä on vain yksi laitteisto. Kruchten (1995) mainitsee skenaarionäkymän olevan kuitenkin hyödyllinen kaikissa tapauksissa.

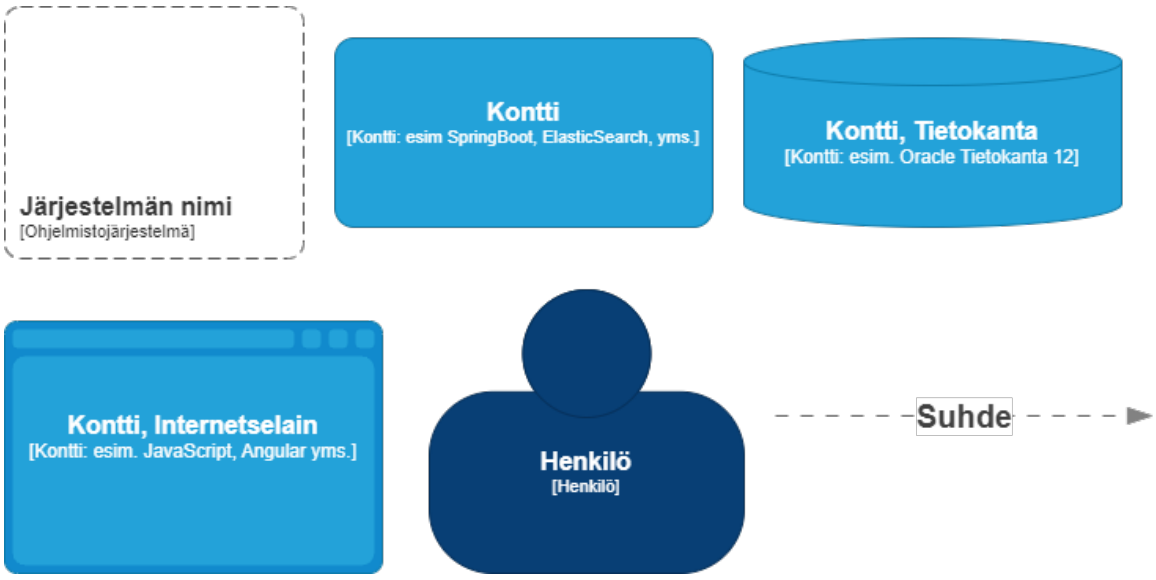


Kuvio 2. Kruchtenin (1995) 4+1-malli

3.3.2 C4-malli

C4-malli on Simon Brownin kehittämä ohjelmistoarkkitehtuurin kuvaustapa ja notaatiotekniikka (“The C4 model for visualising software architecture” 2024). Mallin tarkoituksena on Brownin (2024) mukaan auttaa ohjelmistokehittäjiä kuvaamaan ja ymmärtämään ohjelmistojärjestelmän toimintaa ja minimoimaan ohjelmistoarkkitehtuurin kuvauksen ja lähdekoodin välistä eroa. C4-malli määrittelee ohjelmistojärjestelmän neljän näkymän kautta, jotka ovat järjestelmän hierarkkisia tasoja (Vázquez-Ingelmo, García-Holgado ja García-Peñalvo 2020). C4-mallin muodostavien elementtien notaatio on kuvattu kuviossa 3.

C4-mallin ensimmäinen taso kuvaa ohjelmistojärjestelmän suhdetta muihin järjestelmiin ja käyttäjiin. Suhteita edustavat katkoviivoitetut nuolet, jotka sisältävät kuvauksen kahden osapuolen välisestä suhteesta. Toinen taso jakaa ohjelmistojärjestelmän kontteihin (engl. container). Tämä näkymä kuvaa tarvittavat kontit järjestelmän tarjoamien palveluiden toteuttamiseen (Vázquez-Ingelmo, García-Holgado ja García-Peñalvo 2020). Brown (2024) määrittää kontin olevan erikseen käyttöönotettava tai ohjelmistoyksikkö, joka suorittaa koodia tai tallentaa tietoja. Kolmannessa tasossa kuvataan yksittäistä konttia, mistä komponenteista se koostuu ja mitkä ovat komponenttien vastuut (Vázquez-Ingelmo, García-Holgado ja García-Peñalvo 2020). Neljäs taso kuvaa UML-kaavioiden, kuten luokkakaavioiden avulla, miten yksittäinen komponetti on toteutettu ohjelmakoodina (“The C4 model for visualising software architecture” 2024). Brown (2024) suosittelee kahden ensimmäisen tason käyttämistä kaikissa projekteissa, kun taas kaksi viimeistä tasoa ovat harkinnanvaraisia.



Kuvio 3. C4-mallin muodostavien elementtien notaatio

4 Systemaattinen kirjallisuuskatsaus

Tutkielmassa suoritettiin systemaattinen kirjallisuuskatsaus (engl. systematic literature review), jonka tavoitteena on vastata tutkimuksen tutkimuskysymyksiin. Tutkielma toteutettiin soveltaen Kitchenhamin ja Chartersin (2007) ohjeita systemaattisen kirjallisuuskatsauksen suorittamisesta ohjelmistotekniikan alalla.

4.1 Määritelmä

Kitchenham ym. (2007) määrittelevät systemaattisen kirjallisuuskatsauksen tutkimusmenetelmäksi, jossa tarkoituksena on tunnistaa, arvioida ja käsittää kaikki tiettyyn tutkimusaiheeseen tai -kysymyksiin liittyvät tutkimukset. Tutkimusmenetelmän juuret ovat lääketieteen tutkimuksessa, missä sitä on käytetty tukemaan empiiristen tutkimusten näyttöön perustuvaa lääketiedettä. Keskeisessä roolissa on tunnistaa ja raportoida tutkimukset, jotka eivät tue tutkimushypoteeseja sekä tutkimukset, jotka tukevat niitä. Systemaattinen kirjallisuuskatsaus luokitellaan sekundääriseksi tutkimukseksi, kun taas tutkimusmenetelmässä käytetyt tutkimukset luokitellaan primäärisiksi tutkimuksiksi (Kitchenham ym. 2007). Systemaattisen kirjallisuuskatsauksen perusperiaatteen kiteyttää hyvin Isaac Newtonin metafora ”If I have seen further it is by standing on the shoulders of giants.” (Scotchmer 1991).

Systemaattiselle kirjallisuuskatsaukselle samankaltaisia tutkimusmenetelmiä ovat systemaattinen kirjallisuuskartoitus (engl. systematic mapping study) ja tertiäärinen katsaus (engl. tertiary review). Systemaattisen kirjallisuuskartoituksen tavoitteena on tarjota laaja yleiskuva tutkimusalueesta ja tunnistaa jatkotutkimuksen alueita systemaattiselle kirjallisuuskatsaukselle tai primääritutkimuksille. Systemaattisessa kirjallisuuskartoituksessa hakeminen ei ole yhtä tarkennettu tai rajattu kuten systemaattisessa kirjallisuuskatsauksessa, jonka takia systemaattisessa kirjallisuuskartoituksessa tulosten määrä on suurempi ja tuloksissa keskitytään laajaan kattavuuteen kohdennettujen tuloksien sijaan. Toinen keskeinen ero näiden tutkimusmenetelmien välillä koskee tiedon keräämisen vaihetta: systemaattisessa kirjallisuuskartoituksessa tiedonkeruuprosessi on laajempi kuin systemaattisessa kirjallisuuskatsauksessa ja keskeisessä osassa sitä on tiedon luokittelu avainsanojen kautta, jotta tutkimukset ovat tun-

nistettavissa jatkokäyttöä varten (Kitchenham ym. 2007).

Tertiäärinen katsaus on systemaattinen kirjallisuuskatsaus systemaattisista kirjallisuuskatsauksista, jossa pyritään vastaamaan laajempiin tutkimuskysymyksiin. Tertiäärinen katsaus voidaan suorittaa, jos tutkimuskohteesta on jo olemassa systemaattisia kirjallisuuskatsauksia (Kitchenham ym. 2007).

Kitchenham ym. (2007) tiivistävät systemaattisen kirjallisuuskatsauksen vaiheet kolmeen päävaiheeseen: katsauksen suunnitteluun (engl. planning the review), katsauksen suorittamiseen (engl. conducting the review) ja katsauksen raportointiin (engl. reporting the review). Nämä vaiheet ja niiden suorittaminen tutkimuksessa on käsitelty seuraavissa alaluvuissa Kitchenham ym. (2007) ohjeiden mukaisessa järjestyksessä.

4.2 Motivointi

Arkitehtuurin hyödyntämisestä ohjelmistotestauksessa ei ole tarpeeksi tutkimustietoa luotettavaan hyötyjen arviointiin tutkimustulosten pohjalta. J. Hagar (2021) tutki testausarkkitehtuurin hyödyntämistä kaiuttimia valmistavassa yrityksessä, mutta todennettujen hyötyjen raportointi jäi puutteelliseksi immateriaalioikeuksien takia. J. Hagar (2021) mainitsee kyseisen yrityksen laajentaneen testaustiimiään ja olevan yhä markkinoilla tutkimuksen jälkeen. Tutkimuksen keskeisin johtopäätös oli, että testausarkkitehtuuri tarjoaa hyötyjä myös pienemmille ohjelmistojärjestelmille. Lisää tutkimustietoa arkkitehtuurin hyödyntämisestä ohjelmistotestauksessa todennäköisesti saadaan, kun testausarkkitehtuurin käsite on vakiinnuttanut asemansa ohjelmistotestauksessa.

Mahdollisia arkkitehtuurin tarjoamia hyötyjä testauksessa voidaan kuitenkin kartoittaa tarkastelemalla ohjelmistoarkkitehtuuria. Ohjelmistoarkkitehtuurin tutkitut hyödyt pätevät pääosin myös testausarkkitehtuuriin, koska nykyiset testausarkkitehtuurin käytännöt perustuvat ohjelmistoarkkitehtuurin käytäntöihin ja standardeihin (Masuda ym. 2022).

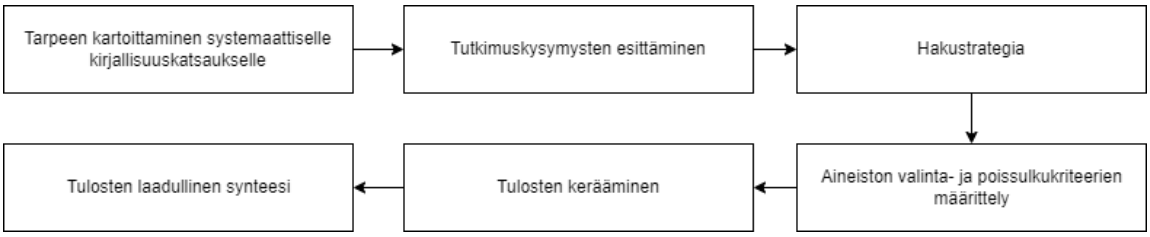
Garlan (2008) listaa seitsemän ohjelmistokehityksen näkökulmaa, joissa ohjelmistoarkkitehtuurilla voi olla merkittävä rooli. Nämä seitsemän näkökulmaa ovat ymmärtäminen, uudelleenkäyttö, rakentaminen, evoluutio, analysointi, hallinta ja kommunikaatio.

Arkkitehtuurin tarjoama abstraktitason kuvaus järjestelmästä auttaa ymmärtämään järjestelmän mallia ja siihen liittyviä rajoitteita sekä mahdollistaa järjestelmän kattavamman analysoinnin. Arkkitehtuurikuvauksesta voidaan todentaa toiminnallisten ja ei-toiminnallisten vaatimusten toteutumista järjestelmässä ja miten arkkitehtuuri ratkaisee eri sidosryhmien huolenaiheet. Tämä toimii työkaluna päätöksenteossa sekä riskien ja kustannusten arvioinnissa että hallinnassa. Arkkitehtuurikuvausta voidaan käyttää myös välineenä eri sidosryhmien välisessä kommunikoinnissa ja päätöksenteossa (Garlan 2008).

Arkkitehtuurikuvaus auttaa ohjelmistokehittäjiä ohjelmiston kehittämisessä määrittämällä keskeisimmät ohjelmiston komponentit ja niiden riippuvuudet. Kuvauksesta pystytään myös todentamaan komponenttien vastuut ja suhteet. Arkkitehtuurikuvaus helpottaa myös järjestelmään tehtäviä muutoksia erottamalla selvästi järjestelmän komponentit ja mekanismit, jotka mahdollistavat komponenttien vuorovaikutuksen. Arkkitehtoniset mallit tukevat isojen komponenttien ja kehyksien uudelleenkäyttöä, varsinkin alakohtaisina ratkaisuina. Komponenttien ja kehysten udelleenkäyttö vähentää ohjelmistokehityksen kustannuksia ja minimoi suunnitteluun liittyviä riskejä (Garlan 2008).

4.3 Katsauksen suunnittelu

Katsauksen suunnittelu aloitetaan kehittämällä katsausprotokolla, joka sisältää metodit, joita noudatetaan systemaattisen kirjallisuuskatsauksen toteuttamisessa. Protokollan määrittäminen vähentää tutkijan aiheeseen liittyvien odotusten ja ennakko-oletusten vaikutuksen riskiä (Kitchenham ym. 2007). Tutkimuksen protokolla on kuvattu kuviossa 4.



Kuvio 4. Katsausprotokollan rakenne

4.3.1 Tarve systemaattiselle kirjallisuuskatsaukselle

Kitchenham ym. (2007) määrittelevät katsauksen suunnittelun ensimmäiseksi tehtäväksi systemaattisen kirjauskatsauksen tarpeen tunnistamisen. Kitchenham ym. (2007) mukaan systemaattinen kirjallisuuskatsaus sopii tutkimuksiin, joissa tavoitteena on tiivistää olemassa olevaa tietoa tutkittavasta aiheesta mahdollisimman tarkasti ja ilman tutkijan omaa ennakkoasennetta. Tämän lisäksi systemaattisen kirjallisuuskatsauksen avulla voidaan tunnistaa mahdollisia aukkoja aiheen tutkimuksessa ja täten tarjota jatkotutkimuskohteita. Systemaattinen kirjallisuuskatsaus sopii tutkimusmenetelmäksi tähän tutkielmaan, koska tutkimuksen tavoitteena, ja samalla yhtenä tutkimuskysymyksenä, on selvittää, mitä tieteellistä kirjallisuutta on testausarkkitehtuurista ja miten testausarkkitehtuuri ja sen osat ovat määritelty tieteellisessä kirjallisuudessa. Tarkalla ja kattavalla hakustrategialla varmistetaan, että tutkimuksen kannalta keskeisimmät päälähteet sekä muut olennaiset tutkimukset löydetään. Tämän merkitys korostuu vielä tutkimusaiheeseen perehtyessä, koska testausarkkitehtuuri ei ole kovin tutkittu käsite eikä aiheesta ole tehty paljon tutkimuksia. Tämän lisäksi tieteellisessä yhteisössä on ilmaistu tarve jatkotutkimuksille (Jon D. Hagar 2017).

4.3.2 Tutkimuskysymykset

Seuraava tehtävä on määrittää tutkimuksen tutkimuskysymykset. Kitchenham ym. (2007) mukaan tutkimuskysymysten määrittäminen on tärkein osa systemaattista kirjallisuuskatsausta, sillä tutkimuskysymykset pitää ottaa huomioon hakuprosessissa, tiedon keräämisessä ja raportoinnissa. Tämän tutkielman tutkimuskysymykset ovat seuraavat:

1. Mitä tieteellisiä lähteitä löytyy testausarkkitehtuurista?
2. Miten testausarkkitehtuurin käsite on määritelty tieteellisessä kirjallisuudessa?
3. Miten testausarkkitehtuuri ja sen osat voidaan määritellä pohjautuen aiheen tieteelliseen kirjallisuuteen ja ohjelmistoarkkitehtuuriin?

4.4 Katsauksen suorittaminen

Katsauksen suorittamisessa haetaan ensin tutkimuskysymyksiin liittyvät tutkimukset hakustrategian avulla. Tämän jälkeen hakutuloksista valitaan valinta- ja poissulkukriteerien avulla

tutkimuskysymysten kannalta olennaisimmat lähteet katsausta varten.

4.4.1 Hakustrategia

Systemaattista kirjallisuuskatsausta varten tieteellisiä lähteitä haettiin seuraavista tietokannoista: IEEE Xplore, ACM Digital Library ja Google Scholar. Tietokantojen valintaan vaikuttivat tieteenalan tutkimusten määrä tietokannassa ja lukuoikeudet tietokannan tutkimuksiin. Hakusanat tunnistettiin tutkielman aiheen pääkäsitteistä ja tutkimuskysymyksistä, jonka jälkeen hakusanat yhdistettiin totuusarvomuuttujien (AND, OR, NOT) avulla hakulausekkeeksi. Hakulausekkeen tarkkuutta testattiin suorittamalla haku ja tarkastelemalla tulosten määrää ja relevanssia. Hakulausekkeeksi valikoitui ”test architecture” AND ”software” hakulausekkeen optimoinnin jälkeen. Haut eri tietokannoista suoritettiin 27.4.2024

IEEE Xploren tietokannassa haku suoritettiin käyttämällä ”Advanced Search” -hakutoimintoa. ”Test architecture” kirjoitettiin ensimmäiseen hakukenttään lainausmerkeissä, jotta käsite tunnistetaan kokonaisuudessan hakuterminä, toiseen hakukenttään kirjoitettiin ”software” ja kentän hakuoperaattoriksi valittiin ”AND”. Lopullinen hakulauseke ja tulokset ovat kuvattu taulukossa 1.

ACM Digital Libraryn tietokannassa haku suoritettiin kirjoittamalla hakulauseke suoraan tietokannan hakukenttään. Lopullinen hakulauseke ja tulokset ovat kuvattu taulukossa 2.

Google Scholar tietokannassa haku suoritettiin samankaltaisesti kuin ACM Digital Library -tietokannassa kirjoittamalla hakulauseke hakukenttään. Lopullinen hakulauseke ja tulokset on kuvattu taulukossa 3.

Taulukko 1. IEEE Xplore hakulauseke ja haun tulosten määrä

IEEE Xplore	
Hakulauseke	**Tulokset**
("All Metadata":"test architecture") AND ("All Metadata":software)	120

Taulukko 2. ACM Digital Library hakulauseke ja haun tulosten määrä

ACM Digital Library	
Hakulauseke	**Tulokset**
[All: "test architecture"] AND [All: software]	161

Taulukko 3. Google Scholar hakulauseke ja haun tulosten määrä

Google Scholar	
Hakulauseke	**Tulokset**
"test architecture"AND "software"	6200

4.4.2 Aineiston valinta- ja poissulkukriteerit

Systemaattisen kirjallisuuskatsauksen suorittamisessa käytettiin seuraavia aineiston valinta- ja poissulkukriteereitä:

Valintakriteerit:

- Tutkimus käsittelee testausarkkitehtuuria ohjelmistojen kontekstissa tai esittelee sen käsitteen
- Tutkimus on tieteellinen artikkeli tai konferenssijulkaisu
- Tutkimus on kirjoittu englanniksi
- Julkaisuvuosi on vuosien 1995 ja 2024 välillä
- Tutkimukseen on lukuoikeus Jyväskylän Yliopiston tunnuksilla tai ilman

Poissulkukriteerit:

- Tutkimus ei käsittele testausarkkitehtuuria ohjelmistojen kontekstissa tai esittele sen käsitettä
- Harmaan kirjallisuuden lähteet poissuljetaan
- Tutkimus on kirjoittu muulla kielellä kuin englanniksi
- Julkaisuvuosi on vuosien 1995–2024 ulkopuolella
- Tutkimukseen ei ole lukuoikeutta Jyväskylän Yliopiston tunnuksilla tai ilman

Valinta- ja poissulkukriteerit on määritelty niin, että tutkimuskysymysten kannalta olennaiset lähteet löydetään, kuitenkin rajaamatta lähteitä liikaa. Tutkimusten sisällön olennaisuus arvioitiin lukemalla tutkimusten otsikot ja abstraktit. Poissulkukriteereiden lisäksi Google Scholarin hakutulokset rajattiin ensimmäisiin 12 sivuun, eli 120 tulokseen. Rajaus tehtiin, jotta tulosten valinta- ja arviointiprosessi ei kasvaisi hallitsemattomaksi. Tämän lisäksi tarkasteltaessa kyseisen rajan jälkeisiä hakutuloksia, huomattiin, että tulokset eivät olleet tutkimuskysymysten näkökulmasta relevantteja. Valintaprosessin lopuksi valituista tuloksista poistettiin duplikaatit. Taulukossa 4 on kuvattu valittujen tulosten määrä tietokannoittain duplikaattien poistamisen jälkeen.

Taulukko 4. Valittujen tulosten määrä yhteensä ja tietokannoittain

Valittujen tulosten määrä tietokannoittain		
Tietokanta	**Hakutuloksia**	**Valittuja**
IEEE Xplore	120	16
ACM Digital Library	161	2
Google Scholar	120	4
Yhteensä	**401**	**22**

5 Tulokset

Tässä luvussa esitetään systemaattisen kirjallisuuskatsauksen tulokset ja kuvataan tulosten ominaisuuksia. Luvussa myös laadullisesti syntetisoidaan systemaattisen kirjallisuuskatsauksen tulokset määritellen testausarkkitehtuurin käsite ja sen osat. Tutkimuksia katsaukseen valittiin 22 kappaletta.

5.1 Valitut tutkimukset

Taulukoissa 5, 6, 7 ja 8 on esitelty valitut 22 tutkimusta ja eritelty niiden tiedot kuten tutkijat ja julkaisuvuosi, otsikko, missä tietokannassa tutkimus on julkaistu sekä tutkimuksen aihe.

Taulukko 5. Valittujen tutkimusten tiedot (1/4)

Tutkijat ja julkaisuvuosi	**Otsikko**	**Tietokanta**	**Aihe**
(Cusick 1999)	Deriving Software Test Environments From Architecture Styles	Google Scholar	Ohjelmistotestausympäristöjen johtaminen ohjelmistoarkkitehtuurityyleistä
(Hagar ja Wendland 2023)	Defining Software Test Architectures with the UML Testing Profile	IEEE Xplore	Testausarkkitehtuurin määrittäminen UML-testausprofiilin avulla
(J. Hagar 2021)	Multi-company Consumer Product Software Test Architecture Industry Experience Report	IEEE Xplore	Tapaustutkimus testausarkkitehtuurin käyttöönotosta
(Jon D Hagar 2018)	Software Test Architectures and Advanced Support Environments for IoT	IEEE Xplore	IoT testausarkkitehtuuri

Taulukko 6. Valittujen tutkimusten tiedot (2/4)

Tutkijat ja julkaisuvuosi	**Otsikko**	**Tietokanta**	**Aihe**
(Jon D Hagar 2022)	Software Architecture Elements Applied to Software Test: View, Viewpoints and Containers	IEEE Xplore	Testausarkkitehtuurin mallinnus
(Jon D. Hagar 2017)	Defining the Phrase "Software Test Architecture"Emerging Idea	IEEE Xplore	Onko testausarkkitehtuuri universaalisti määritelty ja käytetty käsite
(Jon ja Hagar 2020)	Identifying Software Test Architect Skills and Knowledge	IEEE Xplore	Testausarkkitehtuurin taitojen ja tietämyksen tunnistaminen
(Lahami ym. 2012)	Using Knapsack Problem Model to Design a Resource Aware Test Architecture for Adaptable and Distributed Systems	Google Scholar	Resurssitietoisen testausarkkitehtuurin suunnitteleminen mukautuville ja hajautetuille järjestelmille
(Lee 2009)	Double layered SOA test architecture based on BPA - simulation events	ACM Digital Library	SOA testausarkkitehtuuri
(Lee ja Kang 2014)	Towards Test Architecture Based Software Product Line Testing	IEEE Xplore	Testausarkkitehtuurin käyttöönotto ohjelmistotuotelinjan testauksessa

Taulukko 7. Valittujen tutkimusten tiedot (3/4)

Tutkijat ja julkaisuvuosi	**Otsikko**	**Tietokanta**	**Aihe**
(León-Carrillo 2023)	Elements for a Test(-ware) Architecture Language	IEEE Xplore	Testausarkkitehtuurin käsite muodollisen testauksen näkökulmasta katsottuna
(Masuda ym. 2022)	Software Test Architecture Definition by Analogy with Software Architecture	IEEE Xplore	Testausarkkitehtuurin käsitteen määrittäminen ohjelmistoarkkitehtuurin avulla
(Masuda, Nishi ja Suzuki 2020)	Complex Software Testing Analysis using International Standards	IEEE Xplore	Monimutkaisten ohjelmistojen testauksen analysointi
(Nidagundi ja Lukjanska 2016)	Introduction to adoption of lean canvas in software test architecture design	Google Scholar	Lean canvasin käyttö testausarkkitehtuurin suunnittelussa
(Nishi 2012)	Viewpoint-based Test Architecture Design	IEEE Xplore	Notaation esittäminen testausarkkitehtuurin suunnitteluun
(Nishi 2015)	Design principles in test suite architecture	IEEE Xplore	Testausarkkitehtuurin suunnittelu
(Nishi 2016)	Difference in Quality of Test Architecture between Service Providers and Subcontractors	IEEE Xplore	Erot testausarkkitehtuurin laadussa palveluntarjoajien ja alihankkijoiden välillä
(Nishi ja Shibasaki 2021)	Boosted Exploratory Test Architecture: Coaching Test Engineers with Word Similarity	IEEE Xplore	Tehostettu tutkiva testausarkkitehtuuri

Taulukko 8. Valittujen tutkimusten tiedot (4/4)

Tutkijat ja julkaisuvuosi	**Otsikko**	**Tietokanta**	**Aihe**
(Nishi ym. 2018)	A Test Architecture for Machine Learning Product	IEEE Xplore	Testausarkkitehtuuri koneoppimistuotteelle
(Nishi, Katayama ja Yoshizawa 2013)	Combinatorial Test Architecture Design Using Viewpoint Diagram	IEEE Xplore	Testausparametrien ja -yhdistelmien vähentäminen testausarkkitehtuurin suunnittelussa
(Paulisch ja Zimmerer 2016)	Collaboration of Software Architect and Test Architect Helps to Systematically Bridge Product Lifecycle Gap	ACM Digital Library	Ohjelmistoarkkitehdin ja testausarkkitehdin yhteistyö
(Rodrigues ym. 2005)	Towards an Integration Test Architecture for Open MAS	Google Scholar	Integraatiotestausarkkitehtuuri avoimissa moniagenttijärjestelmissä

5.2 Tutkimusaiheet

Valittujen tutkimusten tutkimusaiheet voidaan jakaa karkeasti neljään luokkaan: testausarkkitehtuurin käsitteen määrittelyyn, testausarkkitehtuurin käyttöönoton tutkimiseen, testausarkkitehtuurin suunnitteluun spesifioitua käyttökohdetta varten ja testausarkkitehdin roolin kuvaamiseen. Eniten tutkimuksia löytyi testausarkkitehtuurin käsitteen määrittelyyn ja testausarkkitehtuurin suunnitteluun spesifioituun käyttökohteeseen liittyen.

Keskeisimmät tutkijat testausarkkitehtuurin käsitteen määrittelyssä ovat Yasuharu Nishi ja Jon Hagar. Nishi (2012) julkaisi aluksi testausarkkitehtuuria ja sen notaatiota käsittelevän tutkimuksen, jonka jälkeen hän on julkaissut yksin tai yhdessä tutkijakollegojensa kanssa yhteensä 7 tutkimusta, pääosin laajentaen testausarkkitehtuurin käsitteeseen ja soveltamiseen liittyvää aiempaa tutkimusta. Jon Hagar on myös täydentänyt aiempia testausarkkiteh-

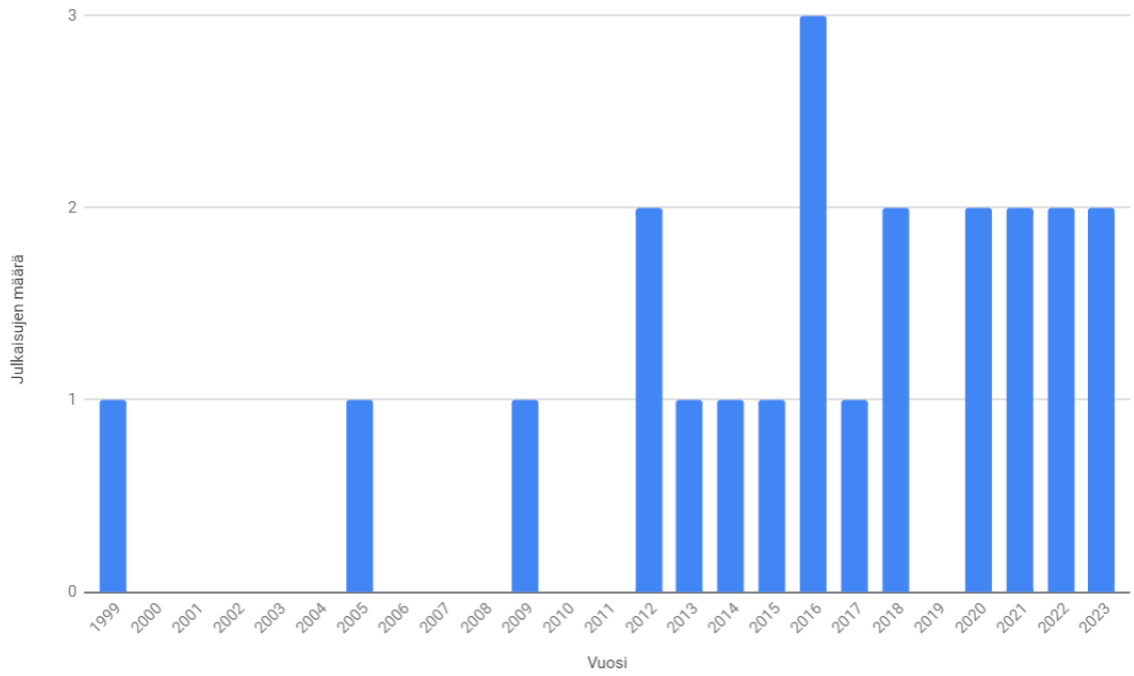
tuurin käsitteeseen liittyviä tutkimuksia jatkotutkimuksillaan soveltaen ohjelmistoarkkitehtuurin elementtejä testausarkkitehtuurin mallintamiseen (Jon D Hagar 2022) ja määrittäen testausarkkitehtuurin käsitteen UML-testausprofiilin avulla (Hagar ja Wendland 2023).

Testausarkkitehtuuria sovellettiin spesifioituihin käyttökohteisiin useissa tutkimuksissa. Jon D Hagar (2018) esitti testausarkkitehtuurin käyttämistä IoT-laitteiden testaamisessa demonstroiden IoT näkymien hyödyntämistä testaamisessa. Lee (2009) tutki testauskehyksen kehittämistä Service Oriented Architecture (SOA) -järjestelmään luoden referenssiarkkitehtuurin. Rodrigues ym. (2005) esittivät integraatiotestausarkkitehtuurin käyttämistä moniagenttijärjestelmässä luomaan testausraportteja sääntömääritelmien esityksen analysoinnin avulla. Lahami ym. (2012) tutkivat resurssitietoisen testausarkkitehtuurin suunnittelemista mukautuville ja hajautetuille järjestelmille.

Testausarkkitehdin roolia testausarkkitehtuurin kontekstissa ovat kuvanneet Jon ja Hagar (2020) ja Paulisch ja Zimmerer (2016). Paulisch ja Zimmerer (2016) esittivät miten testausarkkitehdin ja ohjelmistoarkkitehdin välinen yhteistyö ohjelmistokehityksen elinkaaren aikana on hyödyllistä. Jon ja Hagar (2020) tunnistivat tutkimuksessaan testausarkkitehdin taitoja ja vastuita.

5.3 Julkaisuvuodet

Kuviossa 5 on kuvattu tulosten määrä julkaisuvuosittain. Eniten tutkimuksia julkaistiin vuonna 2016. Kuviosta voidaan huomata testausarkkitehtuuriin liittyvien tutkimusten määrän olleen nousussa vuodesta 2012 lähtien. Kyseisenä vuonna Nishi julkaisi ”Viewpoint-based Test Architecture Design” -nimisen tutkimuksen, joka voidaan nimetä yhdeksi keskeisimmäksi testausarkkitehtuurin tutkimukseksi. Tutkimus tarjosi alalla pohjan testausarkkitehtuurin jatkotutkimuksille (Nishi 2012).



Kuvio 5. Valittujen tulosten määrä julkaisuvuosittain

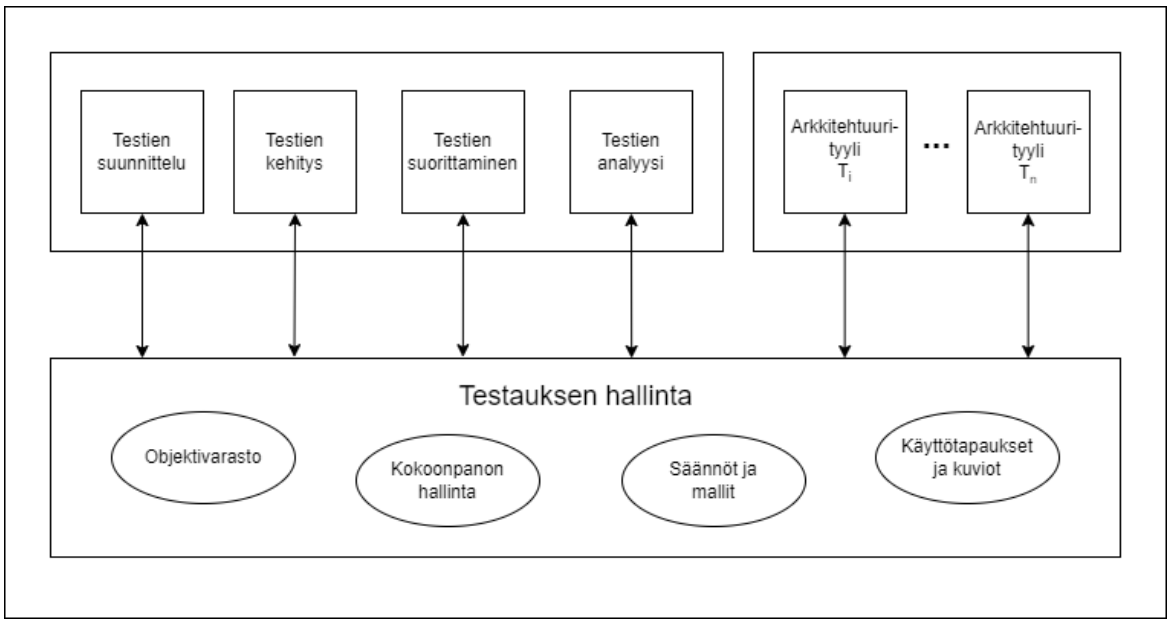
5.4 Tutkimustietokannat

Katsauksen tuloksista 16 tutkimusta eli noin 72 prosenttia kokonaistulosten määrästä valittiin IEEE Xplore -tutkimustietokannasta. Tulosten määrä tietokannoittain on kuvattu taulukossa 4 luvussa 4.3.2. Testausarkkitehtuuriin liittyviä ja valintakriteereitä vastaavia tutkimuksia löytyi IEEE Xploresta eniten, koska vuosittain International Conference on Software Testing, Verification and Validation – ICST:n – yhteydessä järjestettävä International Workshop on Software Test Architecture – InSTA – ohjeistaa tutkijat lähettämään tutkimuspaperinsa työpajaan IEEE konferenssijulkaisumuodossa. Työpajan päättymisen jälkeen lähetetyt tutkimukset julkaistaan IEEE Xploressa. Työpajassa tutkijat keskustelevat tutkimuksista ja uusista testausarkkitehtuuriin liittyvistä ideoista ja siksi suurin osa testausarkkitehtuuriin liittyvästä tutkimuksesta on julkaistu InSTA:n työpajoissa. InSTA tarjoaa myös suosituksia tutkimuksen aiheille ja teemoille vuosittain ”Call for papers” -ohjeissaan. Vuoden 2023 työpajan suositusten kategorioita olivat testausarkkitehtuurin käsitteet, testausarkkitehtuurin suunnittelu, testien vaatimusanalyysi ja testausarkkitehtuurin soveltaminen (“InSTA” 2023).

Muista tietokannoista tuloksia valittiin selkeästi vähemmän. Keskeisin syy vähäiselle määrälle oli se, että suurin osa muiden tietokantojen hakutuloksista eivät vastanneet valintakriteereitä. Google Scholarin hakutuloksissa oli mukana myös IEEE Xploressa julkaistuja tutkimuksia duplikaatteina. Tulosten määrästä käy ilmi myös kuinka vähän tieteellisiä julkaisuja on ja kuinka tarpeellista jatkotutkimus aiheesta on.

5.5 Testausarkkitehtuurin määritelmä

Cusick (1999) määritteli ensimmäisenä testausarkkitehtuurin ”ohjelmistotestiympäristön suunnittelussa käytettyjen alustojen, komponenttien ja lähestymistapojen välisinä suhteina ja rajoitteina suorittaa ohjelmistosovelluksen verifiointi ja validointi”. Cusick (1999) esitti myös, että testiympäristöt tulisi suunnitella ohjelmistoarkkitehtuurityylien näkökulmasta. Cusickin (1999) testausarkkitehtuurin viitekehys koostuu kolmesta osasta: testauksen hallinnasta, testaustoiminnasta ja testattavan ohjelmiston testitarpeista sen arkkitehtuuriin perustuen. Cusickin testausarkkitehtuurin viitekehys on kuvattu kuviossa 6.



Kuvio 6. Cusickin (1999) testausarkkitehtuurin viitekehys

Noin vuosikymmen tämän jälkeen Nishi esitti testausarkkitehtuurin jakamista kahteen osaan: testijärjestelmän arkkitehtuuriin ja testisarjan arkkitehtuuriin (Nishi 2012), (Nishi, Katayama ja Yoshizawa 2013). Hänen mukaansa testijärjestelmän arkkitehtuuri olisi tarkoitettu testijärjestelmälle, testattavalle ohjelmistolle, alustalle missä testattava ohjelmisto suoritetaan ja testitapausten generaattorille. Testisarjan arkkitehtuuri olisi tarkoitettu testisarjoille, jotka koostuvat testitapausten joukoista, testitasoista, testityypeistä ja testiehdoista.

Edistääkseen testausarkkitehtuurin tutkimusta ja käytänteitä Nishi (2012) esitti testauksen hallinnan ja suunnittelun erottamista toisistaan ja määritteli uudelleen testausprosessin suunnittelun näkökulmasta luoden testauskehityksen elinkaaren mallin (engl. Test Development Life Cycle). Tämä koostui neljästä vaiheesta: testivaatimusten analysoinnista, testausarkkitehtuurin suunnittelusta, testien yksityiskohtien suunnittelusta ja testauksen toteuttamisesta.

Testausarkkitehtuurin määritelmän kehitys eteni seuraavissa tutkimuksissa lähemmäksi ohjelmistoarkkitehtuurin ja Cusicking (1999) esittämää määritelmää kohti. Jon D. Hagar (2017) määritteli testausarkkitehtuurin järjestelmäsuunnittelun ja ohjelmistoarkkitehtuurin kautta olevan sekä prosessi että tuote testien suunnittelussa ja rakentamisessa sekä testausrakenteissa. Masudan, Nishin ja Suzukin (2020) toteuttama tutkimus oli suuri askel kohti testausarkkitehtuurin ja ohjelmistoarkkitehtuurin määritelmien yhtenäistämistä. Tutkimuksessa esitettiin metodi testausarkkitehtuurin suunnitteluun suurille ja monimutkaisille ohjelmistoille perustuen ohjelmistoarkkitehtuurin kansainvälisiin IEEE-standardeihin. Testausarkkitehtuurin käsitteellistäminen, arviointi ja tarkentaminen ovat metodin kolme prosessia (Masuda, Nishi ja Suzuki 2020). Käsitteellistämisessä määritellään ongelma-alue, arkkitehtoniset tavoitteet, kriittiset menestyskriteerit ja ratkaisutila. Arvioinnissa arvioidaan testaustasoja ja -tyyppejä testaustapoihin, raportoidaan arvioinnin tulokset ja viestitään suosituksista testausarkkitehtuurin parantamiseen. Tarkentamisessa tarkennetaan testaustasoja ja -tyyppejä arviointitulosten perusteella ja kuvataan testaustasoja ja -tyyppejä (Masuda, Nishi ja Suzuki 2020). Testaustasot ja -tyypit ryhmitellään ohjelmistoarkkitehtuuristandardin mukaisesti käyttäen testausnäkymää.

Uusimmat testausarkkitehtuurin tutkimukset määrittelevät testausarkkitehtuurin käsitteen noudattamalla ohjelmistoarkkitehtuurin määritelmää ja sen standardeja (Masuda ym. 2022), (Hagar ja Wendland 2023). Masuda ym. (2022) määrittelevät testausarkkitehtuurin sen tavoit-

teen näkökulmasta: ”Testausarkkitehtuurin tavoitteena on auttaa laajamittaisten ja monimutkaisten ohjelmistojen testaamisessa soveltamalla ohjelmistoarkkitehtuurin käytäntöjä ohjelmistotestaukseen.” Masuda ym. (2022) mainitsevat kuitenkin testausarkkitehtuurin käsitteen määritelmän olevan yhä epämääräinen, vaikka käsite on tunnistettu ja käytetty ohjelmistotestauksen alalla ja kirjallisuudessa. Käsitteestä ei ole myöskään yksimielisyyttä standardien maailmassa (León-Carrillo 2023).

Tässä tutkielmassa testausarkkitehtuuri määritellään olevan järjestelmän tai ohjelmiston testaamisessa käytettävä käsitteellinen esitys/mallinnus, joka määrittää testausympäristöt, testausjärjestelmän, testausautomaation, rakenteet sekä arkkitehtoniset kuvaukset ja näkökulmat noudattaen ohjelmistoarkkitehtuurin käytänteitä.

5.6 Testausarkkitehtuurin kuvaus ja tyyli

Masuda ym. (2022) mukaan testausarkkitehtuurissa näkökulmat ovat samat kuin ohjelmistoarkkitehtuurissa ja testauksen näkymät voidaan kuvata samalla tavalla kuin ohjelmistoarkkitehtuurissa. Testausarkkitehtuurin kuvaus pystytään myös luomaan esimerkiksi Kruchtenin (1995) 4+1-mallia noudattaen. Jon D Hagar (2022) määritti ISO/IEC/IEEE standardeihin perustuen testausarkkitehtuurin kuvaukseen liittyvät käsitteet, kuten testausnäkymän, testaushuolenaiheen, testausnäkökulman, testauskontin ja testausarkkitehtuurikehyksen. Jon D Hagar (2022) mukaan arkkitehtonisten testauskäsitteiden käyttäminen auttaa testauksen suunnittelua sekä vakiinnuttaa testausarkkitehtuurin ja sen käsitteiden asemaa alalla.

Arkkitehtuurin tyylillä tarkoitetaan arkkitehtuurin luokitusta, joka tarjoaa yhtenäisen sanaston suunnittelulle ja spesifioi semanttiset oletukset sanastosta (Miao, Sun ja Cao 2006), (Garlan 2008). Lisäksi arkkitehtuurin tyyli määrittää joukon sääntöjä komponenttien yhdistämisestä (Moaven ym. 2008). Hagar ja Wendland (2023) määrittelevät testausarkkitehtuurin tyyleiksi perusohjelmistojärjestelmän (engl. basic software system), perusohjelmistojärjestelmä plussan (engl. basic software system plus), erikoisohjelmistojärjestelmän (engl. special software system) ja erikoisohjelmisto- ja uniikkitestauslaitteistojärjestelmän (engl. special software and unique test hardware system).

5.7 Roolit

Testausarkkitehdillä tarkoitetaan ”henkilöä, joka määrittelee tavan, jolla testaus rakennetaan tietylle järjestelmälle, mukaan lukien testaustyökalut ja testausdatan hallinnan” (“ISTQB Glossary” 2024). Jon ja Hagar (2020) täydentävät tätä määritelmää määrittelemällä testausarkkitehdin henkilönä, joka käyttää strategioita, menetelmiä, käytäntöjä, prosesseja ja taktiikoita testausarkkitehtuurin kehittämiseen. Testausarkkitehtiä tarvitaan monimutkaisissa järjestelmissä, joissa on keskenään vuorovaikutuksessa olevia ohjelmistoelementtejä ja tarve testausarkkitehtuurille on tunnistettu (Jon ja Hagar 2020).

Testausarkkitehdin keskeisimpänä vastuuna on määrittää ja soveltaa asianmukaiset testausstrategiat, teknologiat, menetelmät, testausautomaatio ja testausympäristöt järjestelmän testaussuunnitelmalle. Testausarkkitehdin vastuulla on myös edellä mainittujen asioiden ylläpito ja kehitys (Jon ja Hagar 2020). Lisäksi testausarkkitehti vastaa testausarkkitehtuurin kuvaamisesta ja dokumentaatiosta. Kuvaamiseen kuuluvat testausarkkitehtuurikuvausten luominen käyttäen eri näkymiä, jotka käsittelevät testausjärjestelmän osia tietystä näkokohdasta sekä testaustasojen mallintaminen. Dokumentaatioon kuuluvat testausarkkitehtuurin riskien seuranta ja raportointi sekä arkkitehtonisten näkökulmien, kuvausten ja testausjärjestelmän suunnittelukonseptien määrittely yhdessä testauksesta vastuussa olevien henkilöiden kanssa (Jon ja Hagar 2020). Testausarkkitehdin tehtävien tarkoituksena on parantaa järjestelmän yleistä laatua tehokkaalla ja laadukkaalla testauksella järjestelmän koko elinkaaren ajan (Paulisch ja Zimmerer 2016).

Jon ja Hagar (2020) sekä Paulisch ja Zimmerer (2016) ovat tarkastelleet testausarkkitehdiltä vaadittavia taitoja. Keskeisimmiksi testausarkkitehdin taidoiksi he listaavat testaukseen, testausmetodeihin, -teknologioihin, suunnitteluun, arkkitehtuuriin, hallintaan, testattavaan ohjelmistoon, tuotealueen ymmärtämiseen ja ohjelmiston vaatimuksiin liittyvät taidot.

Testausarkkitehdin pitää tehdä yhteistyötä ohjelmistoarkkitehdin kanssa ohjelmistokehityksen alussa ja sen koko elinkaaren ajan (Paulisch ja Zimmerer 2016). Yhteistyön tarvitsee olla molemminpuolista tarkoittaen, että ohjelmistoarkkitehti osallistuu testauksen toimintoihin aktiivisesti ja testausarkkitehti osallistuu ohjelmistoarkkitehtoonisiin toimiin. Yhdessä ohjelmistoarkkitehti ja testausarkkitehti varmistavat, että testattava järjestelmä ja testijärjestelmä

toimivat yhdessä testattavuuden näkökulmasta tarkasteltuna ja testausautomaation arkkitehtuurin lähestymistapa on vaikuttava ja tehokas (Paulisch ja Zimmerer 2016).

6 Pohdinta

Tutkielmassa toteutettiin systemaattinen kirjallisuuskatsaus soveltaen Kitchenham ym. (2007) ohjeita systemaattisen kirjallisuuskatsauksen suorittamiseen. Tutkielmassa määriteltiin ja esitettiin katsausprotokolla, jonka sisältämien metodien mukaan systemaattinen kirjallisuuskatsaus toteutettiin. Tutkielman lopussa suoritettiin tulosten laadullinen syntetisointi mukaillen tutkimusaiheeseen liittyviä tarpeita ja tavoitteita. Tulosten laadullisessa syntetisoinnissa vastattiin tutkielman tutkimuskysymyksiin koskien testausarkkitehtuurin määritelmää.

6.1 Tutkielman validiteetti ja reliabiliteetti

Tutkielman validiteettia voidaan pitää hyvänä, koska tutkielmassa seurattiin metodilähteen ohjeita ja selkeää mallia, ja tuloksissa onnistuttiin vastaamaan tutkimuskysymyksiin. Lisäksi tutkielman aineisto kerättiin kolmesta Kitchenham ym. (2007) määritellystä ohjelmistokehittäjille tärkeästä tietokannasta.

Tutkielman reliabiliteetti on hyvä, koska tutkielmassa esitettiin katsausprotokolla, jossa kuvattiin tutkielmassa toteutetun systemaattisen kirjallisuuskatsauksen vaiheet. Lisäksi tutkielmassa kuvattiin tarkasti käytettävä hakustrategia pitäen sisällään hakutermit ja -lauseet, tietokannat, joista lähteitä etsittiin sekä käytetyt valinta- ja poissulkukriteerit.

6.2 Tutkielman rajoitteet

Tutkielmassa valittujen tutkimusten määrä jäi pieneksi osittain hakutermien ja hakulausekkeen optimoinnin haasteiden takia. Hakutermien ”software” ja ”test architecture” yhdistäminen koettiin haasteelliseksi, koska hakutuloksia löytyi kyseisillä hauilla paljon, mutta tulokset eivät vastanneet valintakriteereitä. Tämän vuoksi iso osa hakutuloksista rajaantui pois. Lisäksi Google Scholarin hakutulosten määrää jouduttiin rajoittamaan, jotta hakutulosten arviointiprosessi pysyisi hallittavissa.

6.3 Jatkotutkimuskohteet

Aihe tarvitsee jatkotutkimuksia testausarkkitehtuurin käsitteen määrittelyyn liittyen, jotta käsitteestä voidaan muodostaa yhteisymmärrys ohjelmistotestauksessa, kuten Jon D Hagar (2022) sekä Hagar ja Wendland (2023) mainitsevat. Yhteisymmärryksen muodostamisen jälkeen tarkemmat jatkotutkimukset ovat helpommin toteutettavissa. Aiheen jatkotutkimuksia kaivattaisiin testausarkkitehtuurin käyttöönottamiseen ja sen hyödyntämiseen yrityksissä. Tämän kaltaisella tutkimuksella konkretisoitaisiin testausarkkitehtuurin hyötyjä ja tulosten pohjalta pystyttäisiin laatimaan malleja alakohtaisista arkkitehtonisista ratkaisuista. Malleja pystyttäisiin uudelleenkäyttää tällöin muissa järjestelmissä. Testausdatan hallinnan ja generoimisen määrittely testausarkkitehtuurissa olisi myös arvokas jatkotutkimuskohde, koska monissa isoissa tietojärjestelmissä pitää pystyä replikoimaan käyttäjiä testausdatalla.

7 Yhteenveto

Tämän tutkielman tavoitteena oli selvittää mitä tieteellisiä lähteitä löytyy testausarkkitehtuurista sekä miten testausarkkitehtuurin käsite on määritelty tieteellisessä kirjallisuudessa ja miten se voidaan määritellä kirjallisuuden ja ohjelmistoarkkitehtuurin mukaan. Tutkielman tuloksista kävi ilmi, että testausarkkitehtuurista on julkaistu vain vähän kirjallisuutta, mutta kiinnostus aihetta kohtaan on noussussa ohjelmistotestauksen alalla. Systemaattiseen kirjallisuuskatsaukseen valituista 22 tutkimuksesta kahdeksan julkaistiin edellisten neljän vuoden aikana. Työpajat, kuten InSTA, ovat edistäneet aiheeseen liittyvää tutkimustyötä. Aihetta koskeva tutkimus kiihtyy todennäköisesti lähivuosien aikana, kun ohjelmistojärjestelmät muuttuvat yhä monimutkaisemmiksi ja niiden kokoluokka kasvaa. Testausarkkitehtuurin jatkotutkimus tulee vahvistamaan samalla myös sen asemaa ohjelmistotestauksessa.

Testausarkkitehtuurin käsitteelle on esitetty tutkimuksissa muutamia määritelmiä, joista uusimmat tarkastelevat testausarkkitehtuuria ohjelmistoarkkitehtuurin ja ohjelmistoarkkitehtuurin standardien näkökulmasta. Tässä tutkielmassa testausarkkitehtuuri määritellään olevan järjestelmän tai ohjelmiston testaamisessa käytettävä käsitteellinen esitys/mallinnus, joka määrittää testausympäristöt, testausjärjestelmän, testausautomaation, rakenteet sekä arkkitehtoniset kuvaukset ja näkökulmat noudattaen ohjelmistoarkkitehtuurin käytänteitä. Tutkielman tuloksista voidaan havaita, että ohjelmistoarkkitehtuurin käytänteiden soveltaminen testausarkkitehtuuriin on aiheen tulevaisuuden kehityssuunta. Testausarkkitehtuurin käsitteen määrittely vaatii vielä jatkotutkimuksia ja universaalien standardien laatimista.

Testausarkkitehtuurin kuvausta on käsitelty tutkimuksissa vähän. Masuda ym. (2022) mainitsivat ohjelmistoarkkitehtuurin kuvaukseen liittyvien käytäntöjen pätevän myös testausarkkitehtuurin kuvaukseen. Jon D Hagar (2022) määritti testausarkkitehtuuriin ja sen kuvaukseen liittyvät käsitteet. Tutkimusten vähäinen määrä kuvaa osittain, miten testausarkkitehtuurin käsitteeseen liittyvistä esityksistä ei ole vielä selkeää yhteisymmärrystä, jonka takia tarkempia jatkotutkimuksia ei ole suoritettu.

Testausarkkitehdin tehtäviä, vastuita ja taitoja on kuvattu tutkimuksissa sen sijaan kattavasti. Testausarkkitehdin rooli on myös kirjallisuudessa määritelty ja käsitteelle löytyy viralli-

sia kansainvälisiä määritelmiä (“ISTQB Glossary” 2024). Testausarkkitehdin pitää olla vuorovaikutuksessa eri ohjelmistokehityksen tiimien sekä ohjelmistoarkkitehdin kanssa. Jon ja Hagar (2020) sekä Paulisch ja Zimmerer (2016) toteavat testausarkkitehdin työnkuvan olevan laajempi kuin mikä on yleinen käsitys testausarkkitehdistä ja sen tehtävistä.

Lähteet

Anand, A ja A Uddin. 2019. “Importance of software testing in the process of software development”. *International Journal for Scientfic Research and Development* 12 (6).

Bertolino, Antonia. 2007. “Software Testing Research: Achievements, Challenges, Dreams”. Teoksessa *Future of Software Engineering (FOSE ’07),* 85–103. https://doi.org/10.1109/FOSE.2007.25.

Brannigan, Vincent. 1985. “Acceptance Testing - The Critical Problem in Software Acquisition”. *IEEE Transactions on Biomedical Engineering* BME-32 (4): 295–299. https://doi.org/10.1109/TBME.1985.325451.

Brar, Hanmeet Kaur ja Puneet Jai Kaur. 2015. “Differentiating Integration Testing and unit testing”. Teoksessa *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom),* 796–798.

Cusick, James. 1999. “Deriving Software Test Environments From Architecture Styles”. Kesäkuu.

Dookhun, Avishek Sharma ja Leckraj Nagowah. 2019. “Assessing The Effectiveness Of Test-Driven Development and Behavior-Driven Development in an Industry Setting”. Teoksessa *2019 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE),* 365–370. https://doi.org/10.1109/ICCIKE47802.2019.9004328.

Garlan, David. 2008. “Software architecture”.

Hagar, Jon. 2021. “Multi-company Consumer Product Software Test Architecture Industry Experience Report”. Teoksessa *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 158–161. https://doi.org/10.1109/ICSTW52544.2021.00036.

Hagar, Jon ja Marc-Florian Wendland. 2023. “Defining Software Test Architectures with the UML Testing Profile”. Teoksessa *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 271–280. https://doi.org/10.1109/ICSTW58534.2023.00056.

Hagar, Jon D. 2018. “Software Test Architectures and Advanced Support Environments for IoT”. Teoksessa *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 252–256. https://doi.org/10.1109/ICSTW.2018.00057.

———. 2022. “Software Architecture Elements Applied to Software Test: View, Viewpoints and Containers”. Teoksessa *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 248–252. https://doi.org/10.1109/ICSTW55395.2022.00051.

———. 2017. “Defining the Phrase "Software Test Architecture"Emerging Idea”. Teoksessa *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 313–316. https://doi.org/10.1109/ICSTW.2017.58.

Holling, Dominik, Andreas Hofbauer, Alexander Pretschner ja Matthias Gemmar. 2016. “Profiting from Unit Tests for Integration Testing”. Teoksessa *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST),* 353–363. https://doi.org/10.1109/ICST.2016.28.

“InSTA”. 2023. Viitattu 11. toukokuuta 2024. https://www.aster.or.jp/workshops/insta2023/.

“ISTQB Glossary”. 2024. Viitattu 12. toukokuuta 2024. https://glossary.istqb.org/.

Jon, D ja Laura Hagar. 2020. “Identifying Software Test Architect Skills and Knowledge”. Teoksessa *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 213–215. https://doi.org/10.1109/ICSTW50294.2020.00044.

Karac, Itir ja Burak Turhan. 2018. “What Do We (Really) Know about Test-Driven Development?” *IEEE Software* 35 (4): 81–85. https://doi.org/10.1109/MS.2018.2801554.

Kitchenham, Staffs ym. 2007. *Guidelines for performing systematic literature reviews in software engineering.*

Klammer, Claus ja Albin Kern. 2015. “Writing unit tests: It’s now or never!” Teoksessa *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 1–4. https://doi.org/10.1109/ICSTW.2015.7107469.

Kruchten, Philippe. 1995. “The 4+1 View Model of Architecture”. *IEEE Software* 12 (marraskuu): 45–50. https://doi.org/10.1109/52.469759.

Kruchten, Philippe. 2008. “What do software architects really do?” *Journal of Systems and Software* 81 (12): 2413–2416.

Lahami, Mariam, Moez Krichen, Mariam Bouchakwa ja Mohamed Jmaiel. 2012. “Using knapsack problem model to design a resource aware test architecture for adaptable and distributed systems”. Teoksessa *Testing Software and Systems: 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 19-21, 2012. Proceedings 24,* 103–118. Springer.

Lee, Jihyun ja Sungwon Kang. 2014. “Towards Test Architecture Based Software Product Line Testing”. Teoksessa *2014 IEEE 38th Annual Computer Software and Applications Conference,* 596–597. https://doi.org/10.1109/COMPSAC.2014.83.

Lee, Youngkon. 2009. “Double layered SOA test architecture based on BPA - simulation event”. Teoksessa *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human,* 218–223. ICIS ’09. Seoul, Korea: Association for Computing Machinery. ISBN: 9781605587103. https://doi.org/10.1145/1655925.1655964. https://doi-org.ezproxy.jyu.fi/10.1145/1655925.1655964.

León-Carrillo, Luis-Vinicio. 2023. “Elements for a Test(-ware) Architecture Language”. Teoksessa *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 292–299. Huhtikuu. https://doi.org/10.1109/ICSTW58534.2023.00059.

Liu, Zheng ja Paul Mei. 2014. “Automated testing for large-scale critical software systems”. Teoksessa *2014 IEEE 5th International Conference on Software Engineering and Service Science,* 200–203. https://doi.org/10.1109/ICSESS.2014.6933544.

Lukas, Spendla ja Hrcka Lukas. 2014. “Proposal of System Testing Integration into Safety Critical System Design Process Supported by SysML”. Teoksessa *2014 European Modelling Symposium,* 251–256. https://doi.org/10.1109/EMS.2014.74.

Maier, Mark W, David Emery ja Rich Hilliard. 2001. “Software architecture: Introducing IEEE standard 1471”. *Computer* 34 (4): 107–109.

Masuda, Satoshi, Jon Hagar, Yasuharu Nishi ja Kazuhiro Suzuki. 2022. “Software Test Architecture Definition by Analogy with Software Architecture”. Teoksessa *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 244–247. https://doi.org/10.1109/ICSTW55395.2022.00050.

Masuda, Satoshi, Yasuharu Nishi ja Kazuhiro Suzuki. 2020. “Complex Software Testing Analysis using International Standards”. Teoksessa *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 241–246. https://doi.org/10.1109/ICSTW50294.2020.00049.

Mathur, Sonali ja Shaily Malik. 2010. “Advancements in the V-Model”. *International Journal of Computer Applications* 1 (12): 29–34.

McBride, Matthew R. 2004. “The software architect: essence, intuition, and guiding principles”. Teoksessa *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications,* 230–235. OOPSLA ’04. Vancouver, BC, CANADA: Association for Computing Machinery. ISBN: 1581138334. https://doi.org/10.1145/1028664.1028764. https://doi-org.ezproxy.jyu.fi/10.1145/1028664.1028764.

Meng, Sichen, Zilong Pan, Wei Li, Shuai Xie, Chang Liu, Kang He ja Hongli Yang. 2010. “The “4+1“ view model on safe home system architecture”. Teoksessa *2010 IEEE International Conference on Software Engineering and Service Sciences,* 352–355. https://doi.org/10.1109/ICSESS.2010.5552450.

Meyer, Bertrand. 2008. “Seven Principles of Software Testing”. *Computer* 41 (8): 99–101. https://doi.org/10.1109/MC.2008.306.

Miao, Huaikou, Junmei Sun ja Xiaoxia Cao. 2006. “Formalizing and analyzing service oriented software architecture style”. Teoksessa *2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC’06),* 387–390. https://doi.org/10.1109/EDOC.2006.29.

Miller, Roy ja Christopher T Collins. 2001. “Acceptance testing”. *Proc. XPUniverse* 238.

Moaven, Shahrouz, Jafar Habibi, Hamed Ahmadi ja Ali Kamandi. 2008. “A Decision Support System for Software Architecture-Style Selection”. Teoksessa *2008 Sixth International Conference on Software Engineering Research, Management and Applications,* 213–220. https://doi.org/10.1109/SERA.2008.26.

Nidagundi, Padmaraj ja Margarita Lukjanska. 2016. “Introduction to adoption of lean canvas in software test architecture design”. *Computational Methods in Social Sciences* 4 (2): 23.

Nishi, Yasuharu. 2012. “Viewpoint-based Test Architecture Design”. Teoksessa *2012 IEEE Sixth International Conference on Software Security and Reliability Companion,* 194–197. https://doi.org/10.1109/SERE-C.2012.15.

———. 2015. “Design principles in test suite architecture”. Teoksessa *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 1–4. https://doi.org/10.1109/ICSTW.2015.7107426.

———. 2016. “Difference in Quality of Test Architecture between Service Providers and Subcontractors”. Teoksessa *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 14–16. https://doi.org/10.1109/ICSTW.2016.46.

Nishi, Yasuharu, Tetsuro Katayama ja Satomi Yoshizawa. 2013. “Combinatorial Test Architecture Design Using Viewpoint Diagram”. Teoksessa *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops,* 295–300. https://doi.org/10.1109/ICSTW.2013.82.

Nishi, Yasuharu, Satoshi Masuda, Hideto Ogawa ja Keiji Uetsuki. 2018. “A Test Architecture for Machine Learning Product”. Teoksessa *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 273–278. https://doi.org/10.1109/ICSTW.2018.00060.

Nishi, Yasuharu ja Yusuke Shibasaki. 2021. “Boosted Exploratory Test Architecture: Coaching Test Engineers with Word Similarity”. Teoksessa *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW),* 173–174. https://doi.org/10.1109/ICSTW52544.2021.00038.

Omrani, Valiallah ja Seyyed Ali Razavi Ebrahimi. 2013. “Software Architecture Viewpoint Models: A Short Survey”. *ACSIJ Advances in Computer Science: an International Journal* 2 (5).

Paulisch, Frances ja Peter Zimmerer. 2016. “Collaboration of software architect and test architect helps to systematically BRIDGE product lifecycle gap”. Teoksessa *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities,* 11–13. BRIDGE '16. Austin, Texas: Association for Computing Machinery. ISBN: 9781450341530. https://doi.org/10.1145/2896935.2896936. https://doi-org.ezproxy.jyu.fi/10.1145/2896935.2896936.

Premraj, Rahul, Gaco Nauta, Antony Tang ja Hans van Vliet. 2011. “The Boomeranged Software Architect”. Teoksessa *2011 Ninth Working IEEE/IFIP Conference on Software Architecture,* 73–82. https://doi.org/10.1109/WICSA.2011.19.

Reddy, Jogannagari Malla ja S. V. A. V. Prasad. 2016. “The role of verification and validation in software testing”. Teoksessa *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom),* 1298–1301.

Rodrigues, L, G Carvalho, R Paes ja C Lucena. 2005. “Towards an integration test architecture for open MAS”. Teoksessa *Proc. of the 1st Workshop on Software Engineering for Agent-oriented Systems/SBES.*

Runeson, P. 2006. “A survey of unit testing practices”. *IEEE Software* 23 (4): 22–29. https://doi.org/10.1109/MS.2006.91.

Scotchmer, Suzanne. 1991. “Standing on the shoulders of giants: cumulative research and the patent law”. *Journal of economic perspectives* 5 (1): 29–41.

Solms, Fritz. 2012. “What is software architecture?” Teoksessa *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference,* 363–373. SAICSIT '12. Pretoria, South Africa: Association for Computing Machinery. ISBN: 9781450313087. https://doi.org/10.1145/2389836.2389879. https://doi-org.ezproxy.jyu.fi/10.1145/2389836.2389879.

Tang, Antony, Jun Han ja Pin Chen. 2004. “A comparative analysis of architecture frameworks”. Teoksessa *11th Asia-Pacific software engineering conference,* 640–647. IEEE.

“The C4 model for visualising software architecture”. 2024. Viitattu 20. toukokuuta 2024. https://c4model.com/.

Valle-Gómez, Kevin J., Pedro Delgado-Pérez, Inmaculada Medina-Bulo ja José Magallanes-Fernández. 2019. “Software Testing: Cost Reduction in Industry 4.0”. Teoksessa *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST),* 69–70. https://doi.org/10.1109/AST.2019.00018.

Vázquez-Ingelmo, Andrea, Alicia García-Holgado ja Francisco J. García-Peñalvo. 2020. “C4 model in a Software Engineering subject to ease the comprehension of UML and the software”. Teoksessa *2020 IEEE Global Engineering Education Conference (EDUCON),* 919–924. https://doi.org/10.1109/EDUCON45650.2020.9125335.

Whittaker, J.A. 2000. “What is software testing? And why is it so hard?” *IEEE Software* 17 (1): 70–79. https://doi.org/10.1109/52.819971.