OHJELMISTON TESTAUKSEN AUTOMATISOINTI

Pentti Pohjolainen
Pro gradu -tutkielma
Tietojenkäsittelytieteen laitos
Kuopion yliopisto
Joulukuu 2003

KUOPION YLIOPISTO, informaatioteknologian ja kauppatieteiden tiedekunta
Tietojenkäsittelytieteen koulutusohjelma
Tietojenkäsittelytiede
POHJOLAINEN PENTTI, A.: Ohjelmiston testauksen automatisointi
Pro gradu -tutkielma, 89 s., 1 liite (16 s.)
Pro gradu -tutkielman ohjaajat:
FT, Anne Eerola
FM, Tanja Toroi
Joulukuu 2003

Avainsanat: Testaus, testauksen vaihejako, testausmenetelmät, testauksen automatisointi, automatisoinnin apuvälineet (työkalut)

Tämän pro gradu -tutkielman tarkoituksena on esitellä testausta sen historiasta nykypäivään. Varsinainen testausosuus, jossa on testauksen historia, vaihejako ja eri menetelmät, käsitellään lyhyesti referoiden.

Pääosa tutkimuksesta keskittyy testauksen automatisointiin. Tässä osiossa selvitetään, mitä testauksen automatisointi tarkoittaa, millaisia kuvauksia automatisointiprosessista on olemassa, miten niitä on hyödynnetty, missä on epäonnistuttu ja missä onnistuttu. Tarkastellaan myös automatisointityökalujen sopivuutta ohjelmistokehityksen eri vaiheisiin ja työkalun valintaprosessia.

Tutkielman lopussa esitetään PlugIT-projektin yhteydessä suoritetun yrityskyselyn tuloksia testauksen ja tarkastuksen osalta. PlugIT on TEKESin, muutamien Suomen sairaanhoitopiirien ja ohjelmistotalojen rahoittama projekti, jossa selvitetään terveydenhuollon tietojärjestelmien integrointimahdollisuuksia.

ESIPUHE

Tämä pro gradu -tutkielma on tehty Kuopion yliopiston Tietojenkäsittelytieteen laitokselle vuonna 2003. Tutkielmani ohjaajina toimivat FT Anne Eerola ja FM Tanja Toroi. Kiitän molempia alkuvaiheessa annetuista työn suuntaviivoista ja erityisesti Anne Eerolaa kärsivällisyydestä ja kannustavasta ohjauksesta koko pitkän työrupeaman ajan. FM Tomi Tikkanen antoi oman panoksensa lukemalla graduni läpi sen loppuvaiheessa ja esittämällä korjauksia ja parannuksia työhöni – kiitos.

Kiitoksen ansaitsevat myös Etramar Oy, Kuopion yliopiston Tietojenkäsittelytieteen laitos ja TEKESin ym. rahoittama PlugIT-projekti, joissa minulla on ollut onni työskennellä tutkielmaa tehdessäni.

Kiitos myös kaikille yrityskyselyyn vastanneille ja yht. yo Tarja-Liisa Kärkkäiselle vastausten tilastollisesta analysoinnista. Yrityskyselystä sain arvokasta tietoa tutkimukseeni.

Viimeisenä, mutta ei suinkaan vähäisimpänä lenkkinä tässä ketjussa on ollut vaimoni Marjatan tuki ja ymmärtäminen vaikeinakin hetkinä. Paljon kiitoksia Sinulle.

Kuopiossa 12.12.2003

_____

Pentti Pohjolainen

SISÄLLYS

1 JOHDANTO

"It was on one of my journeys between the EDSAC room and the punching equipment that ”hesitating at the angles of the stairs” the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.”

Maurice Wilkes, 1949 [Wil85]

Testauksen merkitystä ohjelmointiprosessissa ei ole syytä vähätellä, sen huomasi jo Wilkes yli puoli vuosisataa sitten [Wil85]. Mahdollisimman oikein toimivien ohjelmien ja ohjelmistojen luominen tehokkaasti ja taloudellisesti on ollut ongelma aina ohjelmoinnin alkuajoista saakka. On hyvin tiedossa, että systeemien ylläpitokustannukset voivat nousta todella korkeiksi – testauksessa ja jäljityksessä jopa 50 % - 80 % kokonaiskustannuksista [Bei84]. Testausmenetelmien ja apuvälineiden (työkalujen) käyttö on yksi ratkaisu tähän ongelmaan. Parhaimmillaan testauksen automatisointi on vähentänyt testauksen kustannuksia 80 % [FeG99]. Kuitenkin on huomattava, että yleensä automatisointi vie aikaa 3 – 10 kertaa enemmän kuin manuaalisen testin kertasuoritus, mutta säästö saadaan testauksen toistoista [Kan97]. Automatisointi ei ole suinkaan ongelmatonta. Monessa organisaatiossa on yllätytty siitä, että automatisointi on tullut kalliimmaksi kuin testaaminen manuaalisesti. Testaukset, jotka automatisoidaan, pitää valita huolellisesti. Testauksen automatisointiin vaikuttaa ainoastaan, kuinka taloudelliseksi automatisointi tulee, ja onko menetelmä kehityskelpoinen. Automatisointi ei saa olla mikään itsetarkoitus [FeG99].

Varsinainen testausosuus käsitellään alkuosassa hyvin lyhyesti referoiden. Testauksen automatisointi ja sen toteutus on pääasiallinen kohde, ja se käsitellään tarkemmin. Osuudessa käytetään hyväksi myös enemmän kuvia. Pientä asioiden päällekkäisyyttä eri luvuissa ei ole voinut välttää, koska useat kirjoittajat tarkastelevat samoja asioita lähes samoista näkökulmista. [FeG99]

Toisessa luvussa esitetään erilaisia määritelmiä testauksesta ja eri näkemyksiä siitä, miksi testausta tarvitaan, miten se suoritetaan ja milloin se voidaan lopettaa. Lopussa on lyhyt yhteenveto testauksen kehityksestä sen alkuajoista nykypäivään.

Kolmannessa ja neljännessä luvussa esitetään testauksen vaihejako ja erilaisia testausmenetelmiä.

Viides luku keskittyy selvittämään, mitä testauksen automatisointi tarkoittaa. Luvussa esitetään monen eri henkilön ajatuksia automatisoinnista. Eräs luvun keskeisimpiä osia on automatisoidun testauksen elinkaaren läpikäynti [DuR99]. Elinkaari käsittää päätöksen testauksen automatisoinnista, testityökalun hankintaprosessin, perehtymisen automatisoituun testaukseen, testauksen suunnittelun ja kehityksen, testien suorituksen ja hallinnan, sekä koko prosessin tarkastelun ja arvioinnin. Yhtenä osana lukua ovat menetelmät ja määritelmät testitapauksen arviointiin. Tämän jälkeen kerrotaan havaituista automatisoinnin eduista ja haitoista, ja esitellään onnistunut testauksen automatisoinnin strategia. Luvun lopussa pohdiskellaan automatisoidun testauksen ylläpidettävyyden ongelmia ja mahdollisia ratkaisuja ongelmiin. Lisäksi esitellään testauksen suunnittelun, suorituksen ja tulosten vertailun automatisointia.

Kuudennessa luvussa esitellään testauksen apuvälineitä ja tarkastellaan niiden sijoittumista systeemin elinkaaren eri vaiheisiin Fewsterin [FeG99] ja Tervosen [Ter00] mukaan, sekä eräs tapa miten apuvälineitä voidaan ryhmitellä eri luokkiin.

Seitsemännessä luvussa tarkastellaan apuvälineen valintaa ostamisen ja oman rakentamisen näkökulmista. Lisäksi tarkastellaan miten välineen käyttöönotto yrityksen sisällä saattaisi tapahtua.

Kahdeksas luku sisältää kokemuksia sekä onnistumisista että epäonnistumisista automatisoinnin yhteydessä. Lisäksi tässä luvussa esitellään tuloksia PlugIT-projektin yhteydessä suoritetun ohjelmistotuotannon nykytilakyselyn vastauksista testauksen ja tarkastuksen osalta [Plu03].

Viimeisessä luvussa esitän omia pohdintojani kaikesta edellä mainitusta ja muista tämän työn aikana esiin tulleista asioista.

2 TESTAUKSEN HISTORIAA JA NYKYPÄIVÄÄ

2.1 TESTAUKSEN MÄÄRITELMIÄ

Mitä testaus tarkoittaa? Myersin mukaan [Mye79] monet esittävät tästä prosessista virheellisiä määritelmiä, kuten esimerkiksi: ”Testauksella osoitetaan, että virheitä ei esiinny”, ”Testauksen tarkoitus on osoittaa, että ohjelma suorittaa sille määritellyt toiminnot oikein” ja ”Testauksella varmistutaan siitä, että ohjelma tekee sen, mitä sen oletetaan tekevän”. Oikeampi määritelmä kuitenkin on: ''Testaus on ohjelman suorittamista virheiden löytämiseksi''. Kun lisäämme tähän vielä, että hyvä *testitapaus* (syöte, ohjelman suoritus ja saadun tuloksen vertailu odotettuun tulokseen) on sellainen, jolla on suuri todennäköisyys löytää vielä havaitsematon virhe, ja että onnistunut testitapaus on sellainen, joka löytää uuden virheen, meillä on koossa kolme tärkeintä testauksen periaatetta.

Haikala [HaM01] määrittelee testauksen ''suunnitelmalliseksi virheiden etsimiseksi ohjelmaa tai sen osaa suorittamalla''. Testaus tapahtuu usein kokeilemalla ohjelmaa umpimähkäisesti jollain testiaineistolla. Tällöin testauksessa pyritään pikemminkin osoittamaan ohjelman toimivuus, kuin löytämään virheitä.

Testausta ovat määritelleet monet muutkin. Adrion sanoo testauksen olevan ''ohjelman käyttäytymisen tutkimista suorittamalla ohjelmaa testiaineistoilla'' [AdB82]. Glassin mukaan ''ohjelmaa suoritetaan tarkoituksena selvittää ovatko sen tuottamat tulokset oikeita'' [Gla79]. Dijkstra määrittelee ''testauksen selvittävän, onko ohjelmassa virheitä, mutta ei sitä, ettei niitä ole'' [DaD72]. Hennell väittää, että ''tarkoitus ei ole löytää virheitä, vaan vakuuttua siitä, että niitä ei ole''. Toisin sanoen osoitetaan, että tiettyjä virheluokkia ei esiinny [HeH84]. Hetzelin päätelmä on, että ''testaus on mikä tahansa toiminto, jolla arvioidaan ohjelman ominaisuuksia ja kelpoisuutta'', eli mitataan ohjelman laatua [Het85]. Rothmanin mukaan ''testauksen tarkoitus on selvittää perusteellisesti vaatimukset, päättää mitä laatu merkitsee ja määritellä testaussuunnitelma ja julkaisukriteerit siten, että ne ovat mitattavissa suunnitelman toteutumisen jälkeen'' [RoL99]. Whittakerin mielestä ''testaus on ohjelmiston suoritusprosessi, joka osoittaa toimiiko ohjelmisto määritysten mukaan sille tarkoitetussa ympäristössä'' [Whi00]. Tässä tutkielmassa *testaus* määritellään Myersin mukaisesti, eli testaus on ohjelman suorittamista virheiden löytämiseksi.

2.2 TESTAUKSEN TARKOITUS

Testausta tarvitaan, koska ohjelmisto ei ole mikään massatuotantotuote, joka toteuttaisi joitakin yleisesti määriteltyjä tehtäviä. *Ohjelmisto* on yksilöllisesti tehty kokonaisuus (*moduuleista* eli pienemmistä ohjelman osista), joka on tarkoitettu toimimaan joskus hyvinkin monimutkaisissa ympäristöissä ja sovelluksissa. Kokemus on osoittanut, että testauksesta täytyy huolehtia. [Rop94]

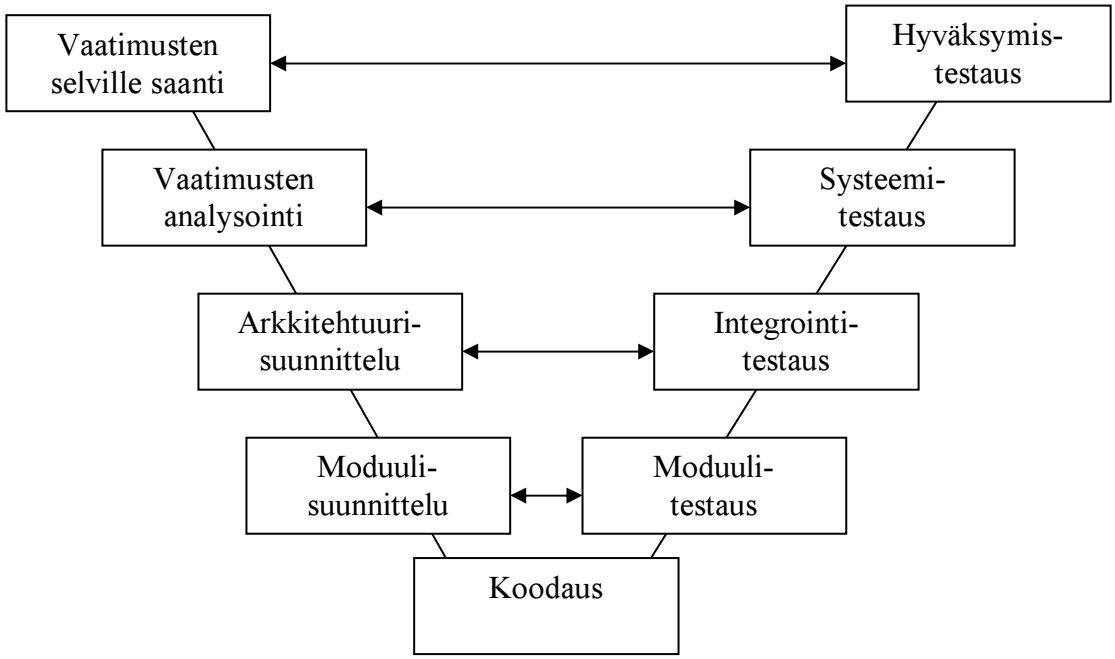
Testattaessa kannattaa tarkkailla lisääntyykö löytyneiden virheiden määrä tuntuvasti. Tällöin voi olla kyse siitä, että on opittu testaamaan paremmin tai on tehty enemmän virheitä tai kyse voi olla molemmista. Asia kannattaa kuitenkin tutkia. Kuuntele testaajia! Mikäli he valittavat, että aika ei riitä raportoimaan jokaisesta löydetystä virheestä, koska niitä on niin paljon, tilanne on todella vakava. Mikäli *integrointivaihe* (määritelty kappaleessa 3.2) kestää aina kauemmin ja kauemmin uusilla kierroksilla, merkitsee se sitä, että *komponentit* (ohjelman palaset) alkavat tulla rikkonaisemmiksi ja sisältävät yhä enemmän virheitä. Ratkaisu tähän ongelmaan on integrointi lyhyemmin aikavälein. Kuuntele kehittäjiä! Jos he valittavat, että aika ei riitä kaikkien testien suorittamiseen, siihen on syytä suhtautua vakavasti. Kiinnitä huomiota resurssivaatimuksiin! Aina, kun löytyy jokin *pullonkaula* eli kohta, jossa työskentely ei etene niin kuin sen pitäisi, se on arvokasta tietoa ongelmista. Kannattaa tarkastella ongelmakohtaa edeltävää vaihetta, onko siellä kaikki kunnossa. [Hen01]

Miksi testausta ei tehdä riittävän ajoissa, ja miksi kaikki eivät testaa? Eräs tyypillisimpiä väitteitä on, että moduulien testaaminen vie liikaa aikaa. Todellisuudessa integrointi- eli moduulien yhteenliittämisvaiheessa havaittujen virheiden korjaaminen vaatii aikaa moninkertaisesti enemmän kuin ajoissa suoritettu moduulien yksittäinen testaus. Ohjelmoijilla on myös usein vahva tunne omasta osaamisestaan. Virheen sattuessa hyvä ohjelmoija korjaa sen nopeasti ja vieläpä oikein. Kuitenkin parhaatkin ohjelmoijat tekevät virheitä. Nämä virheet ovat yleensä kaikkein vaikeimmin korjattavia. Kun tällaiset ohjelmoijat saadaan testaamaan, he huomaavat kehittyvänsä vielä myös ohjelmoijina. Monta kertaa heille on yllätys, miten paljon ja millaisia virheitä heidän koodistaan löytyy. [ThH02]

On hyvin tavallista, että testaajilla on erilainen näkökulma asioihin kuin ohjelmiston kehittäjillä. Erilaiset lähestymistavat lisäävät löytyneiden ratkaisujen määrää. Molemmin-

puolinen luottamus parantaa ryhmän ongelmanratkaisukykyä. Hyvät testaajat tajuavat, että ohjelmiston testaus on useasti yksitoikkoista saman asian toistamista. Monet kehittäjät vihaavat toistamista ja pyrkivät automatisoimaan sen. Hyvät testaajat uskaltavat myös raportoida löytämistään virheistä – sehän on heidän työnsä. Sitä vastoin heille ei kuulu mennä opastamaan kehittäjiä. [Pet00]

Testaaminen määriteltiin virheiden löytämiseksi. Virheistä voidaan löytää huomattava osa jo koodin *katselmuksessa* ja *tarkastuksessa* [Ter00]. Testauksen vaihejako *V-mallissa* alhaalta ylöspäin (Kuva 1) on moduulitestaus, integrointitestaus, systeemitestaus, hyväksymistestaus ja näihin kaikkiin liittyen regressiotestaus. Jokainen vaihe suoritetaan suunnittelupuolen (kuvassa vasemmalla) määrityksiin verraten. Vaatimusten selville saanti ja vaatimusten analysointi yhdessä muodostavat *vaatimusten määrittelyn* [Paa00].



Kuva 1. V-malli [Paa00]

Edellä olevassa ohjelmistokehityksen V-mallissa testausvaihe sijaitsee vasta prosessin loppuvaiheessa, kun kaikki kehitys- ja suunnittelutyö on jo tehty. Tässä vaiheessa ohjelmistosta löytyneiden virheiden kustannukset ovat kuitenkin hyvin korkeat. Ylläpitovaiheessa löytyneiden virheiden kustannukset ovat jopa 300 kertaa suuremmat

kuin jos virheet olisivat löytyneet vaatimusten määrittelyvaiheessa. Kokonaisvaltaisempi näkemys testauksesta onkin, että testausta pitää suorittaa koko ohjelmiston kehitysprosessin ajan. [GrH00]

Testaustyöhön pitää asettaa parhaat saatavilla olevat resurssit, antaa heille käyttöön tarvittavat apuvälineet ja kouluttaa käyttämään niitä. Testaajia täytyy kuunnella, kun he esittävät mielipiteitään ohjelmiston laatuvaatimuksista [Whi00]. On tärkeää huomata, että olio-ohjelmien testaus poikkeaa suuresti perinteisten ohjelmien testauksesta. Anu Partanen on käsitellyt pro gradussaan erityisesti olio-ohjelmien testausta [Par03].

Testitapausten laadinta ja suunnittelu on erittäin oleellinen osa testausta ja siten myös suuri ongelma [Mye79]. Ongelma on se, että testitapausten määrä pitäisi pystyä minimoimaan ja samanaikaisesti testauksen kattavuus pitäisi maksimoida. Mitä aikaisemmassa vaiheessa ohjelmistotuotantoprosessia testitapausten suunnittelu aloitetaan, sitä aikaisemmin virheet havaitaan ja sitä halvempaa on niiden korjaaminen [Jän03]. Jäljelle jää vielä siitä päättäminen, milloin ja missä vaiheessa testausta voidaan ja kannattaa automatisoida. Marickin mukaan [Mar98] kannattaa miettiä tarkoin automatisoinnin vaatima työmäärä, kustannukset ja testin elinikä. Kuinka monta koodin muutosta automatisoitu testaus kestää? Kustannuksia tulee pohtia säästettyjen manuaalisten testien määrässä, ja sen suhteen montako virhettä jää huomioimatta manuaalisessa testauksessa. Yhteenvetona edellisestä voi esittää kysymyksen: kannattaako automatisoida lainkaan?

Milloin testaus voidaan lopettaa? Kaksi yleisintä kriteeriä ovat testaukselle varatun ajan ylittyminen ja kaikkien testitapausten suorittaminen löytämättä yhtään virhettä. Molemmat ovat kelvottomia, koska ensinnäkin aika voidaan ylittää tekemättä yhtään mitään ja toiseksi voidaan olla kiinnittämättä huomiota testitapausten kattavuuteen. On siis mahdollista suunnitella sellaiset testitapaukset, että ohjelma näyttää toimivan, vaikkei näin todellisuudessa ole [Mye79]. Artikkelissaan ”Software Negligence and Testing Coverage” Cem Kaner luettelee yli 100 erilaista asiaa, jotka kuuluvat testauksen kattavuuteen [Kan96]. Esimerkkeinä mainittakoon *rivikattavuus* (*line coverage)*, jolloin jokainen ohjelman rivi käydään läpi, *silmukkakattavuus (loop coverage)*, jossa tutkitaan silmukoita, jotka suoritetaan useammin kuin kerran ja jokaisen *keskeytystilanteen läpikäynti*.

Myersin mukaan on olemassa kolme käyttökelpoisempaa kriteeriä testauksen lopettamiselle: Ensinnäkin voidaan käyttää testitapausten suunnittelumenetelmiä, kuten *moniehtokattavuutta, raja-arvoanalyysiä* ja *ekvivalenssiluokitusta* [Mye79]. Moni-ehtokattavuus tarkoittaa sitä, että testataan ehtolauseita ehtojen kaikilla kombinaatioilla. Mikäli 100 % kattavuus on saavutettu, voidaan sitä käyttää lopettamiskriteerinä. Raja-arvo-analyysissä tarkastellaan esimerkiksi taulukoiden ylä- ja alarajoja silmukan sisällä. Ekvivalenssiluokituksessa jaetaan testitapaukset ominaisuuksiensa perusteella osajoukkoihin, joita käsitellään yhtenäisinä kokonaisuuksina. Jokaisesta joukosta valitaan yksi tai useampia edustajia testitapauksiksi. Näin saadaan pienennetyksi testitapausten määrää [Paa00].

Toisessa tapauksessa testaus on suoritettu, kun tietty määrä virheitä on löytynyt. Vaikeutena on tietää etukäteen se virheiden määrä, joka pitää löytyä. Arviointi jollakin menetelmällä on miltei ainoa mahdollisuus. Tässä piilee kuitenkin yliarvioinnin vaara – jos virheitä ei ole olemassa niin paljon kuin niitä yritetään löytää jolloin testaus ei lopu koskaan. [Mye79]

Kolmas ja viimeisin menetelmä näyttää helpolta, mutta vaatii paljon asiantuntemusta ja kokemusta. Siinä piirretään graafi löytyneistä virheistä tietyssä aikayksikössä ja sen perusteella päätetään, voidaanko tämä testausvaihe lopettaa ja siirtyä seuraavaan. Paras tapa käytännössä lienee käyttää kaikkien edellä mainittujen kolmen kriteerin yhdistelmää. [Mye79]

Eräs menetelmä ohjelmiston valmiusasteen varmistamiseksi on käyttää *julkaisukriteerejä*. Tämä tarkoittaa sitä, että tutkitaan täyttääkö tuote kaikki ne kriteerit, jotka vaaditaan, jotta se voidaan luovuttaa asiakkaalle. Kriteerien olisi hyvä täyttää seuraavat ominaisuudet: y*ksityiskohtainen (specific),* mitattavissa oleva *(measurable), saavutettava (attainable), olennainen (relevant)* ja *seurattava (trackable).* Yksityiskohtaisuus tarkoittaa, että kriteeri kuuluu tiettyyn elinkaaren vaiheeseen. Mitattavuus auttaa vertaamaan onko kriteeri jo saavutettu. Saavutettavuus tarkoittaa, että kriteerit on mahdollista täyttää. Olennaisuudella tarkoitetaan ohjelman arviointia asiakkaan ja johdon vaatimuksien suhteen. Seurattavuudella mahdollistetaan ohjelman tilojen arviointi koko projektin ajan. [Rot02]

Testauksessa, kuten missä tahansa virheiden etsimisprosessissa, kannattaa pitää välillä lepohetkiä. Pienen tauon jälkeen ajatukset toimivat paljon kirkkaammin ja löytävät uusia polkuja kuljettavaksi. [Mar95]

2.3 TESTAUKSEN KEHITYS

Ohjelmistojen testauksen historia kuvastaa itse ohjelmistojen kehitystä. Alkuaikoina testaussuunnitelmat kirjoitettiin paperille ja itse testaus tapahtui myös kynän ja paperin avulla ns. *pöytätestauksena*. Testaus kohdentui vuokaavioihin ja toimintopolkuihin näissä kaavioissa. Koko systeemi testattiin äärellisellä määrällä testausmenetelmiä. Testaus aloitettiin yleensä vasta projektin loppuvaiheessa, ja sen suorittivat ne henkilöt, joilla silloin sattui olemaan aikaa. Henkilökohtaisten työasemien tulo aloitti uuden aikakauden testauksessa. On-line -systeemien testaaminen vaati aivan uuden lähestymistavan testauksen suunnitteluun ja toteuttamiseen, koska töitä voidaan kutsua suoritukseen melkein missä järjestyksessä tahansa. [DuR99]

Roperin mukaan varhaisemmilla testaustekniikoilla pyrittiin saavuttamaan ja määrittämään parempia testauksen kattavuuksia. Tämä tapahtui testaamalla ohjelmaa tutkien ohjelman rakennetta, kuten muuttujia, haarautumia, moniehtokattavuuksia ja tiedon kulkua ohjelmassa. 1980-luvun puolivälistä lähtien on kiinnostuttu virhepohjaisesta testauksesta. Sen sijaan, että yritettäisiin suorittaa ohjelmaa suuremmilla ja suuremmilla testiaineistoilla, keskitytään generoimaan testitapauksia, jotka ovat kattavampia virheen löytämisen kannalta. [Rop94]

Eräs tämän päivän avainsanoja on testauksen automatisointi (ks. luvut 5 - 7). Apuvälineitä automatisointiin on valmistettu todella paljon ja valmistetaan koko ajan lisää [Poh02]. Kirjassaan ''Software Testing in The Real World'' Kit toteaakin: ”Ohjelmistojen testauksen apuvälineiden aika on nyt” [Kit95].

3 TESTAUKSEN VAIHEJAKO

Testauksen vaihejaossa lähdetään liikkeelle V-mallin (Kuva 1, sivu 11) alaosasta ja liikutaan oikeaa haaraa ylöspäin, siirtyen yksittäisten moduulien testauksesta aina suurempien kokonaisuuksien testaukseen.

3.1 MODUULITESTAUS

Moduulitestauksessa (unit testing) testataan yksittäisiä moduuleja. Testauksen suorittaa yleensä moduulin tekijä yksin tai työparinsa kanssa. Tässä vaiheessa voidaan joutua käyttämään *testipetejä*, joilla ohjelman toimintaa kokeillaan, eli *ajuri- ja tynkämoduuleita (driver and stubs)*, jotka simuloivat ohjelman ympäristöä. Näiden käyttö selitetään tarkemmin kappaleissa 4.5 ja 4.6. Ohjelman toimintaa verrataan moduulisuunnittelun tuloksiin. Tähän vaiheeseen kuuluvat *lasilaatikko-* ja *harmaalaatikko* -testaukset, jotka esitellään tarkemmin luvussa 4. [HaM01]

Artikkelissaan ”Learning to Love Unit Testing” Thomas ja Hunt perustelevat, miksi moduulitestaus kannattaa tehdä paloina. He määrittelevät kyseisen testauksen pienten ohjelmapalasten testaamiseksi. Palasia ovat metodit, tietyt ohjelmapolut ja funktiot. Ohjelmoija tekee koodin paloittain ja testaa jokaisen osan erikseen. Seuraavaan vaiheeseen siirrytään vasta, kun edellinen vaihe toimii. Pienet askeleet antavat tekijälle onnistumisen tunteen ja ovat helpompia hallita. [ThH02]

3.2 INTEGROINTITESTAUS

Integrointitestauksessa (integration testing) yhdistetään moduuleita ja moduuliryhmiä. Myös tähän vaiheeseen liittyvät käsitteet ajuri ja tynkämoduuli. Ajurillla korvataan kutsuva moduuli ja tynkämoduulilla kutsuttavat moduulit. Lähinnä tutkitaan, kuinka moduulien väliset rajapinnat toimivat. Tuloksia verrataan tekniseen määrittelyyn. Integrointi etenee joko ”ylhäältä alas*” jäsentävästi (top-down)* tai *”alhaalta ylös” kokoavasti (bottom-up)* [HaM01]. Menetelmänä voi olla myös *kertarysäys (big-bang)*, jolloin kaikki moduulit

linkitetään yhteen samalla kertaa. Jäsentävän ja kokoavan menetelmän yhdistelmää kutsutaan *kerrosvoileipä (sandwich)* tekniikaksi [Paa00]. Tekniikoita käsitellään enemmän luvussa 4.

3.3 SYSTEEMITESTAUS

Systeemitestaus l. järjestelmätestaus käsittää koko järjestelmän testaamisen ja tulosten vertaamisen määrittelydokumentaatioon. Järjestelmätestaukseen kuuluvat myös ei toiminnalliset osuudet: kuormitustestit, luotettavuustestit, asennustestit ja käytettävyystestit. [HaM01]

Kuormitustestaus (load testing) voidaan jakaa kolmeen osaan: k*uormitus (stress)*, *pysyvyys (stability)* ja *paikallistaminen (isolation).* Kuormittamalla selvitetään, kuinka suurta kuormitusta järjestelmä kestää. Pysyvyystestauksessa tutkitaan systeemin tilaa pitkällä aikavälillä – aina vuorokaudesta useisiin viikkoihin – kuormituksen pysyessä koko ajan samana ja melko korkeana. Järjestelmän pitäisi säilyttää toimintakelpoisuutensa muuttumattomana. Paikallistaminen auttaa testaajaa selvittämään, missä päin sovellusta havaittu virhe sijaitsee. Testaaminen tapahtuu suorittamalla samat testitapaukset aina uudelleen ja uudelleen täsmälleen samalla tavalla. Kaikki edellä mainitut vaiheet liittyvät erityisesti verkkosovellusten testaukseen. [Asb00]

3.4 HYVÄKSYMISTESTAUS

Ennen lopullista käyttöönottoa asiakas ja käyttäjät yhdessä varmistavat *hyväksymistestauksella*, että systeemi täyttää sille asetetut todelliset tavoitteet. Testaus tapahtuu asiakkaan luona ja sitä kutsutaan *beta-testaukseksi*. Mikäli mukana on myös valmistajataho ja testaus suoritetaan valmistajan laitteistoilla, kyseessä on *alfa-testaus*. [Paa00]

3.5 REGRESSIOTESTAUS

Regressiotestaus tarkoittaa järjestelmän tai komponentin testausta, jotta voidaan osoittaa, että korjaukset ja muutokset eivät ole aiheuttaneet uusia virheitä, ja että ohjelma toimii määritysten mukaisesti. Regressiotestaus on tärkeää, koska muutokset ja korjaukset ovat alkuperäistä kehitystyötä virhealttiimpia. Menetelmä sopii hyvin laadunvarmistukseen ja käytettäväksi kehitystyön ja etenkin ylläpidon aikana [Paa00]. Käytännössä erittäin suuri osa testauksesta on tätä. Automatisointi soveltuu erittäin hyvin regressiotestaukseen [HaM01].

3.6 KÄYTETTÄVYYSTESTAUS

Käytettävyystestauksella varmistetaan, että tuotteesta tulee loppukäyttäjälle helppo käyttää. Tämä testaus on usein käyttöliittymän testausta. Sitä voidaan tehdä jo määrittelyvaiheessa käyttöliittymäprototyypin avulla. Testejä varten valitaan pieni otos tulevista käyttäjistä. Heidän suoriutumistaan eri tehtävissä seurataan valvotussa tilanteessa. [HaM01]

4 TESTAUSMENETELMÄT

Testauksen menetelmät ovat mustalaatikko-, lasilaatikko-, harmaalaatikko-, integraatio- (kertarysäys, jäsentävä, kokoava ja kerrosvoileipä) sekä tilastollinen testaus [Paa00]. Edellisten lisäksi on vielä muuttumattomuustestaus [Wei01] sekä virheiden kylväminen ja mutaatiotestaus [HaM01], tutkiva testaus [Bac01], äärimmilleen viety testaus [Jef99] ja aluetestaus [KaB02].

4.1 MUSTALAATIKKO

Mustalaatikkotestauksessa (black-box) testitapaukset valitaan testattavan ohjelman spesifikaatioiden perusteella tutustumatta ohjelman toteutukseen. Testitapauksia valittaessa voidaan syöteavaruus jakaa *ekvivalenssiluokkiin*. Tätä kutsutaan *ekvivalenssiositukseksi.* Oletuksena on, että jos ohjelma toimii yhdellä luokan edustajalla, se toimii myös muillakin, eli testauksessa ei tarvita kaikkia luokan testitapauksia. Tässä yhteydessä tulevat esille käsitteet *kelvollinen (valid)* syöte, *kelvoton (invalid)* syöte ja *virheellinen (illegal)* syöte. Ekvivalenssiluokkien tyypillisten edustajien lisäksi testitapauksiksi valitaan luokkien rajoilla olevia tapauksia. Tätä kutsutaan *raja-arvoanalyysiksi*. Mitä ylemmäs V-mallin oikeassa haarassa siirrytään, sitä enemmän testaus muuttuu mustalaatikkotestaukseksi. [HaM01]

4.2 LASILAATIKKO

Lasilaatikkotestauksessa (white-box) testitapausten valinnassa käytetään hyväksi tietoa ohjelman toteutuksesta [HaM01]. Toisin kuin mustalaatikkotestauksessa ohjelmakoodi on nyt käytettävissä. Lasilaatikkotestaus voidaan jakaa kontrollivirtaan perustuvaan testaukseen, silmukkatestaukseen ja tietovirtaan perustuvaan testaukseen [Paa00].

Kontrollivirtaan perustuvassa testauksessa tarkastellaan ohjelmaa usealta eri taholta. Tutkittaessa *lausekattavuutta* katsotaan, onko kaikille ohjelmasta muodostetun vuokaavion lauseille olemassa vähintään yksi suorituspolku, joka sisältää ko. lauseen. *Päätöskattavuutta* testattaessa pitäisi jokaiselle vuokaavion kaarelle olla ainakin yksi

suorituspolku, joka sisältää kyseisen kaaren. *Ehtokattavuuskriteeri* täyttyy, jos kaikki ehtosolmun erilliset ehdot saavat arvot tosi ja epätosi. *Moniehtokattavuudessa* täytyy jokaisen ehtosolmun erillisten ehtojen kaikki kombinaatiot käydä läpi ainakin kerran. *Polkukattavuuden* ehdot täyttyvät 100 prosenttisesti, jos suorituspolkujen joukossa ovat kaikki suorituspolut alkusolmusta lopetussolmuun. Käytännössä täydelliseen polkukattavuuteen on mahdotonta päästä, koska ohjelman silmukat lisäävät mahdollisten polkujen määrän liian suureksi. Tämän vuoksi onkin kehitetty menetelmä, jota kutsutaan *riippumattomaksi polkukattavuudeksi*. Siinä otetaan mukaan testaukseen kaikki sellaiset polut, jotka eivät muodostu mistään toisten polkujen alipoluista. [Paa00]

Silmukkatestausta suoritetaan, koska silmukat ovat ohjelman ydinrakenteita. Ne ovat myös tutkimusten mukaan ohjelman virhealttiimpia alueita. Silmukoita täytyy testata useammalla arvolla kuin mitä kattavuuskriteerit määräävät. Silmukoita voi olla yksinkertaisia, sisäkkäisiä, peräkkäisiä ja ei-rakenteisia. [Paa00].

Tietovirtatestauksessa keskitytään selvittämään tietovirran kulku ohjelmasta muodostetun graafin solmujen välillä. Solmun sisäinen tietovirta ei ole tärkeää. Tietovirtatestauksen riittävyyttä mitataan alipolkuina muuttujan määrittelysolmuista laskentasolmuihin. Menetelmiä ovat esimerkiksi: *kaikki määrittelyt (all-definitions), kaikki käytöt (all-uses)* ja *kaikki määrittelyt/käytöt (all-du-paths).* Kaikki määrittelyt -kriteeri täyttyy, kun jokainen määrittely saavuttaa ainakin yhden käyttönsä. Kaikki käytöt -tapauksessa kaikki määrittelyt saavuttavat kaikki mahdolliset käyttönsä. Viimeisessä ja myös kattavimmassa kaikki määrittelyt/kaikki käytöt -tapauksessa kaikki määrittelyt saavuttavat kaikki käyttönsä kaikkia mahdollisia määrittelyvapaita polkuja pitkin. *Määrittelyvapaa polku* ei sisällä uutta jo määritellyn muuttujan määrittelyä. [Paa00]

4.3 HARMAALAATIKKO

Harmaalaatikkotestauksessa (gray-box) käytetään hyväksi tietoa ohjelman toteutusperiaatteista. Voidaan hyödyntää esimerkiksi sellaista tietoa, että luvut ohjelmaan luetaan merkki kerrallaan ja muunnetaan samalla 32-bittisiksi kokonaisluvuiksi. [HaM01]

4.4 KERTARYSÄYS

Kertarysäyksessä (big-bang) moduulit yhdistetään ja testataan yhtenä kokonaisuutena. Ratkaisu on helppo toteuttaa mutta, jos virheitä esiintyy, on vaikeaa päätellä missä päin ohjelmistoa ne ovat. Tätä testausmuotoa ei voida aloittaa, ennen kuin kaikki moduulit on testattu toimiviksi yksinään. Tämän takia kertarysäys ei olekaan suositeltava monimutkaisille ja isoille ohjelmistoille. Kertarysäys on integrointitestauksen menetelmä. [Paa00]

4.5 JÄSENTÄVÄ

Jäsentävässä (top-down) integroinnissa aloitetaan moduulien yhdistäminen ylhäältä ja edetään järjestyksessä tasoittain alaspäin. Toisin sanoen aloitetaan pääohjelmasta ja siirrytään eteenpäin sen kutsumiin moduuleihin, rekursiivisesti eteenpäin alimmalle tasolle saakka. Voi olla, että kaikki kutsuttavat moduulit eivät ole vielä valmiina, ja silloin tarvitaan tynkämoduuleita, jotka pystyvät vastaamaan kutsuun. Tämä menetelmä mahdollistaa nopean testauksen ohjelmiston päätoiminnoille. Haittapuolina ovat tynkien ohjelmointityö ja se, että alimman tason moduuleihin päästään käsiksi viimeiseksi. Nämähän ovat usein kaikkein eniten käytetyt ja myös kriittisimmät systeemin toiminnan kannalta.[Paa00]

4.6 KOKOAVA

Kokoavassa (bottom-up) integroinnissa testataan ensin alimman tason moduulit itsenäisesti ja siirrytään ylöspäin niihin moduuleihin, jotka kutsuvat jo testattuja moduuleja. Nyt voidaan tarvita ajureita (driver), jotka ovat kutsuvan moduulin roolissa. Ajuri on usein helpompi koodata kuin tynkä. Menetelmällä on myös heikkouksia. Ylimmän tason moduulit voivat olla käyttäjälle tärkeimpiä ja nyt niihin päästään viimeiseksi. Näinpä käytet-tävyysvirheiden löytyminen siirtyy testauksen loppuvaiheeseen. Tämä voi olla systeemin kannalta vaarallista, koska ylimmän tason virheet voivat johtua huonosta suunnittelusta ja vaativat siten paljon korjaustyötä. [Paa00]

4.7 KERROSVOILEIPÄ

Kerrosvoileipä (sandwich) -integroinnissa yhdistetään jäsentävä ja kokoava menetelmä. Jäsentävää menetelmää sovelletaan ylimmän tason moduuleihin ja samanaikaisesti kokoavaa menetelmää alimmalla tasolla. Keskikerroksissa voidaan käyttää soveltuvin osin molempia menetelmiä. [Paa00]

4.8 TILASTOLLINEN

Tilastollisessa (statistical) testauksessa keskitytään niihin ohjelmiston osa-alueisiin, joita todennäköisimmin käytetään useimmin. Menetelmä perustuu järjestelmän toimintaprofiilin tutkimiseen. Tämä tapahtuu jakamalla syöteavaruus erillisiin luokkiin ja antamalla jokaiselle luokalle todennäköisyysarvo. Arvo kertoo, millä todennäköisyydellä käyttäjä antaa kyseisen luokan alkion järjestelmän syötteeksi. [Paa00]

4.9 MUUTTUMATTOMUUS

Muuttumattomuustestauksessa (idempotency) syöte A ajetaan ohjelmiston läpi. Saatu tuloste B ajetaan syötteenä uudelleen saman prosessin läpi. Mikäli uusi tuloste C ja tuloste B ovat yhtäläiset, on kyse muuttumattomuudesta ja ohjelmisto toimii oikein. Esimerkiksi testattaessa ohjelmaa, joka poisti tiedostosta moninkertaiset samat tietueet huomattiin, että kaksi kertaa esiintyvät poistuivat ensimmäisessä ajossa, mutta kolme kertaa esiintyvät poistuivat vasta toisessa ajossa. Muuttumattomuussääntö ei toteutunut, ja ohjelma ei siis toiminut oikein. [Wei01]

4.10 VIRHEIDEN KYLVÄMINEN

Virheiden kylvämisessä (error seeding) virheitä sijoitetaan tarkoituksellisesti lähdekoodiin ja pyritään sitten löytämään ne. Menetelmän huono puoli on se, että se aiheuttaa lisätyötä ja joitakin kylvettyjä virheitä voi vahingossa jäädä ohjelmaan. [Mye79]

4.11 MUTAATIOTESTAUS

Mutaatiotestauksessa (*mutation testing)* testauksen tehokkuutta arvioidaan tekemällä ohjelmasta useita eri versioita, joihin jokaiseen on kylvetty eri virhe, jotka pyritään sitten löytämään. [HaM01]

4.12 TUTKIVA TESTAUS

Tutkivassa testauksessa (exploratory testing) edetään ryöppysarjoina. Samanaikaisesti opetellaan tuntemaan tuote, suunnitellaan testit, suoritetaan ne ja raportoidaan virheistä. Testaus perustuu tuotemalliin, jota kehitetään samalla. Tämä testausmuoto edellyttää hyvin tiivistä henkilökohtaista kommunikointia testaajien ja tuotteen käyttäjien välillä. [Bac01]

4.13 ÄÄRIMMILLEEN VIETY TESTAUS

Äärimmilleen viety testaus (extreme testing) vaatii toteutettavaksi kaksi periaatetta. Aina kun ohjelmistoon lisätään uusi piirre, on yksikkötestaus suoritettava. On myös varmistuttava siitä, että koko systeemissä mikään ei toimi virheellisesti, eli kaikki yksikkötestaukset on suoritettava lisäyksen jälkeen uudelleen, ja niiden on onnistuttava sataprosenttisesti. [Jef99]

4.14 ALUETESTAUS

Alue on joukko, joka sisältää funktion kaikki mahdolliset muuttujan arvot. *Aluetestauksessa (domain testing)* identifioidaan funktiot ja muuttujat. Muuttujat voivat olla joko syötteitä tai tulosteita. Mahdolliset arvot jaetaan ekvivalenssiluokkiin ja jokaisesta luokasta otetaan pieni määrä edustajia mukaan. Tämän jälkeen testataan valituilla edustajilla jokainen funktio, joka sisältää nämä edustajat syötteenä tai tulosteena. [KaB02]

5 TESTAUKSEN AUTOMATISOINTI

5.1 MITÄ TESTAUKSEN AUTOMATISOINTI ON?

Automatisointi on manuaalisen testauksen suorittamista koneellisesti, jonkin testausohjelman avulla. Kuitenkin ihminen on edelleen testausjärjestelmän tärkein osa.

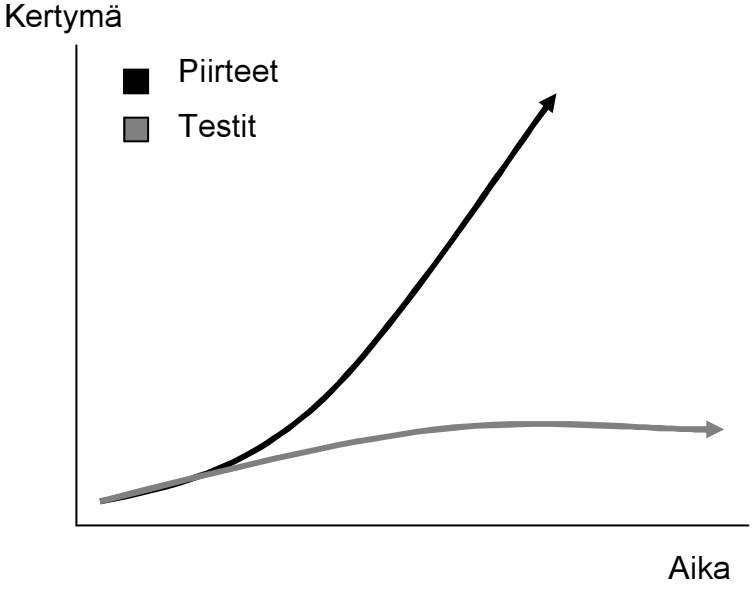
Ennen kuin voidaan automatisoida, täytyy toimiva manuaalinen testausjärjestelmä olla olemassa. Sen täytyy sisältää vähintään seuraavat piirteet: Ensinnäkin yksityiskohtaiset testitapaukset sekä odotetut tulokset, jotka perustuvat vaatimusmäärittelyihin ja suunnitteludokumentteihin. Toiseksi erillinen testausympäristö, jossa on testitietokanta, joka voidaan aina tallentaa uudelleen tietyssä vakiomuodossa siten, että testitapaukset voidaan suorittaa uudelleen, kun sovellukseen tulee muutoksia. [Zam98]

Manuaalisesti testin voi yleensä tehdä yhden kerran huokeammalla ja se vie vähemmän aikaa kuin saman testin tekeminen automaattisesti. Kannattaa siis pohtia seuraavaa: ''Jos testi automatisoitaisiin, mitä käsin tehtyjä testejä jäisi tekemättä? Kuinka monta virhettä jäisi huomaamatta ja kuinka pahoja virheitä nuo olisivat?'' Jos vastaus ensimmäiseen kysymykseen on alle viisi, voi tuollaisessa tapauksessa automatisoiminen olla kannattavaa. Tällöin sopivan automaation avulla saadaan parannettua testaamisen tarkkuutta ja siten ohjelman laatua. Jos taas ollaan testaamassa jotain nopeasti muuttuvaa tuotetta, jossa pelkästään käsin tutkimalla löydetään useita virheitä samassa ajassa, jonka uuden automatisoidun testauksen tekeminen veisi, on kannattavampaa käyttää manuaalista testausta. [Mar98]

Testauksen automatisointi on vaikeaa. Testausprojektit epäonnistuvat yhtä todennäköisesti tai todennäköisemmin kuin muutkin ohjelmistoprojektit, koska yritykset eivät panosta yhtä paljon testausvälineisiin kuin toimitettaviin tuotteisiin. Vieläkin ollaan vaiheessa, jossa ihaillaan sokeasti testauksen automatisointia, tunnistamatta sen vaaroja. Automatisointiin pyritän vain, koska tietokoneita pidetään nopeampina, halvempina ja luotettavampina kuin ihmisiä. [Bac99]

Apuvälineitä (työkaluja, apuohjelmia) automatisointiin on olemassa todella runsaasti, ainakin 600 kpl. Jokaiseen V-mallin vaiheeseen löytyy useita erilaisia välineitä [Poh02].

Linda Hayesin mielestä automatisoinnilla on saavutettava kolme hyötyä*: testin toistettavuus (repeatability)* monta kertaa samanlaisena, *laajennettavuus (leverage*) eli mahdollisuus suorittaa testejä, jotka ovat manuaalisesti mahdottomia, ja *kertymä* (*accumulation)* eli mahdollisuus suoriutua sovelluksen muutoksista vähemmällä määrällä testitapauksia, kuin manuaalisessa testauksessa (Kuva 2). Viimeinen etu saavutetaan, kun testitapaustietokanta suunnitellaan niin hyvin ylläpidettäväksi, että ohjelmiston piirteiden määrän lisääntyessä testien määrä ei lisäänny. [Hay95]



Kuva 2. Kertymä ajan suhteen [Hay95]

Eräs menetelmä testauksen automatisoinnissa on käyttää automatisoitua *skriptien* generointia simuloimaan aloittelevia käyttäjiä. *Testiskriptit* muodostuvat sarjasta käskyjä ja ohjeita siitä, mitä testaustyökalun tulisi tehdä testattavalle ohjelmalle, jotta testi tulisi suoritettua [Vir02]. Näin voidaan systeemin virheet löytää varhaisemmassa vaiheessa kuin vasta beta-testauksessa tai tuotannossa. Näissä vaiheissa löytyneet virheet ovat kalliita korjata ja hyvin turhauttavia varsinkin aloitteleville käyttäjille. [KaG96]

Automatisoinnin avulla on voitu suorittaa muutamassa minuutissa testejä, jotka ovat aikaisemmin vaatineet aikaa useita tunteja. *Nauhoitus/toisto* -menetelmällä voidaan samat testitapaukset toistaa kuinka monta kertaa tahansa paljon vähemmällä vaivalla kuin, jos ne

tehtäisiin joka kerta uudelleen. Parhaimmillaan automatisoitu testaus on pienentänyt testauksen kustannuksia 80 % [FeG99]. Epäonnistumisista ja onnistumisista kerrotaan luvussa 8.

Kun me automatisoimme testausta, me automatisoimme joitakin seuraavista aktiviteeteista: testauksen ehtojen tunnistaminen, testitapausten suunnittelu, testien rakentaminen, testien suorittaminen ja saatujen tulosten vertailu odotettuihin tuloksiin. Automatisointia varten kootaan yleensä ryhmä. Ryhmässä on oltava ehdottomasti kokemusta ohjelmoinnista, testauksesta ja automatisoinnista. Täytyy harkita tarkkaan, mitä ruvetaan automatisoimaan. Kuormitustestaukset, kuten myös regressio-testaus, ovat yleensä otollinen automatisoinnin kohde, koska niitä joudutaan toistamaan lähes samanlaisina useampia kertoja [FeG99]. Marko Jäntti toteaa gradussaan, että kuormitustestauksen työkalut ovat kalliita [Jän03].

Brian Marick kertoo vieneen kauan aikaa ennen kuin hän huomasi yrittävänsä automatisoida koko ajan liian paljon. Vain osa testeistä oli sellaisia, jotka kannatti ottaa mukaan automatisointiprosessiin. Kun on tehtävä päätös jonkin testin automatisoinnista, täytyy arvioida montako koodimuunnosta tämä testi sietää. Mikäli ei kovin monta, testin on oltava erittäin hyvä löytämään virheitä, ennen kuin sen automatisointi kannattaa. Koko ajan on hyvä pitää mielessä, että ihminen havaitsee sellaisia virheitä, jotka kone ohittaa. Toisaalta ihminen on konetta huonompi tulosten tulkinnassa. Jos virhe sijaitsee esimerkiksi luvun seitsemännessä desimaalissa, automatisoinnin työkalu löytää sen varmasti, mutta ihminen todennäköisesti ei. [Mar98]

Testauksen automatisoinnin kannattavuutta voidaan mitata myös niiden manuaalisten testien lukumäärällä, jotka automatisoinnin takia jäävät suorittamatta ja niiden virheiden lukumäärällä, jotka näin jäävät löytymättä. Testitapaushan on suunniteltu testaamaan jotakin tiettyä ohjelman piirrettä. Kun automatisoitu testaus löytää uudelleen suoritettaessa virheen, jolla ei näytä olevan mitään tekemistä alkuperäisten määritysten kanssa, on todennäköistä, että virheitä löytyy vielä lisää. Myös tämä ominaisuus mittaa automatisoidun testauksen tehokkuutta. [Mar98].

Suurimpia virheitä, mitä testauksen automatisoinnissa on tehty, on ollut se, kun on muodostettu testausryhmä kokemattomista ohjelmoijista tai pelkästään loppukäyttäjistä. On

hyvä, jos ryhmässä on eri alan asiantuntemusta, mutta ehdottomasti siinä täytyy olla kokemusta. Yrityksen kokenein ohjelmoija tai suunnittelija on oikea henkilö johtamaan ryhmää. Ennen automatisoinnin aloittamista, täytyy hanke perustella hyvin yritysjohdolle. Ei riitä, että johto antaa suostumuksensa työkalun hankintaan, vaan tarvitaan sen sitoutuminen hankkeeseen koko toteutusajaksi. Hyvät selvitykset valitun työkalun ominaisuuksista, arvioinnit sen soveltuvuudesta, tulevista kustannussäästöistä jne. auttavat asiaa. [FeG99]

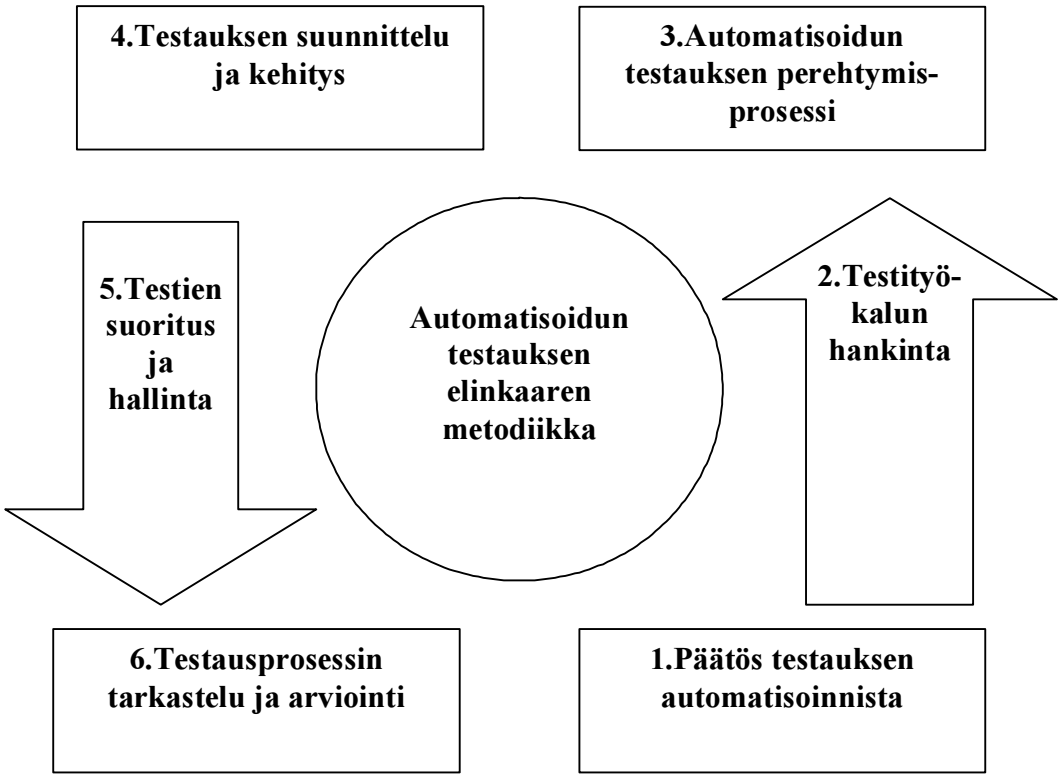
Useimmilla kokeneilla testaajilla on selvä käsitys siitä, mitä ohjelmiston testaaminen on. Kuitenkin tämä käsitys hämärtyy, kun ryhdytään automatisoimaan testausta. He eivät ymmärrä, että automatisointi on itsenäinen ohjelmiston kehitysprosessi, jossa on noudatettava tiettyä käytäntöä. Kuka sitten voi suorittaa automatisoinnin? Automatisoijalla täytyy olla hyvät taidot testauksesta ja ohjelmiston kehitystyöstä. Hänellä täytyy olla hyvä arviointikyky ja kosketus ohjelmointityöhön. Varo testaajia, jotka yrittävät automatisoida kaiken. Mitä sitten valitaan automatisoitavaksi? Kiinnitetään huomio sellaisiin testausvaiheisiin, jotka vaativat eniten manuaalista työtä. Näitä voidaan harkita automatisoitaviksi. Mikäli olet kokenut testaaja, sinulla on käsitys kohteista, joita tulisi testata, mikäli olisi riittävästi aikaa. Älä tuhlaa aikaasi tällaisten osien automatisointiyrityksiin, koska ne voivat joka tapauksessa jäädä kokonaan testaamatta. [Pet01b]

Robinsonin mukaan testauksen automatisoijan täytyy käyttää omaa järkeään mutta hyödyntää myös tietokoneen antamat mahdollisuudet. On turha tehdä käsin sellaista, minkä kone voi suorittaa paljon nopeammin. [Rob00]

TerMaat kertoo oppineensa, työskennellessään automatisoidun testien generoinnin parissa, periaatteen, jonka mukaan generointi voi epäonnistua, kun syötteiden välillä on liikaa riippuvuuksia, ja onnistua, kun voidaan saada aikaan yksi syöte isosta joukosta syötteitä [Ter01].

5.2 AUTOMATISOIDUN TESTAUKSEN ELINKAARI

Dustin, Rashka ja Paul esittävät seuraavan mallin *automatisoidun testauksen elinkaaren metodiikasta* (Kuva 3). Elinkaareen sisältyy kuusi eri vaihetta. Ensimmäiseksi tehdään päätös automatisoinnista. Tämän jälkeen suoritetaan työkalun valinta. Perehtymisprosessissa analysoidaan yrityksen nykyinen testauskäytäntö ja haetaan mahdollisia kehityskohteita. Neljännessä vaiheessa suunnitellaan testaus, sen dokumentointi ja edelleen kehittäminen. Testien suorittamisen jälkeen arvioidaan saavutetut tulokset. Tätä kehää voidaan kiertää aina uudelleen ja uudelleen. [DuR99]

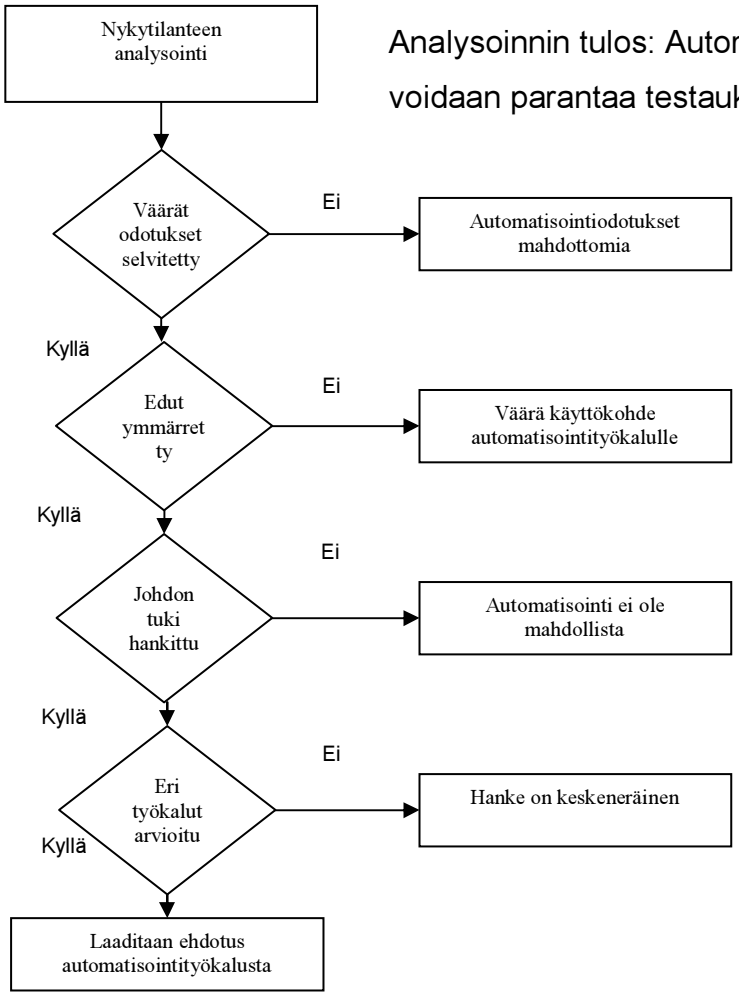


Kuva 3. Automatisoidun testauksen elinkaaren vaiheet [DuR99]

Seuraavissa kappaleissa (5.2.1 - 5.2.6) käydään läpi tarkemmalla tasolla kuvan 3 vaiheet 1 – 6.

5.2.1 Päätös automatisoinnista

Yrityksessä on selvitetty, että nykyinen testaussysteemi ei ole tarpeeksi tehokas. Analyysit ovat osoittaneet, että manuaalista testaussysteemiä on parannettava. Kehitystutkimuksen perusteella on päädytty kokeilemaan automatisoitua testausta. Kuvassa 4 esitetään päätösprosessi automatisoinnista vaiheittain. [DuR99]



Kuva 4. Testauksen automatisoinnin päätösprosessi [DuR99]

Päätöksissä automatisoida testausprosessi esiintyy paljon virheellisiä odotuksia. Monet uskovat, että automatisointityökalulla voidaan hoitaa kaikki tehtävät - testauksen suunnittelusta testauksen suoritukseen - ilman manuaalisia toimenpiteitä. Tällaista apuvälinettä ei ole vielä olemassa. Hyvin yleinen on myös luulo, että automatisointi vähentää heti testauksen työmäärää ja antaa kustannussäästöjä. Etuja saavutetaan vasta

pidemmällä aikavälillä. Automatisointia ei yleensä kannata ruveta edes suunnittelemaan kertaluonteisia testaustapahtumia varten. Manuaalinen testaus on tällöin käytännöllisempää ja halvempaa. Poikkeuksena tähän ovat testaukset, joita ei voida suorittaa manuaalisesti. [DuR99]

Automatisoidulla testauksella (yhdessä manuaalisen kanssa) saavutetaan monia etuja, mikäli se toteutetaan oikein. Tärkeimpiä etuja ovat: luotettavan järjestelmän syntyminen, testaustyön laadun paraneminen ja testauksen ja sen suunnittelun työmäärän minimointi. Luotettavuus lisääntyy vaatimusten määrittelystä aina parantuneeseen suorituskyky- ja kuormitustestaukseen saakka. [DuR99]

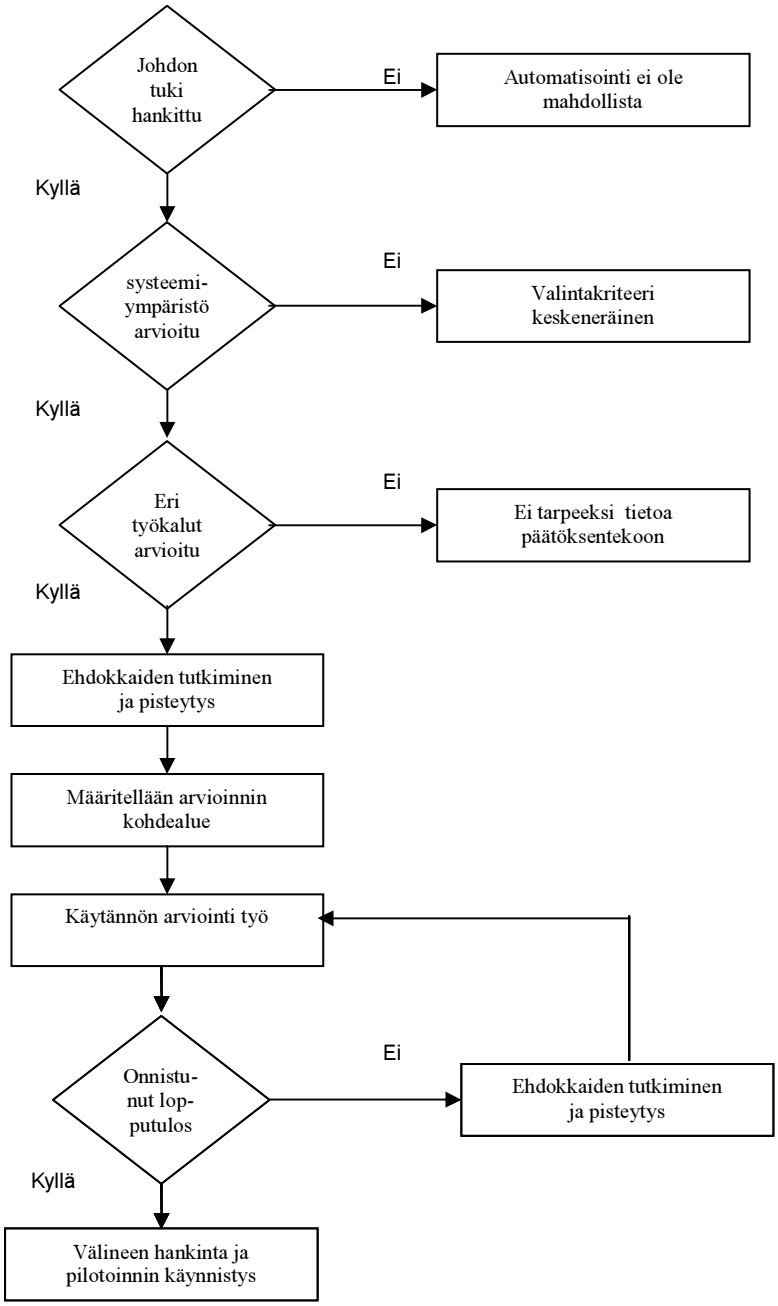
Suuri haaste testausryhmälle on siinä, että se pystyy esittelemään automatisointiprojektin parhaat puolet yritysjohdolle. Ryhmän täytyy myös korjata johdon odotukset uudesta systeemistä realistiselle tasolle. Tämä tapahtuu välittämällä asiallista tietoa projektista johtohenkilöille. Ellei johdon tukea ole hankittu, tai se on riittämätön, automatisointi on tuomittu epäonnistumaan. [DuR99]

Käytettävissä olevien testaustyökalujen läpikäynti on tärkeä askel. Minkälaiset työkalut tukevat parhaiten käynnistymässä olevaa automatisointia kehityksen elinkaaren eri vaiheissa? Testausryhmän täytyy arvioida kaupallisilla markkinoilla saatavissa olevat työkalut kiinnittäen huomiota siihen, kuinka ne tukevat yrityksen systeemien kehitysympäristöä. Päätöksentekoon vaikuttaa se, voidaanko määriteltyjä systeemin vaatimuksia todentaa käyttämällä yhtä tai useampaa työkalua. [DuR99]

5.2.2 Testityökalun hankinta

Työkalun valinta on aikaa vievä tehtävä vielä sen jälkeen, kun sovelluksen kehitysalustasta ja -välineistä on päätetty. Ihannetilanteessa työkaluksi voidaan valita se, mikä on ollut käytössä jo pilotointivaiheessa. Tässä tilanteessa täytyy kuitenkin ottaa huomioon kustannukset, jotka aiheutuvat työkalun hankinnasta ja koulutuksesta. Tämä johtaa siihen, että valitun työkalun täytyy olla yhteensopiva koko organisaation entisen *systeemiympäristön* kanssa. Systeemiympäristö käsittää alustat, ohjelmistot, laitteistot jne. Jotta

yhteensopivuus saavutettaisiin, testiryhmän täytyy suorittaa työkalun arviointi ja valinta strukturoidusti. Kuva 5 esittää korkealla tasolla testityökalun valintaprosessin. [DuR99]



Kuva 5. Automatisoidun työkalun valintaprosessi [DuR99]

Kun yritysjohto on sitoutunut hankkimaan vaaditut resurssit, testauksen johtaja käy läpi yrityksen systeemiympäristön. Tarkoituksena on varmistaa, että valittu apuväline on yhteensopiva mahdollisimman monen käyttöjärjestelmän, ohjelmointikielen tai muun

sellaisen piirteen kanssa, mitä yrityksessä on käytössä. Kaikki esiinnousseet kysymykset ja huolenaiheet kirjataan ylös tässä vaiheessa. [DuR99]

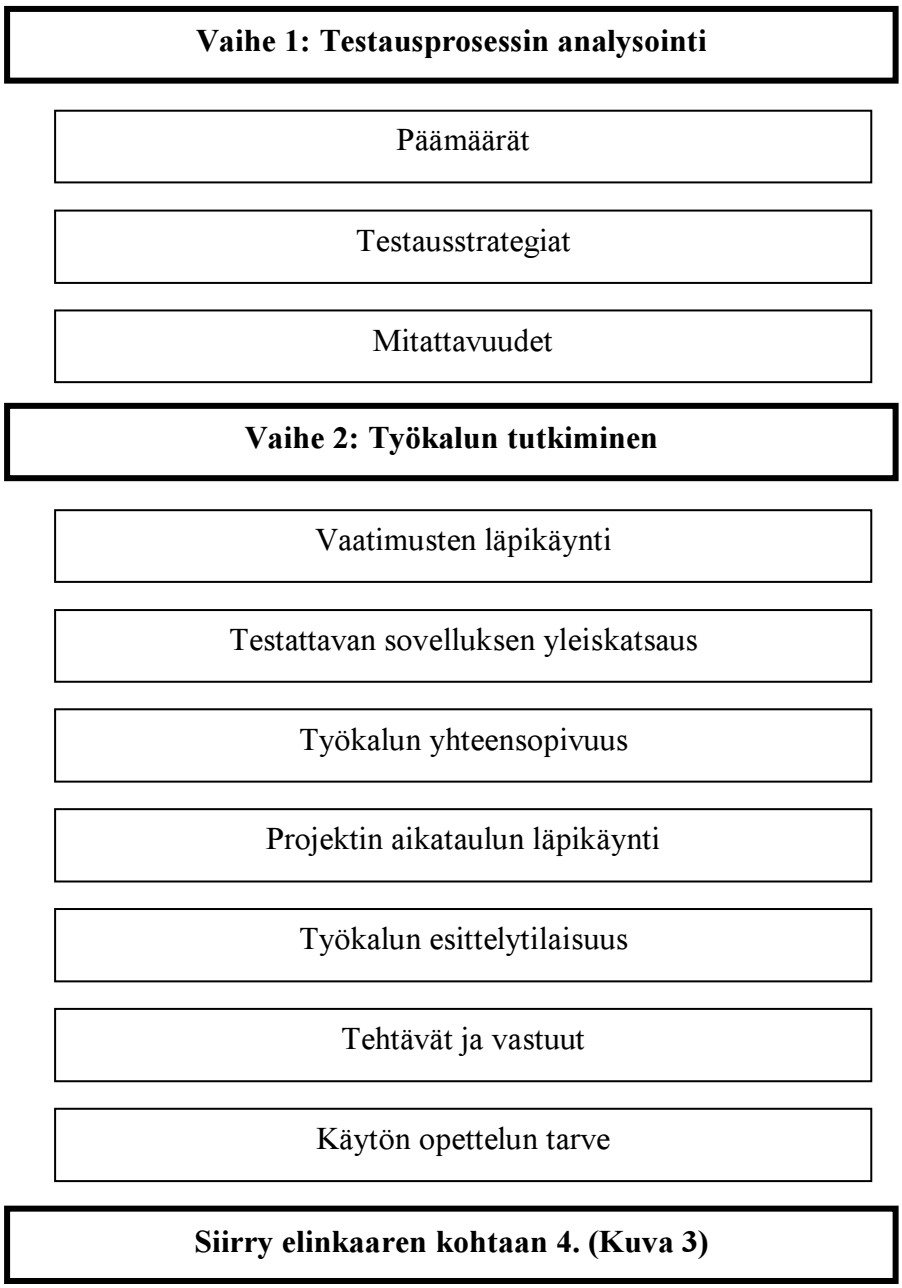
Näin saadusta informaatiosta voidaan laatia ''toivomuslista'', josta käy ilmi, minkälaiset piirteet tukevat yrityksen vaatimuksia. Testauksen johtajan on nyt kohdennettava nämä ominaisuudet yhteen tai useampaan ehdolla olevaan työkaluun ja *pisteytettävä* nämä. Pisteyttäminen tarkoittaa, että otetaan jokaisesta työkalusta samat ominaisuudet tarkasteltaviksi ja arvioidaan ominaisuudet antamalla niille pisteitä esimerkiksi yhdestä kymmeneen. Pisteet lasketaan yhteen ja korkeimman pistemäärän saanut on voittaja. Listalle on otettava mahdollisimman monia erilaisia toivomuksia. Työkaluista tutkittavia piirteitä ovat esimerkiksi: helppokäyttöisyys, työkalun muunnettavuus, tuetut alustat, monikäyttömahdollisuus, virheenjäljitys, työkalun toimivuus, tulostuskapasiteetti, suorituskyky, yhteensopivuus eri versioiden työkalujen kanssa, toimivuus testauksen suunnittelun ja hallinnan työkalujen kanssa, hinta ja myyjän kelpoisuus. Jokainen edellisistä piirteistä voidaan jakaa vielä alaosiin ja nämä pisteytetään kaavalla ''painokerroin x pisteet = lopullinen pistemäärä''. Esimerkiksi hinta arvioidaan seuraavasti: Onko hinta arvioitujen rajojen sisällä? Millainen lisenssisopimus on tarjolla? Onko hinta kilpailukykyinen? [DuR99]

Vaikka työkalun myyjä lähes aina vakuuttaa työkalun toimivan tilanteessa kuin tilanteessa, ei tähän pidä sokeasti uskoa. Kokemus on osoittanut, että usein löytyy erikoistapauksia, joissa väline ei toimi odotetulla tavalla. Kannattaa tehdä arviointisuunnitelma, jossa hahmotellaan yksityiskohtaisesti, kuinka työkalua testataan. Apuvälinettä olisi hyvä testata ensin rajatussa ympäristössä, esimerkiksi testilaboratoriossa. Arvioinnin kattavuus riippuu käytettävissä olevasta ajasta. [DuR99]

Systeemiympäristö ja tarjolla olevat työkalut ovat nyt tulleet tutuiksi. Niinpä voimme yksilöidä henkilöt, jotka suorittavat käytännön arvioinnin testaustyökalu(i)sta. Tässä vaiheessa kannattaa ottaa yhteyttä ainakin etukäteisarvioinnissa parhaalta tuntuneen työkalun myyjään ja pyytää esittelyä [DuR99]. Työkalun valinnasta lisää näkökulmia luvussa seitsemän.

5.2.3 Automatisoidun testauksen perehtymisprosessi

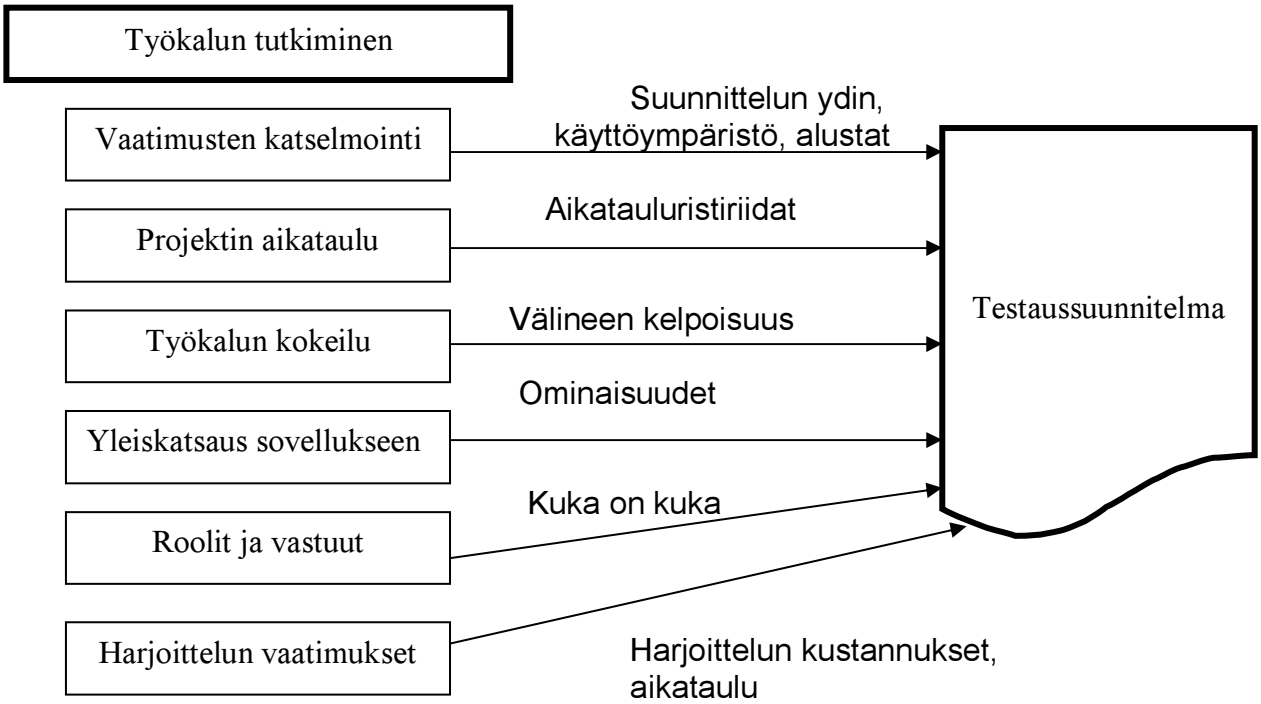
Prosessissa analysoidaan yrityksen nykyinen testauskäytäntö ja haetaan kehityskohteita. Tutustutaan valittuun työkaluun lähtemällä liikkeelle vaatimuksista, käyden läpi välineen kelvollisuus testattavaan sovellukseen ja kokeilemalla sitä käytännössä. Usein väline on otettu käyttöön suunnittelematta käytön tehokkuutta. Tämä on johtanut kertakäyttöisiin testiskripteihin, mikä ei ole tarkoitus. Toinen virhe on työkalun käyttöönotto liian myöhäisessä testauksen vaiheessa. Kuvassa 6 esitetään tutustumisprosessi.



Kuva 6. Testityökaluun tutustumisprosessi [DuR99]

Testausprosessin analysointivaiheessa testiryhmä todentaa, että työkalu on sopiva suurimpaan osaan projektin tarpeista. Aluksi tutustutaan nykyiseen testausprosessiin. Testausprosessin dokumentoinnin on oltava sellainen, että sitä on helppo esitellä muille. Ellei dokumentointia ole suoritettu, prosessin selittäminen muille ja sen toistettavuus on vaikeaa. Testausprosessin analysoinnin tarkoitus on määritellä päämäärät ja menetelmät, jotka ovat luontaisia prosessille. Nämä testauksen suunnittelun osat toimivat koko projektin testausohjelman kulmakivinä. Dokumentoinnin ja toteutuksen aika- ja kustannus-kysymykset nousevat useasti esille. Itse asiassa hyvin suunniteltu ja toteutettu prosessi maksaa itsensä takaisin moninkertaisesti parantuneena virheiden löytymisenä ja lyhentyneinä tuotteen kehitysaikana. Mitattavia elementtejä ovat esimerkiksi: suorituskyky, toistettavuus, jäljitettävyys ja vakaus. [DuR99]

Tutkimisvaiheessa testauksen johtaja tuntee testausprosessin ja näin hän voi päättää jatketaanko apuvälineen käytön harjoittelua. Erityisesti hän pyrkii todentamaan, että valittu työkalu todella toimii käyttöympäristössään ja vastaa vaatimusmäärittelyjä. Kuvassa 7 on esitetty testauksen työkalun tutkimisen eri vaiheet.

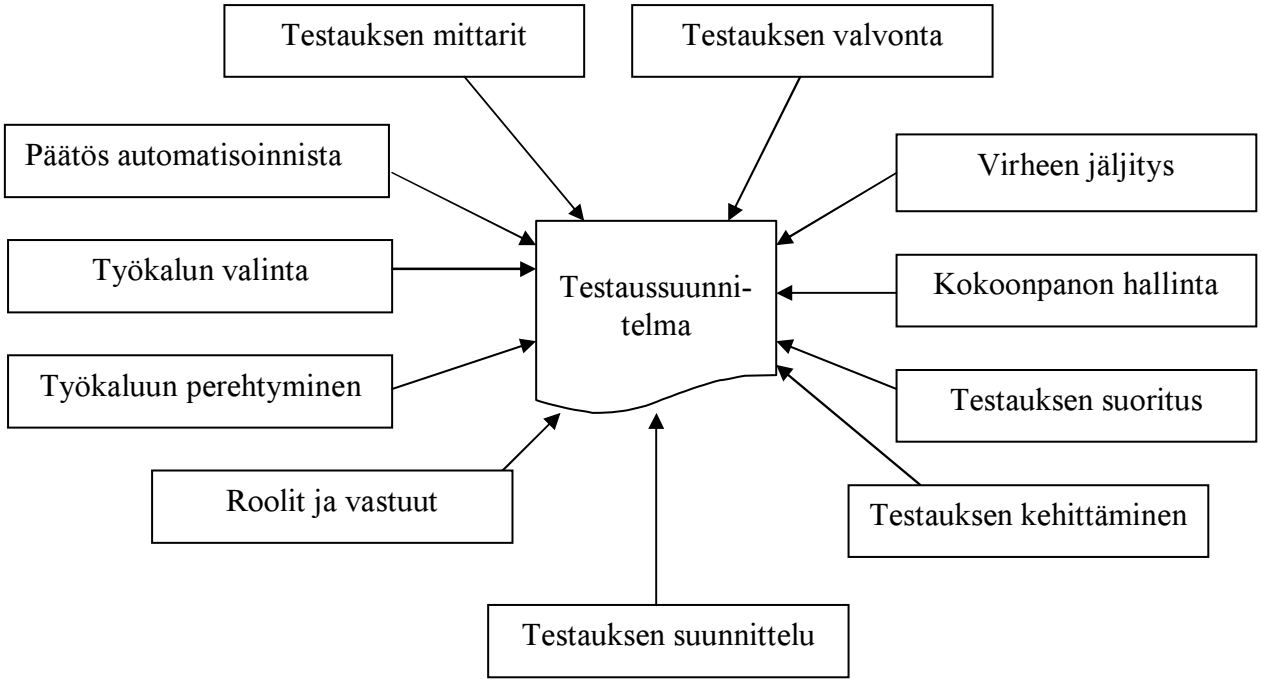


Kuva 7. Testityökaluun tutustumisprosessi – Testityökalun tutkiminen [DuR99]

Ennen kuin työkalu voidaan ottaa tehokkaaseen käyttöön projektissa, täytyy testauksen johtajan tuntea testattavan sovelluksen vaatimukset. Kun projektikohtaiset erityis-vaatimukset ovat selvillä, voi testausryhmä olla luottavainen, että tärkeimmät niistä tulevat huomioiduiksi. Työkaluun perehtyminen kannattaa suorittaa mahdollisimman aikaisessa vaiheessa. Tämä varmistaa sen, että testaajat ehtivät tutustua hyvin työkaluun ja sen uusiin piirteisiin. Sovelluksen teknisten piirteiden tutkimuksella selvitetään, millainen on käyttöliittymä tai kehitysympäristö. Millainen työkalu on yhteensopiva nykyisen alustan ja ohjelmiston kanssa? Testiryhmä tarkastelee myös sovellusta osa osalta pohtien, mitkä osat sopisivat hyvin automatisoituun testaukseen. Kaikkia vaatimuksia ei voida toteuttaa testauksen automatisoinnilla, eikä pelkästään yhdellä automatisoinnin työkalulla. Parasta on jakaa vaatimukset osajoukkoihin ja päättää, mikä automatisointityökalu on käyttökelpoisin mihinkin vaatimuksiin tai sovelluksen osiin. Kuka on kuka -osiossa selvitetään testausryhmän jäsenten tausta henkilöittäin, kartoitetaan kokemus työkaluista sekä se, ketkä ovat saaneet paljon koulutusta työkalujen käytöstä ja huomioidaan vielä erittäin laajan ja pitkäaikaisen kokemuksen omaavat henkilöt. Lopuksi arvioidaan koulutuksen/harjoittelun tarve nykytilanteessa. [DuR99]

5.2.4 Testauksen suunnittelu ja kehitys

Tehokas testauksen automatisoinnin työkalujen käyttö vaatii huomattavia investointeja testauksen suunnitteluun ja valmisteluun. Testaussuunnitelma sisältää paljon informaatiota – myös testauksen dokumentoinnin vaatimuksista projektissa. Suunnitelmassa täytyy myös olla tietoa jokaisesta elinkaaren eri vaiheesta (Kuva 3, sivu 27). Kuva 8 esittää nämä yhteydet testaussuunnitelmaan.



Kuva 8. Testaussuunnitelman syötteet ja sisältö [DuR99]

Tehokkaalla testausohjelmalla, jossa on mukana testauksen automatisointia, on oma kehittämisen elinkaarensa, jolla on oma toimintasuunnitelma, testauksen vaatimusten määrittely, analyysi, suunnittelu ja koodaus. Kuten ohjelmistosovelluksen kehittäminen, testauksen kehittäminenkin vaatii huolellista suunnittelua. Tämä vaihe sisältää sellaisten työmenetelmien luomisen, jotka ovat hallittavia, uudelleen käytettäviä, yksinkertaisia ja luotettavia. [DuR99]

5.2.5 Testien suoritus ja hallinta

Testit voidaan nyt suorittaa toimivan testausympäristön avulla siinä aikataulussa, kuin testaussuunnitelmassa on määritelty (Kuva 3, sivu 27). Moduuli-, integrointi-, järjestelmä- ja hyväksymistestaus sisältyvät tähän vaiheeseen. Yhdessä edellä mainitut muodostavat koko systeemin testauksen. Jotta testausprosessin laatua voidaan valvoa, testausryhmän täytyy kerätä ja analysoida erilaisia mittareita. Löydettyjen virheiden määrä suhteessa suoritettuihin testeihin on eräs tapa mitata testauksen tehokkuutta. Mikäli virheiden määrä on kovin korkea, joudutaan arvioimaan testausaikataulua tai testaussuunnitelmaa uudelleen. Testaaminen voidaan lopettaa, kun testiryhmä toteaa, että testaus-suunnitelmassa määritellyt hyväksymiskriteerit on saavutettu [DuR99]. Loputtomat muutokset tuotteen vaatimusmäärittelyissä ja muissa dokumenteissa ovat eräitä testauksen hallinnan vaikeimpia alueita [Len01].

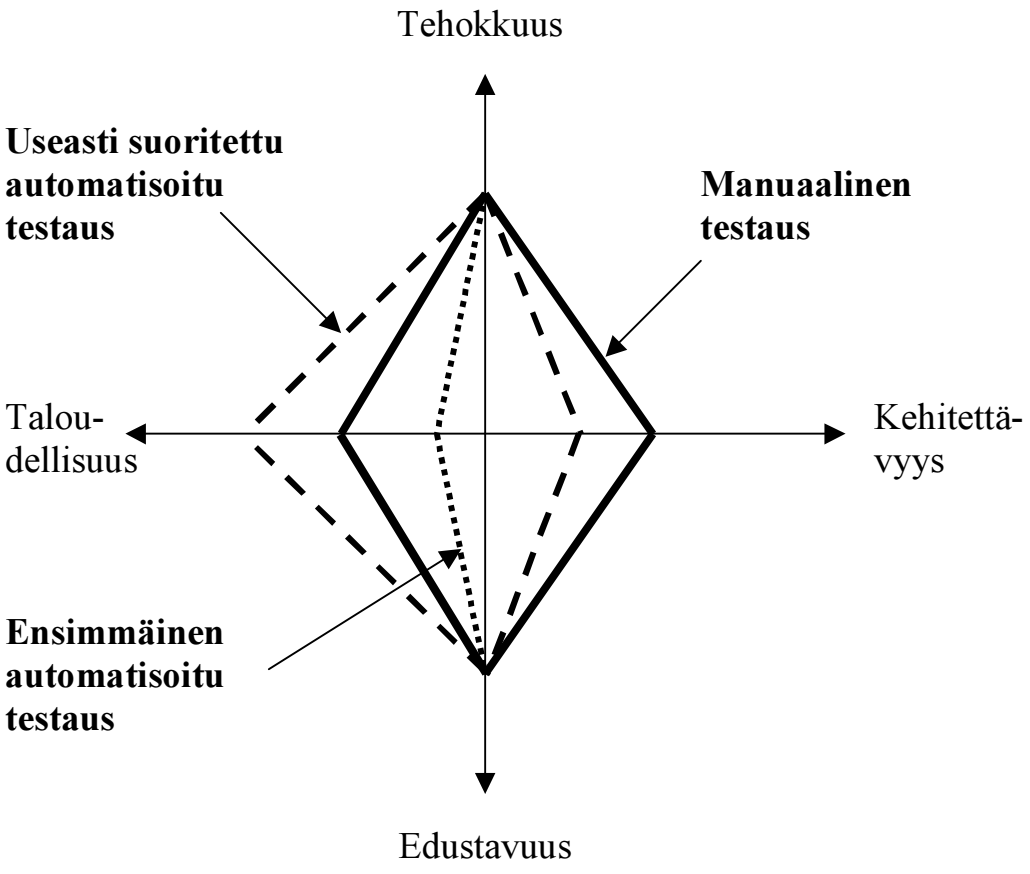
5.2.6 Testausprosessin tarkastelu ja arviointi

Testausryhmän tulee arvioida, kuinka testausohjelman suoritus onnistui, voidakseen päättää, mitä parannuksia voidaan tehdä seuraavalla testikierroksella tai seuraavassa projektissa. Arviointi on elinkaaren (Kuva 3, sivu 27) viimeinen vaihe. Elinkaari on kuitenkin syklinen ja voidaan suorittaa uudestaan. Arvioinnissa päähuomio kohdistetaan prosessin aikana mitattuihin arvoihin. Nämä osoittavat, kuinka hyvin alkuperäiset suunnitelmat ovat toteutuneet. On tärkeää, että testiryhmä dokumentoi huolellisesti kaikki hyvin ja etenkin huonosti menneet toiminnot. Dokumentointia on suoritettava koko elinkaaren ajan, eikä vain lopussa. [DuR99]

5.3 TESTITAPAUKSEN ARVIOINTI

Testitapauksen ”hyvyys” voidaan kuvata tarkastelemalla Keviatin kaavion neljää attribuuttia (Kuva 9) [FeG99]. Yhtenäinen viiva on manuaalisesti ja katkoviiva auto-maattisesti suoritettu testitapaus. Mitä pidempi kunkin attribuutin viiva on, sitä suurempi yhtyvien viivojen muodostama alue ja sitä parempi testitapaus. Attribuutit ovat: taloudellisuus, tehokkuus, kehitettävyys ja edustavuus. Tärkein näistä on tehokkuus

virheiden löytämisessä. Edustava testitapaus testaa useampaa kuin yhtä asiaa. Tehokkuus ja edustavuus pysyvät samana, olipa testi manuaalinen tai automaattinen. Kaksi muuta – taloudellisuus ja kehitettävyys – vaikuttavat kustannuksiin. Kuinka paljon testitapauksen suorittaminen maksaa, ja kuinka paljon ylläpitoa testitapaus vaatii ohjelmiston muutoksen jälkeen? Kuvasta näkee, että ensimmäisellä suorituskerralla automatisoitu testaus on kalliimpi kuin manuaalinen, ja sen kehitettävyys on huono. Mitä useammin testaus suoritetaan, sitä taloudellisemmaksi automatisointi tulee, ja kehitettävyys paranee. Ensimmäisellä suorituskerralla automatisointi ei ole kovin taloudellinen, eikä kehityskelpoinen, vaikka on vaatinut paljon vaivannäköä. Useita kertoja suoritettuna se tulee paljon taloudellisemmaksi kuin vastaava testaus manuaalisesti. [FeG99]



Kuva 9. Keviatin kaavio [FeG99]

Testitapausten laatuun kannattaa kiinnittää aivan erityistä huomiota. Ne suunnitellaan yksilöllisiksi. Mielellään käytetään testitapausten alijoukkoja [FeG99]. Ekvivalenssiluokitus on eräs ratkaisu tähän ongelmaan. Jaetaan testitapaukset osajoukkoihin siten että samaan joukkoon tulevat samankaltaiset tapaukset. Tällöin ei tarvita kaikkia alijoukon tapauksia vaan koko luokka saadaan testattua muutamalla kattavalla tapauksella [Paa00].

Verrattuna Fewsterin edellä esittämiin attribuutteihin [FeG99], Kaner määrittelee hyvällä testitapauksella olevan seuraavia ominaisuuksia: Se löytää suhteellisen luotettavasti ohjelmassa olevia virheitä. Mietitään tilanteita, joissa ohjelma voisi kaatua ja sen jälkeen yritetään keksiä, kuinka tällainen tapaus paljastetaan. Testitapaus ei saa olla tarpeeton – on turhaa hakea samaa virhetilannetta kahdella eri testitapauksella. Jokin testitapaus toisten samankaltaisten joukossa voi olla parempi kuin muut, joten valitaan se. Testausaikaa voidaan lyhentää yhdistämällä samaan testitapaukseen useita eri piirteitä. Tässä pitää olla varovainen, ettei synny liian monimutkaista ja hallitsematonta testitapausta. [KaF99]

5.4 AUTOMATISOINNIN EDUT

Fewsterin mukaan automatisoinnilla saavutetaan huomattavia etuja. Jo olemassa olevia testejä voidaan ajaa uudella ohjelmaversiolla (regressiotestaus). Etu on ilmeinen varsinkin ympäristössä, jossa useita ohjelmia päivitetään säännöllisesti. Kun testit ovat olemassa, voidaan ne suorittaa muutaman minuutin manuaalisella työllä. Enemmän testejä voidaan ajaa entistä useammin. Automatisoinnin selvä etu on kyky ajaa testejä pienemmässä ajassa ja näin mahdollistaa niiden ajaminen useammin. Tämä johtaa suurempaan luotettavuuteen systeemissä. On mahdollista suorittaa testejä, jotka olisivat liian vaikeita toteuttaa manuaalisesti. On mahdotonta testata 200 käyttäjän on-line systeemiä todellisessa käytössä, mutta sitä on helppo simuloida käyttäen automaattista testausta. Resursseja voidaan siis käyttää paremmin. Automatisoimalla toisarvoiset ja yksitoikkoiset tehtävät, esimerkiksi samanlaisilla syötteillä toistettavat testit, saavutetaan suurempi tarkkuus ja parannetaan työintoa. Näin vapautetaan taitavat testaajat suunnittelemaan parempia testitapauksia. Testit ovat johdonmukaisia ja toistettavissa, koska automaattisesti toistetut testit toistetaan tarkasti samanlaisina joka kerta. Tämä antaa

testaukseen luotettavuutta, jota ei voida saada aikaan manuaalisesti. Testejä voidaan käyttää uudelleen, jolloin työpanos päätettäessä, mitä testataan, suunniteltaessa testit ja rakennettaessa testit, voidaan jakaa monelle testien suorituskerralle. Kun testaus on automatisoitu, sitä voidaan suorittaa monta kertaa nopeammin ja useammin kuin manuaalisesti. Tämä voi lyhentää oletettua testausaikaa. Luotettavuus lisääntyy, kun tiedetään, että laaja joukko automatisoituja testejä on suoritettu onnistuneesti. Voidaan olla luottavaisempia, ettei enää tule epämiellyttäviä yllätyksiä ohjelmistoa julkistettaessa. [FeG99]

5.5 AUTOMATISOINNIN YLEISET ONGELMAT

Vastaavasti Fewster näkee automatisoinnissa myös ongelmia. Yrityksillä on epärealistisia odotuksia. Jokaisesta uudesta teknisestä ratkaisusta uskotaan, että se ratkaisee kaikki nykyiset ongelmat. Testauksen apuvälineet eivät ole poikkeus tästä olettamuksesta. Jos testaajien kokemus on huono, testit on järjestetty huonosti, dokumentointia on vähän tai ei ollenkaan ja testit eivät ole kovin hyviä löytämään virheitä, testauksen automatisointi ei tuo ratkaisua ongelmiin. Luulo, että automatisoitu testaus löytää paljon uusia virheitä on väärä. Suurin osa virheistä löydetään ensimmäisellä testauskerralla. Tämän jälkeen löytyy enimmäkseen virheitä, jotka ovat aiheutuneet ohjelman korjaamisesta, eli automatisoinnin hyöty on vähäinen. Vaikka testaus menisi läpi ilman, että löytyy yhtään virhettä, ei saa syntyä väärää turvallisuuden tuntua. Ei ole mitään takeita, että virheitä ei olisi olemassa. Testit voivat olla epätäydellisiä tai sisältävät itse virheitä. Kun ohjelmistoa on muutettu, on usein välttämätöntä päivittää joitakin testejä tai jopa kaikki testit, jotta ne voidaan suorittaa uudelleen. Automatisoitujen testien ylläpidon vaikeus on lopettanut monta testauksen automatisointia heti alkuunsa. Kaupalliset testauksen apuvälineet ovat myyntifirmojen myymiä tuotteita. On kaksinkertainen pettymys huomata, että testausvälinettä ei ole hyvin testattu, eli se itsessään sisältää virheitä. Valitettavasti tällaista kuitenkin tapahtuu. Testauksen automatisointi ei ole yksinkertainen tehtävä. Johdon tuki on välttämätön. Automatisointi täytyy sovittaa organisaation nykyiseen kulttuuriin. Aikaa täytyy varata välineen valintaan, käytön harjoitteluun, tukemiseen, tutkimiseen ja opettelemiseen organisaation sisällä. On saatava selville mikä työkalu toimii parhaiten ja soveltuu parhaiten yrityksen omaan tarpeeseen. [FeG99]

5.6 ONNISTUNUT TESTAUKSEN AUTOMATISOINTISTRATEGIA

Smale esittää artikkelissaan automatisoinnin olevan jatkuva prosessi ja luettelee toimenpiteitä automatisoinnin onnistumiseksi: Tee yksityiskohtainen testaussuunnitelma ennen kuin teet mitään muuta. Ole selvillä automatisoinnin strategiasta ja ota huomioon johdon ja työntekijöiden toiveet. Järjestä testauksen hallinta siten, että kaikilla testaajilla on käytössään samat standardit ja, että kaikki testit ovat ylläpidettäviä ja helposti ymmärrettäviä. Vähennä ylläpitoa – kirjoita yleisiä funktioita ja moduuleita ja käytä niitä kaikkialla. Kirjoita tarkoituksenmukaisia testipäiväkirjoja ja raportoi kaikista oikeista ja virheellisistä tuloksista. Testien täytyy olla suoritettavissa yksinään, kykeneviä havaitsemaan virhetilanteet ja toipumaan niistä. Tee testeistäsi yhteensopivia eri kielien, alustojen ja kokoonpanojen kanssa. Sisällytä testeihisi jotakin satunnaisuutta. Aloita varovasti testeillä, jotka on suoritettu päivittäin, esimerkiksi rakenteen oikeellisuustesteillä. Perusta eteneminen onnistumiseen. Mittaa automatisoinnin tehokkuus löydettyjen virheiden määrällä. Käytä automatisointia kuormitustestaukseen. Suorita testejä tuotteellesi kunnes löytyy virheitä. [Sma99]

Pettichordin mukaan automatisoinnin onnistumiseen vaikuttaa useita eri tekijöitä: Ennen kuin aletaan automatisoida, yritetään vielä kehittää nykyistä testausprosessia. Ennen kaikkea yritetään tehdä tuotteesta helpommin testattava. Määritellään tarkasti vaatimukset, mitä testataan ja mitä voidaan automatisoida. Testauksen automatisoinnin toteuttaminen on osoitettava mahdolliseksi. Aikaa menee aina enemmän, mitä tehtävään haluttaisiin käyttää. Sitoutuminen automatisointiin vaatii tukea monelta eri taholta yrityksessä. Tuotteen testattavuusominaisuudet on otettava huomioon jo suunnit-teluvaiheessa. Kiinnitetään huomiota siihen, että testausmenetelmän suorituskyky ja muut piirteet pysyvät vakaina. Automatisoidusta testauksesta on kehitettävä sellainen menetelmä, että sen pystyy suorittamaan joku muukin, kuin sen rakentaja. Tämä edellyttää hyvää dokumentointia ja virheilmoituksia epätavallisissa tilanteissa. Vaikka automatisointi onnistuisi hyvin, asettaa se myös uusia haasteita. Testausprosessi monimutkaistuu välttämättä automatisoinnin myötä. Henkilöstön on opittava tulkitsemaan automatisoinnin löytämiä virhetilanteita oikein. Ellei näin tapahdu, voidaan luulla, että automatisointi on aiheuttanut nämä virhetilanteet. [Pet01a]

Kaner on tehnyt myös muutamia hyviä havaintoja automatisoinnin onnistumisesta. Hän suosittelee suunnittelemaan testit ensin hyvin, ennen kuin päätetään, mitä automatisoidaan. Tämä ehkäisee sitä, että automatisoidaan testejä, jotka on helppo automatisoida, mutta jotka löytävät huonosti virheitä. Hän suosittelee myös suunnittelemaan automatisoidut testit erillään manuaalisista. Suuri osa automatisoidun testauksen tehokkuudesta saavutetaan sillä, että pistetään tietokone tekemään se, mitä ihminen ei voi tehdä. Opetuksena Kaner haluaa vielä sanoa, että automatisoimalla ilman hyvää suunnittelua saadaan aikaan paljon toimintaa, mutta vähän tuottoa. Toisaalta, ellei ole hyvää tietämystä automatisoinnin mahdollisuuksista, voivat parhaat automatisoinnin ominaisuudet jäädä käyttämättä. [KaB02]

5.7 AUTOMATISOIDUN TESTAUKSEN YLLÄPIDETTÄVYYS

5.7.1 Ongelmat

Automatisointi aiheuttaa lisäkustannuksia, koska merkittävien virheiden löytyminen siirtyy lähemmäs systeemin valmistumista. Tämä johtuu siitä, että automatisointia ei ole saatu valmiiksi tarpeeksi ajoissa. Voi myös käydä niin, että ainoat automatisoidut testit ovat sellaisia, jotka ohjelma on jo läpäissyt. Tunnetusti testiryhmän on alkuvaiheessa mukavampi suunnitella helposti suoritettavia testejä. [Kan97]

Uudet ohjelmistoversiot tarvitsevat uusia testejä. Ne myös useasti muuttavat jo olemassa olevia testejä, koska ohjelmien toiminnot ovat muuttuneet. Edelleen järjestelmän muuttuessa monet testeistä käyvät tarpeettomiksi joko siksi, että toiminnot, joita varten ne olivat, on poistettu tai uudet testit ovat korvanneet ne. Kaikki tämä vaikuttaa ylläpito-kustannuksiin. Automatisoidun testauksen ylläpitokustannukset ovat paljon korkeammat kuin manuaalisen, koska testattaessa manuaalisesti on mahdollista toteuttaa muutoksia välittömästi. Automatisoidussa testauksessa kaikki osaset on määriteltävä tavalla tai toisella. Mitään ei voida jättää työkalun huolehdittavaksi, koska sillä ei ole älyä. Ohjelmistojen ylläpito on itsestään selvyys kaikille, mutta usein ajatellaan liian vähän testien ylläpidettävyyttä. Testauspaketti sisältäen testauksen dokumentoinnin, testitapaukset ja odotetut tulokset on niin arvokas kokonaisuus, että sitä täytyisi pitää yhtä tärkeänä kuin ohjelmistoa, jota sillä testataan. [FeG99]

5.7.2 Ratkaisuehdotuksia

On hyvä tiedostaa, että taloudellinen hyöty saadaan vasta seuraavasta ohjelmaversiosta. On mahdotonta kehittää laajoja testausohjelmistoja ilman huolellista suunnittelua. Testausohjelmisto voi olla suurempi kuin testattava ohjelmisto [Kan97]. *Taulukkopohjaisen (data-driven)* arkkitehtuurin käyttö tarkoittaa sitä, että testitapaukset on kirjoitettu taulukkomuotoon ja voidaan suorittaa siitä. Tämä on helpompaa huonommin ohjelmointia tunteville henkilöille [Pet01]. Käytettäessä *kehyspohjaista (framework-based)* arkkitehtuuria hyödynnetään kirjastossa olevia funktioita. Skriptien kirjoittajat käyttävät niitä aivan kuin ne olisivat työkalun ohjelmointikielen käskyjä. Manuaaliseen testaukseen tarkoitettu testiskripti on testauksen työjärjestyksen ohjeistus [FeG99]. Testiryhmässä täytyy olla myös kokemusta. Johdon on panostettava testaukseen ja ymmärrettävä, että automatisointi vie aikaa. Jatkuvasti pohdittavia asioita ovat: Tehtiinkö tässä automatisoinnissa kaikki oikein? Olisiko jokin toinen apuväline ollut parempi? Oliko automatisointi ylipäätään järkevää? [Kan97]

Kanerin mukaan automatisoidun regressiotestauksen ylläpidettävyyteen vaikuttavia seikkoja ovat muun muassa: Johdon odotusten suhteuttaminen siihen, kuinka nopeasti automatisoinnista saadaan hyötyä. Sen asian havaitseminen, että testauksen automatisoinnin kehittäminen on ohjelmiston kehittämistä. Taulukko- ja kehyspohjaisen arkkitehtuurin käyttäminen. Henkilöstöresurssien huomioiminen. Toisenlaisen automatisoinnin harkitseminen. [Kan97]

Testipaketin ylläpidettävyyttä helpottaa se, ettei sen anneta kasvaa liian suureksi. Tässä olisi löydettävä kultainen keskitie. Testitapausten lisääminen helpottaa ja varmentaa testausta, mutta vaikeuttaa ylläpitoa. Huolellisella suunnittelulla on tässä tärkeä merkitys. Testitapausten määrälle voidaan asettaa tietty yläraja, mutta tämä voi estää hyvienkin uusien testien mukaan pääsyn, koska ne pitäisi pystyä todistamaan paremmiksi kuin jo mukana olevat. On myös hyvä ajoittain käydä läpi olemassa olevat testitapaukset. Ovatko kaikki tarpeellisia? Onko mahdollisesti päällekkäisyyksiä? [FeG99]

Tekstimuotoinen tieto on usein joustavimmin ja helpoimmin käsiteltävissä. Se on myös siirrettävissä eri alustojen ja järjestelmäkokoonpanojen välillä. Mikäli on mahdollista tallentaa testitapaukset ja tulokset esimerkiksi ASCII muodossa, se kannattaa

ehdottomasti tehdä. Tiedon muuntamiseen kuluva aika on usein paljon vähäisempi kuin kustannukset, jotka aiheutuvat erikoismuotojen ylläpidosta. Muunto on helppoa joko editoimalla tekstitietoa tai kehittämällä sitä varten oma muunto-ohjelma. [FeG99]

Testitapausten suoritusaika kannattaa pyrkiä pitämään mahdollisimman lyhyenä. Oletetaan, että testitapauksen suoritus kestää esimerkiksi 30 minuuttia. Ensimmäisellä kerralla suoritus epäonnistuu 5 minuutin kuluttua. Toisella yrityksellä keskeytys tapahtuu 10 minuutin kohdalla. Kolmannella kerralla päästään 15 minuuttiin asti. Tässä vaiheessa on käytetty jo koko 30 minuuttia, mitä testauksen piti kestää, emmekä ole päässeet kuin puoleen väliin testin kokonaissuoritusta. Testitapaukset pitäisi suunnitella sellaisiksi, että niillä on virheen tunnistamiskyky. Pelkkä ilmoitus, että testi ei mennyt läpi, ei riitä. Olisi hyvä, jos työkalu olisi sellainen, että testitapaukset olisi helppo ajaa uudelleen virheenjäljitystä hyväksi käyttäen. [FeG99]

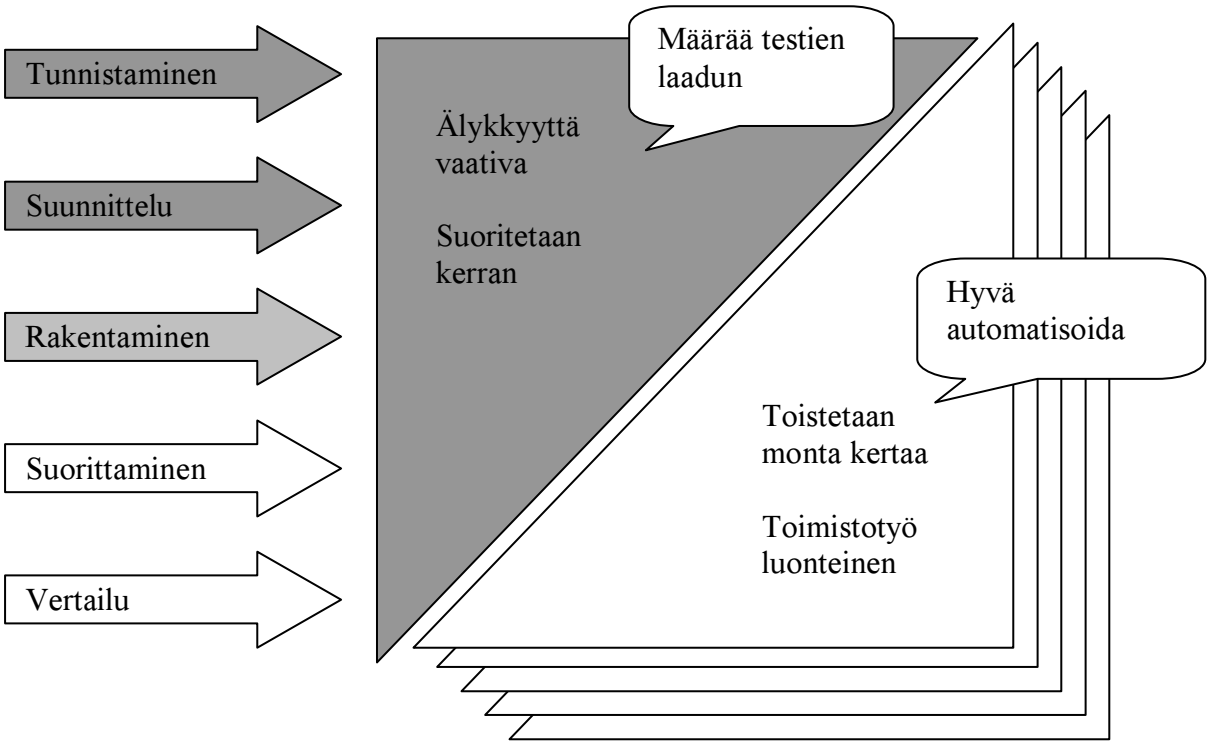
Ehkä tärkein asia ylläpidon keventämisessä on kirjoittaa testitapaukset jollakin abstraktilla testauskielellä, jolloin testitapauksissa ainoastaan osoitetaan käsiteltävät kohteet, ja sen jälkeen käytetään tulkkia ja testiajuria varsinaiseen testin suoritukseen [Pet01b]. Testaus kannattaa aloittaa lyhyillä testeillä ja pienellä testitapausten määrällä. Kun ohjelmisto näyttää toimivan oikein näillä, voidaan testitapausten määrää lisätä [FeG99].

Otetaan heti alussa käyttöön yhteinen käytäntö skriptien ja tiedostojen nimeämiseen. Tämä helpottaa huomattavasti niiden löytämistä myöhemmin. Mikäli nimeäminen tapahtuu vapaasti, se johtaa ennen pitkää kaaokseen. Testitapaukset ja skriptit täytyy dokumentoida. Dokumentointitasoksi riittää lyhyt selvitys testitapauksen tarkoituksesta ja selitys siitä, mitä skripti tekee. Tärkeää on dokumentoinnin laatu, ei sen määrä. [FeG99]

5.8 TESTAUKSEN SUUNNITTELUN AUTOMATISOINTI

5.8.1 Toiminnot, jotka on mahdollista automatisoida

Fewsterin mukaan kuvassa 10 kaksi ensimmäistä testaustoimintoa, testauksen ehtojen tunnistaminen ja testitapausten suunnittelu, ovat pääasiallisesti älykkyyttä vaativia luonteeltaan. Kaksi viimeistä toimintoa, testitapausten suorittaminen ja testin tulosten vertailu, ovat enemmän toimistotyöluonteisia (clerical). Älykkyyttä vaativat toiminnot määräävät testitapausten laadun. Toimistotyöt ovat erityisen työvaltaisia ja sen takia sopivat hyvin automatisoitaviksi. On järkevää kysyä, kannattaako testitapauksen suunnittelu automatisoida, koska se suoritetaan yleensä vain kerran. [FeG99]



Kuva 10. Testausprosessi jakaantuu viiteen erilliseen toimintoon [FeG99]

Kuitenkin on suunniteltu useita apuvälineitä, jotka on tarkoitettu suunnittelun automatisointiin. Esimerkkeinä mainittakoon muun muassa Caliber-RBT, Inferno, RadSTAR, SoftTest ja Validator/Reg. Caliber-RBT on suunniteltu vaatimusmäärittelyihin pohjautuvaan testaukseen. Se antaa projektihenkilöille mahdollisuuden aikaisemmassa vaiheessa tapahtuvaan vaatimusmäärittelyjen hiomiseen ja vahvistamiseen. Inferno on testitapausten

generointiväline. Sen avulla saadaan toistettavia testitapauksia ja näin kustannukset jakaantuvat tasaisemmin koko sovelluksen elinajalle. RadSTAR on yhdistelmä testien suunnittelumetodologiasta ja automaattisesta koneesta, joka suorittaa suunnitellut testit. SoftTest on funktionaalinen testitapausten suunnitteluväline mille tahansa sovellukselle, kielelle tai alustalle. Validator/Reg pystyy generoimaan jopa tuhat testitapausta minuutissa ja se poimii automaattisesti testien määrittelyt projektin malleista. [Poh02]

Testauksen suoritus- ja vertailutoiminnot suoritetaan monta kertaa, kun taas toiminnot, jotka määrittelevät testauksen ehdot ja suunnittelevat testitapaukset suoritetaan vain kerran. Usein toistetut ovat hyviä automatisointikohteita. Kaikki edellä mainitut toiminnot voidaan tehdä myös manuaalisesti, kuten vuosikaudet on tehty, mutta automatisoimalla jälkimmäiset, eli suorituksen ja vertailun testit saavutetaan kaikista suurin hyöty. [FeG99]

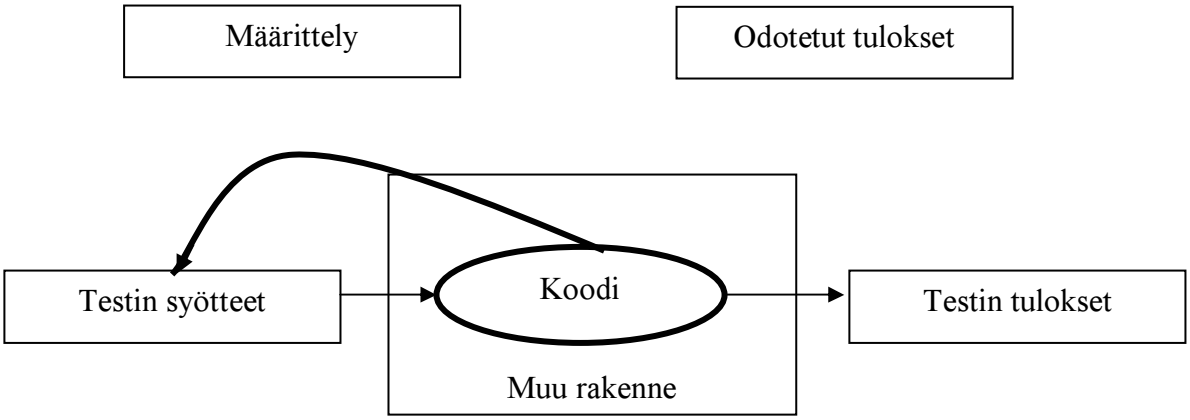
5.8.2 Testitapauksen suunnittelun automatisointi

Voidaanko testitapauksen suunnittelun toiminnot automatisoida? Tähän kysymykseen vastaus on: kyllä voidaan. On paljon tapauksia, joissa ainakin osa testitapauksen suunnittelusta voidaan automatisoida. Tällaisia apuvälineitä ovat esimerkiksi niin sanotut testitapausten generaattorit. Ne ovat hyödyllisiä monissa yhteyksissä, mutta eivät koskaan voi korvata ihmisen rakentamia testitapauksia. Eräs ongelma työkalun käytössä on, että ne voivat generoida todella suuren määrän testejä, joista osa on tarpeettomia. Jotkut välineet osaavat tosin minimoida testitapausten määrää käyttäjän antamien kriteerien mukaan. Ongelmaksi kuitenkin jää, että testitapauksia tulee silti liikaa [FeG99]. Monet valmistajat lupaavat apuvälineensä minimoivan testitapausten määrän, ja samalla maksimoivan niiden kattavuuden. Tarkastellaan seuraavaksi kolmea erilaista tapaa automatisoituun testitapausten generointiin.

5.8.2.1 Koodiin pohjautuva testitapausten generointi

Tämä menetelmä generoi testitapaukset tutkimalla koodin rakennetta. Polku ohjelman läpi koostuu lohkoista, jotka määräytyvät haarautumisten mukaan jokaisessa ehtokohdassa. Näin voidaan helposti tuottaa automaattisesti jokaisen polkulohkon tarvitsemat loogiset ehdot. Tämä on hyödyllinen piirre kattavuuden mittauksessa. Lähestymistapa kylläkin generoi testitapauksia, mutta testauksessa meidän pitäisi pystyä myös vertaamaan

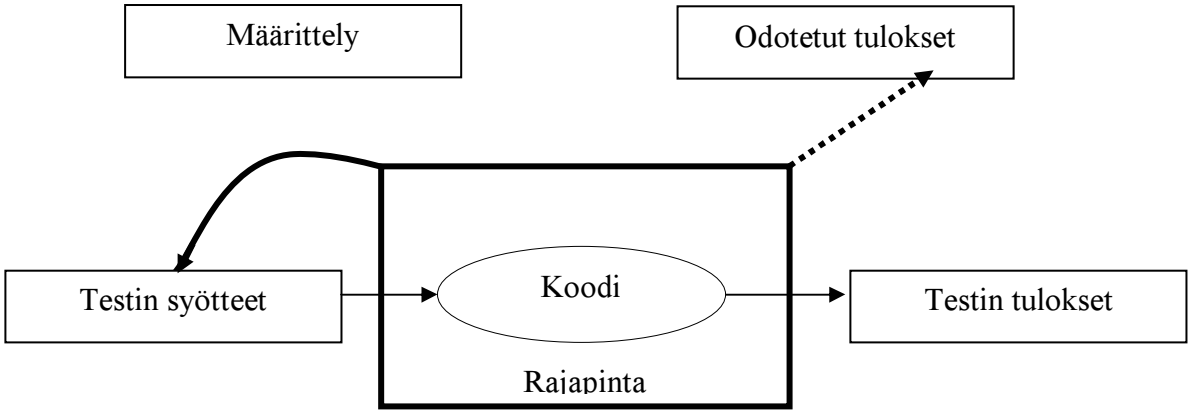
saatuja tuloksia odotettuihin. Menetelmää käyttäessämme meillä ei ole mitään tietoa siitä, saimmeko oikeita tuloksia. Toinen ongelma on, että menetelmä testaa vain olemassa olevaa koodia eikä huomaa ollenkaan puuttuuko osa koodista (Kuva 11). [FeG99]



Kuva 11. Koodiin perustuva testisyötteen generointi [FeG99]

5.8.2.2 Rajapintaan perustuva testitapausten generointi

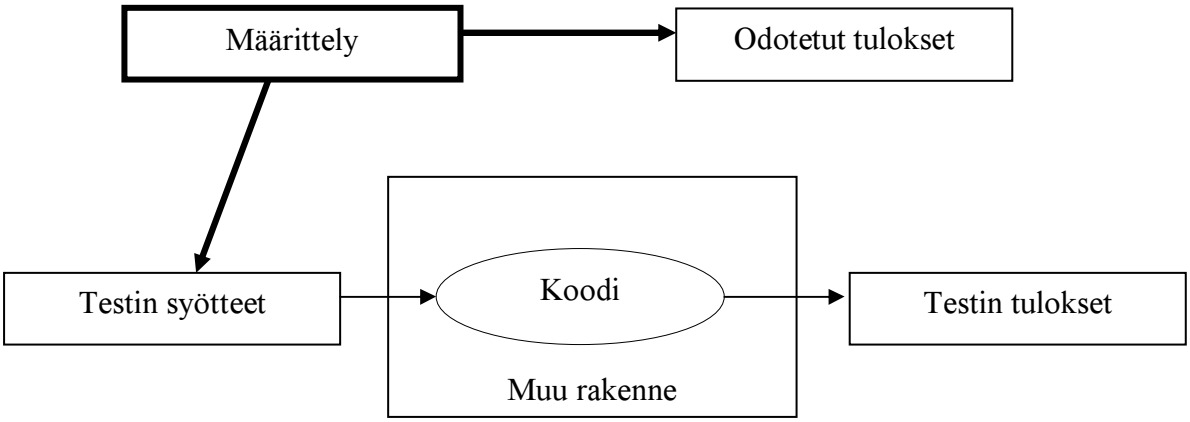
Rajapintaan perustuva testigenerointi (Kuva 12) soveltuu hyvin rajapintojen, kuten graafisten käyttöliittymien ja verkkosovellusten testaukseen. Mikäli näyttö koostuu useista valikoista, painikkeista ja tarkistusruuduista, apuväline voi generoida testitapauksia, jotka käyvät tarkistamassa jokaisen. Esimerkiksi käydään tarkistamassa toimiiko ohje joka kentässä. On myös mahdollisuus määritellä virheen tyyppi. Siten meillä on myös osittainen yhteys odotettuihin tuloksiin, jotta voimme verrata niitä saatuihin. [FeG99]



Kuva 12. Rajapintaan perustuva testisyötteen ja testitapausten generointi [FeG99]

5.8.2.3 Määrittelypohjainen testitapausten generointi

Menetelmällä (Kuva 13) voidaan generoida sekä testitapaukset että odotetut tulokset. Määrittelyjen on oltava sellaisessa muodossa, että apuväline voi tulkita niitä. Ne voivat olla työsääntöjä. Toisaalta mukana voi olla myös teknistä tietoa, esimerkiksi tiloja ja niiden muutoksia. Tietokantojen testaus on myös mahdollista. Saavutettu etu on, että sovellukset todella näyttävät, mitä ohjelman pitäisi tehdä, eikä vain sen, mitä se tekee. [FeG99]



Kuva 13. Määrittelyyn perustuva testitapausten generointi [FeG99]

5.8.2.4 Testitapausten generointi olio-ohjelmoinnin luokille

Ilmentymämuuttujien esiintyminen ja niiden vaikutus luokassa määritellyn metodin suorittamiseen ovat suurimmat ongelmat luokkien testauksessa. Metodi voi antaa oikean tai väärän lopputuloksen riippuen siitä, mitkä kutsuvan ohjelman muuttujien arvot ovat, kun metodia kutsutaan. Luokkia testattaessa onkin tärkeää huomata, että jokainen metodi suoritetaan kutsujan kaikissa eri tiloissa. Perinteiset lähestymistavat ovat luokan tilan tarkastelu ja tietovirta-analyysi. Muodostamme siis yhtenäisen kehyksen, joka käsittää tietovirta-analyysin, symbolisen suorituksen ja automatisoidun päättelyn generoitaessa metodijonoja luokkien strukturoituun testaukseen. Tietovirta-analyysissä käsitellään ns. määrittely-/käyttöpareja. Parien analyysi kohdentuu koko luokkaan tarkasteltaessa pelkästään ilmentymämuuttujia. Näin ollen vertailtavat lauseet voivat kuulua eri metodeihin. Symbolisessa suorituksessa tunnistetaan ehdot, jotka liittyvät polkujen suoritukseen ja muuttujien määrittelyihin. Jokaiselle polulle jokaisessa metodissa haetaan polun suorittamiseen liittyvät ehdot, syöttö- ja tulostusarvojen suhteet ja muuttujien määrittelyt tällä polulla. Automatisointia hyödynnettäessä tunnistetaan yhtenäiset metodien

kutsujonot, jotka suorittavat ensimmäisessä kohdassa löydetyt määrittely-/käyttöparit. Näin ollen jokainen metodijono johtaa ilmentymämuuttujan käyttöön kyseisen muuttujan määrittelyn kautta, lähtien liikkeelle alkutilasta. Tällaiset jonot saadaan aikaiseksi asteittain päättelemällä metodin esi- ja jälkiehdoista, jotka ovat toisen vaiheen tuloste. Tässä esitetyn menetelmän suurin haitta on käytettyjen tekniikoiden monimutkaisuus – erityisesti symbolisessa suorituksessa ja automaattisessa päättelyssä. [BuO00]

5.9 TESTAUKSEN SUORITUKSEN AUTOMATISOINTI

Päivitysohjelmat ja muut samankaltaiset ohjelmat, jotka eivät ole välittömässä vuoro-vaikutuksessa käyttäjän kanssa, ovat helpoimmat automatisoida. Yksinkertainen komentotiedosto voi hoitaa ohjelmien käynnistämisen ja tulosten vertailun. Näin tärkeimmät asiat saadaan tehdyksi. Testauksen automatisointi ainoastaan testien suorituksessa ei kuitenkaan hyödytä meitä kovin paljon. Manuaalinen tarkistus on virhealtista, aikaa vievää ja ikävystyttävää. Kuitenkin monet ihmiset päätyvät työkaluun, joka tuottaa pinoittain paperia käsin tarkistettavaksi. Tällöin ei voida puhua testauksen automatisoinnista. Päämääränä täytyy olla, että apuväline pystyy myös vertailemaan lopputuloksia. [FeG99]

5.9.1 Mitä manuaalisesta testausprosessista automatisoidaan?

Peruslähtökohtana voidaan pitää sitä, että testin automatisoiminen vie vähintään viisi kertaa enemmän aikaa kuin sen suorittaminen manuaalisesti. Monet eri asiat vaikuttavat siihen, miten hankala prosessi automatisointi on. Ensinnäkin se, minkälainen työkalu on käytössä, ovatko siinä kaikki tarpeelliset piirteet, voidaanko tehdä lisämäärityksiä testin aikana jne. Toiseksi asiaan vaikuttaa se, millainen lähestymistapa automatisointiin on. On olemassa paljon erilaisia menetelmiä suorittaa testauksen automatisointi, vaikka monille ihmisille se on vain manuaalisen toiminnan nauhoittamista ja nauhoituksen toistamista. Erittäin tärkeää on muistaa, että automatisointi on paljon enemmän kuin pelkkää nauhoittamista. Voidaan jopa sanoa, että nauhoittaminen ei ole automatisointia lainkaan. Testauksen automatisoijan kokemustaso on erittäin tärkeä. Työkalujen käytön ammattilaiset suoriutuvat tehtävästä nopeammin ja vähemmillä virheillä. He osaavat myös välttää toteutuksia, joissa automatisointi ei toimi ja keskittyä sellaisiin, joissa se onnistuu.

Mikäli testaus ympäristö on sellainen, että se on vaikea toistaa uudelleen testauksessa, on automatisointi vieläkin vaikeampaa. Esimerkiksi sulautetut ja reaaliaikaiset järjestelmät ovat ongelmallisia, koska ne voivat vaatia erikoistyökaluja tai piirteitä, joita ei kaupallisilla markkinoilla ole saatavilla. Sellaisten sovellusten, jotka eivät ole käyttäjän kanssa vuorovaikutuksessa, testauksen automatisointi on paljon helpompaa edellyttäen, että niiden toimintaympäristö voidaan toistaa. [FeG99]

5.9.2 Testaussyötteen automatisointi

Oletamme, että kaupallinen testauksen automatisoinnin apuväline on käytössä. Työkalu nauhoittaa kaikki testaajan toiminnot ja kirjoittaa niistä skriptin, jota apuväline voi lukea. Työkalu voi nyt toistaa testaajan toiminnot kuinka monta kertaa tahansa. Skripti on tiedostona, johon päästään käsiksi ja jota voidaan myös muuttaa ja kehittää ohjelmoijan toimesta. Skriptit voidaan tehdä myös manuaalisesti mutta, koska ne ovat usein kirjoitettuja jollakin formaalilla kielellä, jota väline voi ymmärtää, niiden tekeminen on kokeneen ohjelmoijan työtä. [FeG99]

Automatisoimalla sellaisten testiskriptien generointi, joita aloitteleva testaaja tekee, voidaan säästää paljon henkilötyöaikaa ja saadaan muodostettua beta-testauksen perusta. Virheet löydetään jopa aikaisemmin kuin perinteisessä beta-testauksessa ja kustannussäästöt ovat huomattavat. [KaG96]

5.9.3 Edut automatisoitaessa ainoastaan syöte

Eräs etu manuaalisten testien nauhoittamisesta on se, että näin saadaan suhteellisen nopeasti rakennettua toistettava testimenetelmä. Suunnittelua ei juurikaan tarvita ja hyödyt ovat saatavissa lähes välittömästi. Kuitenkin tilapäisten testausten nauhoittaminen on kuin ohjelmoisimme ilman minkäänlaisia määrityksiä. Lopputuote tekee jotkin asiat oikein, jotkin asiat väärin, joitakin toimintoja se ei tee ollenkaan ja tekee jotain sellaista, mitä ei ole odotettu. [FeG99]

Toinen nauhoitus/toisto-menetelmän piirre on, että se pystyy tuottamaan automaattisesti dokumentaatiota siitä, mitkä testit on suoritettu. Syntyy jäljitysketju, joka kertoo tarkasti,

mitä on tehty. Tämä ei ole välttämättä hyvä pohja tehokkaalle automatisoidulle testaukselle, mutta näin päästään alkuun. Mikäli vaikkapa käyttäjät tulevat kokeilemaan systeemiä ja onnistuvat kaatamaan sen, voidaan nähdä virheeseen johtanut tapahtumaketju. Jäljitysketju voi myös sisältää tietoa ajan käytöstä ja suorituskyvystä, joista manuaalinen testaus ei kerro mitään. [FeG99]

Mikäli tehdään sama ylläpitomuutos suureen määrään tiedostoja, nauhoitus on käyttökelpoinen. Automatisoitu toiminta on paljon nopeammin suoritettavissa kuin manuaalinen ja sen vuoksi syöte on helposti toistettavissa. [FeG99]

5.9.4 Epäkohdat manuaalisten testien nauhoituksessa

Nauhoitusmenetelmän haitat tulevat ilmi, kun sitä käytetään kaiken aikaa. Ensimmäisellä kerralla menetelmä näyttää hyvin tehokkaalta, mutta se ei ole silti hyvä lähtökohta pitkään käytettävälle, tulokselliselle testauksen automatisoinnille. Nauhoitetut skriptit eivät ole tarpeeksi selkeitä ymmärtää, jos joku muu kuin skriptin ohjelmoija tarvitsee niitä myöhemmin. Automatisoidun testauksen arvo on sen uudelleenkäytettävyydessä. Pelkkä nauhoitus ei kerro mitään siitä, mitä on testattu ja miksi. Se on myös hyvin tiiviisti sidottu niihin erityspiirteisiin, joita testattiin. Kun ohjelmisto muuttuu, alkuperäinen skripti ei enää välttämättä toimi oikein, jos jokin sen testaamista piirteistä on muuttunut. Suosituksena voi sanoa, että jos parhaillaan suoritat tilapäistä manuaalista testausta, on parempi kehittää itse testausprosessia kuin ruveta automatisoimaan sitä. [FeG99]

5.10 TESTAUKSEN TULOSTEN VERTAILUN AUTOMATISOINTI

On epäkäytännöllistä, jos kaikkia testin tuloksia verrataan odotettuihin tuloksiin - varsinkin tuloksia, jotka eivät ole muuttuneet. Tavallisesti tarkastellaan vain merkittävimpiä piirteitä. Näyttöruudulle tulevat tulosteet voidaan verrata jo testin suorituksen aikana. Tätä kutsutaan *dynaamiseksi vertailuksi (dynamic comparison).* Muut tulosteet, kuten tiedostoihin ja tietokantoihin menevät, voidaan verrata vasta testin suorituksen päätyttyä. Tätä kutsutaan *suorituksen jälkeiseksi vertailuksi (post-execution comparison)*. Yleensä automatisoidussa testauksessa tarvitaan näiden kahden yhdistelmää. Vertailu on eräs testauksen automatisoitavimmista piirteistä. Laajojen numeeristen listojen, ruudun

näyttöjen tai minkä tahansa tietojen vertaaminen soveltuu paremmin koneelle kuin ihmiselle. [FeG99]

5.10.1 Dynaaminen vertailu

Testiskriptissä olevat ohjeet kertovat työkalulle, mitä pitää verrata, milloin ja mihin verrataan. Vertailuohjeita ei saa lisätä skripteihin manuaalisesti. Monet työkalut antavat mahdollisuuden keskeyttää nauhoitus, niin että nähdään, mitä sillä kohtaa tarkastetaan, ja voidaan tallentaa nykyinen versio odotetuksi lopputulokseksi. Työkalu lisää skripteihin automaattisesti tarkistusohjeet uusinta-ajoille samoihin kohtiin, joita edellisellä kerralla käsiteltiin. [FeG99]

Dynaaminen vertailu antaa mahdollisuuden ohjelmoida jonkin asteista älykkyyttä testitapauksiin – saaden ne toimimaan eri tavalla riippuen kulloisestakin tuloksesta. Esimerkiksi virhetilanteen ilmetessä voidaan kyseisen testitapauksen suorittaminen lopettaa. Jatkaminen olisi turhaa ajan hukkaa. Tällainen älyn lisääminen tekee testiskripteistä myös suoritukseltaan joustavampia. [FeG99]

Kaiken edellä olevan perusteella tuntuisi siltä, että dynaamista vertailua kannatta käyttää niin paljon kuin mahdollista. Asia ei ole aivan näin yksiselitteinen. Käskyjen ja ohjeiden lisääminen skriptien sisälle tekee niistä monimutkaisempia. Niinpä sellaisten testitapausten, jotka joutuvat useaan dynaamiseen vertailuun, luominen ja kirjoittaminen oikein on vaikeampaa ja ylläpitäminen kalliimpaa kuin yksinkertaisempien testitapausten. [FeG99]

5.10.2 Suorituksen jälkeinen vertailu

Testauksen suorituksen työkalut eivät yleensä tue vertailua suoraan, vaan on käytettävä eri työkaluja. Tämän vuoksi automatisointi vaatii enemmän työtä. Vuonna 1999 oli vain muutamia apuvälineitä saatavilla kaupallisilta markkinoilta [FeG99]. Tilanne ei juurikaan ollut parantunut vuonna 2002. Silloin oli hieman alle kymmenen työkalua tarjolla [Poh02].

Joskus välineet myydään pakettina siten, että dynaamisen ja suorituksen jälkeisen vertailun työkaluja ei voi käyttää erillään [FeG99].

Vertailu suorituksen jälkeen antaa enemmän mahdollisuuksia valita toimintasääntöjä ja määrittää vertailun laajuus. Sen sijaan, että vertaisimme aina kaikkia saatuja tuloksia odotettuihin, kannattaa tulokset jakaa kahteen tai useampaan ryhmään. Toista ryhmää ja sen aliryhmiä ryhdytään tutkimaan vain, mikäli ensimmäisestä saatiin hyväksytty lopputulos. Esimerkiksi ensiksi tarkastetaan tulokset yleisluontoisesti ja, jos tulos on virheellinen, ei ole mitään mieltä jatkaa tarkastelua yksityiskohtaisemmalle tasolle. [FeG99]

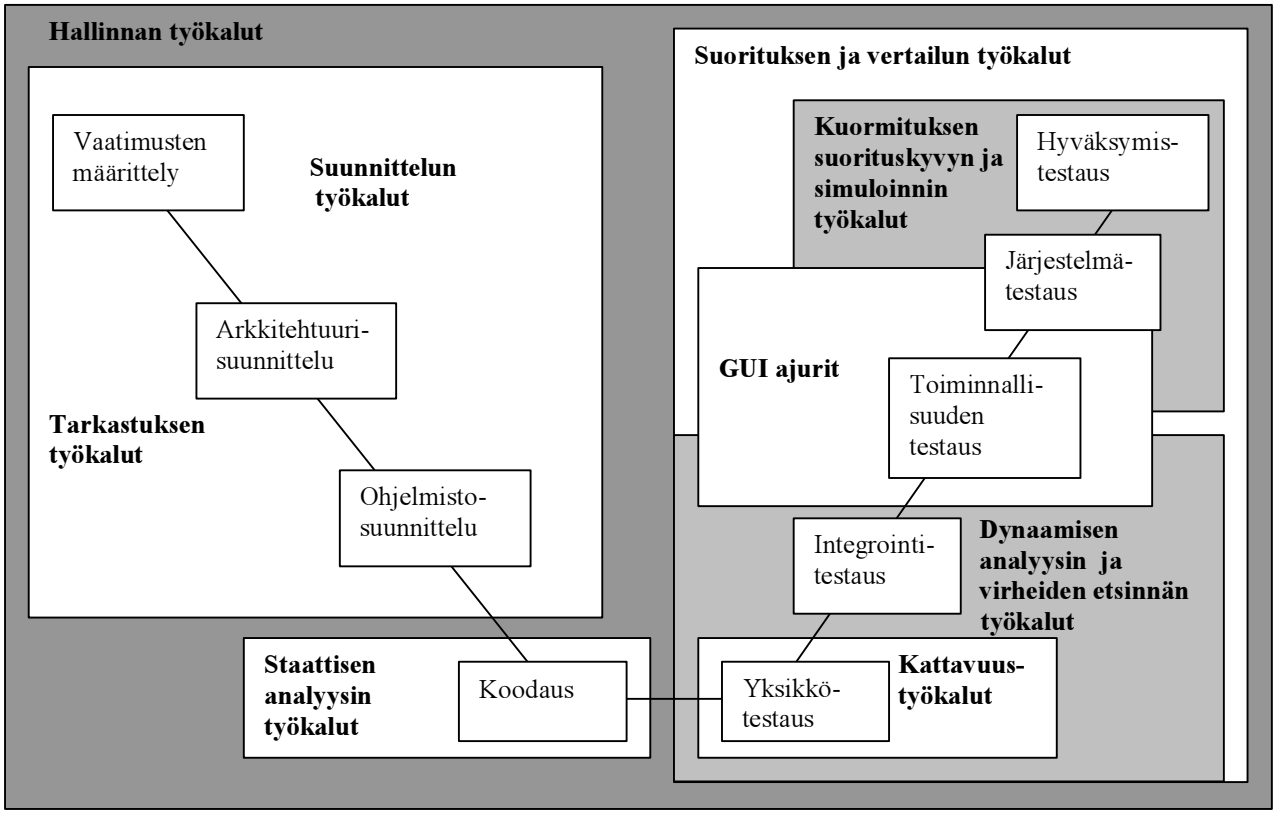
6 TESTAUKSEN APUVÄLINEET

Testauksen apuvälineet voidaan ryhmitellä esimerkiksi seuraaviin joukkoihin niiden käyttökohteen mukaan: suunnittelu, graafinen käyttöliittymä, kuormitus ja suorituskyky, hallinta, toteutus, arviointi, staattinen analyysi. Muita työkaluryhmiä ovat virheen jäljitykseen, verkkosivujen testaukseen soveltuvat ja sekalaiset. Tässä tutkielmassa ei näitä tarkastella sen lähemmin. [Poh02]

6.1 APUVÄLINEIDEN KÄYTETTÄVYYS SYSTEEMIN ELINKAAREN AIKANA

Apuvälineitä voidaan käyttää testaukseen jokaisella ohjelmiston kehityksen elinkaaren tasolla. Kuva 14 (sivu 54) havainnollistaa apuvälineiden eri tyypit ja niiden sijainnin elinkaaren eri tasoilla Fewsterin [FeG99] ja Tervosen [Ter00] mukaan.

Rajat eri ryhmien välillä ovat epäselvät. Jokin työkalu voi omata piirteitä useastakin eri ryhmästä. Hallinnan työkalut ovat käytettävissä koko ohjelmiston kehityksen elinkaaren aikana. Suunnittelun ja tarkastuksen välineitä käytetään vaatimusmäärittelyssä, arkkitehtuurisuunnittelussa ja ohjelmistosuunnittelussa. Suorituksen ja vertailun välineitä voidaan käyttää koko V-mallin oikeassa puoliskossa. Dynaamisen analyysin työkalut kuuluvat yksikkö-, integrointi- ja toiminnallisuustestaukseen (musta laatikko). Ne arvioivat järjestelmän toimintaa ajon aikana. Kattavuustyökalut ovat erikoisesti moduulitestaukseen suunniteltuja. Systeemi- ja hyväksymistestauksessa käytetään kuormitus- ja suorituskykytyökaluja. GUI (Graphical User Interface) -ajureilla on monia muiden ryhmien piirteitä mutta ne ovat selkeästi oma ryhmänsä. Ne ovat käyttökelpoisia koko testauksen toteutus- ja arviointialueella. [Poh02]



Kuva 14. Apuvälineiden sijainti ohjelmistokehityksen elinkaaressa [Poh02]

6.2 TESTAUKSEN SUUNNITTELUTYÖKALUT

Suunnittelun työkalut auttavat määrittämään, millaisia testejä on suoritettava. Ne mahdollistavat testitapausten generoinnin ja ovat käytettävissä (Kuva 14) vaatimusten-määrittely-, arkkitehtuurisuunnittelu- ja ohjelmistosuunnittelutasoilla. Esimerkkinä näistä apuvälineistä on Caliber-RBT, joka on vaatimusmäärittelyihin pohjautuva testitapausten suunnitteluväline. Tavoitteena on saada minimaalisella määrällä testitapauksia maksimaalinen toiminnallinen kattavuus. [Poh02]

Chillarege on sitä mieltä, että ainakin 30 % testauksesta on testitapausten korjaamista. Tämä on hyvin manuaalista työtä ja erittäin sopivaa automatisoitavaksi. Kuitenkaan hän ei pidä nykyisiä automatisoinnin työkaluja vielä tarpeeksi kehittyneinä. Ne tuottavat esimerkiksi liian suuria testitapausjoukkoja. [Chi99]

6.3 GRAAFISET KÄYTTÖLIITTYMÄT (GUI – AJURIT)

Tähän ryhmään kuuluvat työkalut mahdollistavat automatisoidun käyttöliittymien testauksen. Näillä apuvälineillä on mahdollista mm. asiakas/palvelin -järjestelmien kuormitustestaus ja nauhoitus/toisto menetelmät. Yksi välineistä on TestRunner, joka on tarkoitettu järjestelmätestaukseen kaikilla mahdollisilla graafisten käyttöliittymien ja I/O-kanavien kombinaatioilla. Työkaluilla on piirteitä monista eri ryhmistä ja ne sijoittuvat V-mallissa (Kuva 14, sivu 54) oikeanpuoleiseen haaraan. [Poh02]

Automatisoitaessa GUI-testausta on työkalun valintaan suhtauduttava samalla vakavuudella kuin ohjelmiston kehitystyöhön yleensäkin. Hyvällä automatisoinnin työkalulla on monia samoja piirteitä kuin hyvällä kehitysympäristöllä. Sen täytyy olla ylläpidettävä. Siirryttäessä versiosta toiseen on selvittävä mahdollisimman vähillä päivityksillä. Välineen on oltava luotettava. Sen tulee antaa oikeita tuloksia, jotka kohdentuvat tarkasti testattavaan ohjelmaan. Välineen on oltava toimintavarma. Yllättävän virhetilanteen sattuessa välineen tulee toipua itsenäisesti. [Hen99b]

Työkalun vaihtaminen voi koitua kalliiksi. Alussa kannattaa sijoittaa mieluummin vähemmän rahaa kuin liian paljon. Mikäli työkalu osoittautui kelvottomaksi, ovat tappiot näin pienemmät. [Hen99b].

6.4 KUORMITUS- JA SUORITUSKYKYTYÖKALUT

Kuormitus- ja suorituskykytyökalut auttavat toiminnallisuus-, järjestelmä- ja hyväksymistestausvaiheissa (Kuva 14, sivu 54). Ne ovat usein myös GUI-ajureita. Eräs näistä apuväli-

neistä on JavaLoad, joka on tarkoitettu Java ohjelmien kuormitustestaukseen. Se antaa raportteja käyttäjistä, testidatasta, keskimääräisistä vastausajoista jne. yhden istunnon aikana ja vertailee eri istuntojen tuloksia. [Poh02]

6.5 TESTAUKSEN HALLINNAN TYÖKALUT

Hallinnan työkalujen tarkoituksena on automatisoida testauksen suunnittelua, analysointia, dokumentointia ja raportointia. Näillä apuvälineillä voidaan luoda omia ympäristö-kokonaisuuksia. Tällaiset ympäristöt helpottavat paljon testauksen organisointia, analysointia ja raportointia. [Kau96]

Testauksen hallinnan työkalut sijoittuvat koko systeemin elinkaaren alueelle (Kuva 14, sivu 54). Esimerkkeinä mainittakoon Test Manager ja CTB. Test Manager mahdollistaa testauspaketin rakentamisen, suorittamisen ja hallinnan. Se on Javalla toteutettu ja tuottaa interaktiivisen kehitysympäristön, joka mahdollistaa työskentelyn yhdessä regressiotestauspakettien kanssa. CTB (C Test Bed System) on testialustojen generoija moduuli- ja integrointitestausvaiheeseen C-kielisille ohjelmille. Ominaisuuksiltaan CTB tukee parhaiten kokoavaa (bottom-up) mustalaatikko (black box) -menetelmää. Mahdollista on myös jäsentävän (top-down) etenemistavan käyttö. CTB-generaattori luo testattavista moduuleista testiajurin. Tämä käännetään CTB-kirjaston ja tarvittavan C-koodin kanssa lopulliseksi testipetiohjelmaksi. [Kau96]

6.6 TESTAUKSEN TOTEUTUKSEN TYÖKALUT

Testauksen toteutuksen työkalut ryhmä koostuu erilaisista välineistä, jotka auttavat testaamisessa. Tällaisia ovat esimerkiksi testitynkien generointiohjelmat sekä työkalut, jotka pyrkivät mahdollistamaan virheiden helpomman havaitsemisen. Eräs automatisoinnin apuväline on AssertMate for Java, joka antaa Java ohjelmoijille mahdollisuuden asettaa ohjelmiinsa totuusarvoväittämiä. Nämä helpottavat löytämään virheet aikaisemmassa kehitysvaiheessa ja silloinhan ne ovat myös nopeammin ja halvemmalla korjattavissa. Kun käytetään oliopohjaista suunnittelua, auttavat nämä väittämät toteuttamaan luokat luokkakaaviossa oikein. [Poh02]

6.7 TESTAUKSEN ARVIOINNIN TYÖKALUT

Testauksen arvioinnin työkalut ovat testauksen laadunvalvontaan liittyviä apuvälineitä. Koodin kattavuutta, eli siitä onko kaikki ohjelman haarat käyty läpi, mittaa esimerkiksi Rational PureCoverage. Se näyttää aina pyydettäessä, mitkä ohjelmarivit ovat vielä testaamatta. Tarkoitus on nopeuttaa testausta ja saada aikaan virheettömämpää koodia. [Poh02]

Koodin kattavuudella voidaan tarkoittaa useaa eri asiaa. Yksinkertaisimmillaan tutkitaan, mitkä koodirivit ohjelmasta on suoritettu. On varmaa, että riveistä, joita ei suoriteta, ei löydy virheitä. Tätä kutsutaan yleensä *lausekattavuudeksi (statement coverage)*. Hieman tehokkaampia kattavuustutkimuksia ovat *päätöskattavuus (branch coverage*), *ehto-kattavuus (condition coverage)* ja *moniehtokattavuus (multicondition coverage)*. [Mar97]

Lausekattavuus tutkii, että kaikki ohjelman lauseet on suoritettu. Päätöskattavuus varmistaa, että kaikki ohjelman vuokaavion kaaret on käyty läpi. Ehtokattavuudessa kaikki ehtolauseet saavat sekä tosi-, että epätosiarvot. Moniehtokattavuudessa kaikkien ehtolauseiden kaikki mahdolliset tosi- ja epätosiarvojen kombinaatiot käydään läpi. *Riippumaton polkukattavuus (independent path coverage)* käsittää kaikki polut, alkusolmusta loppusolmuun, joita ei voida muodostaa muiden polkujen alipoluista. Täydellistä polkukattavuutta on mahdotonta saavuttaa ohjelmassa olevien silmukoiden vuoksi.[Paa00]

6.8 STAATTISEN ANALYYSIN TYÖKALUT

Staattisen analyysin työkalujen tarkoitus on löytää ohjelmassa olevat virheet ilman ohjelman suorittamista [Kau96]. Esimerkkinä tällaisesta apuvälineestä on kääntäjä, joka havaitsee suuren osan ohjelmakoodissa olevista virheistä. Staattisia analysoijia taas käytetään kompleksisuuden laskemiseen ja ohjelman rakenteen analysointiin.

Eräs staattisen analyysin työkaluista on CMT++ (Complexity Measures Tool for C/C++). Se on kompleksisuuden mittaustyökalu MS-DOS-ympäristöön. Tulokset perustuvat McCaben [McC76] ja Halsteadin [Hal77] ohjelmamittareihin sekä välineen omiin lukuihin

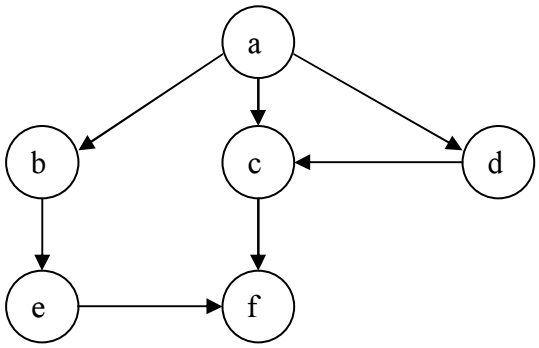
ohjelmarivien määristä. Apuvälinettä voidaan käyttää joko suoraan komentorivikäskyillä tai interaktiivisesti ohjelman antamien vaihtoehtojen kautta [Kau96].

Mittaustuloksina saadaan

- McCaben syklomaattinen luku (v(G))
- Kommenttirivien lukumäärä (LOCcom)
- Ohjelmarivien lukumäärä (LOCpro)
- Ohjelman koko (V)
- Arvio virheiden lukumäärästä ohjelmassa (B)
- Arvio ajasta, joka menee ohjelman ymmärtämiseen (T)

McCaben mukaan [McC76] ohjelman kompleksisuutta kuvaava v(G) lasketaan seuraavan kaavan mukaan: v(G) = e – n + p, missä e on kaarien lukumäärä ohjelmaa kuvaavassa graafissa (Kuva 15), n solmujen lukumäärä ja p on yhdistettyjen komponenttien lukumäärä (alku- ja loppusolmut). Kuvan 15 ohjelman kompleksisuus on kolme (7 – 6 + 2). Ohjelman suositeltava kompleksisuuden yläraja v(G) olisi McCaben mukaan kymmenen.

Mittaustulosten analysoinnissa on huomattu, että funktio-ohjelmien pituuden tulisi olla 4 - 40 riviä, yhden ohjelman pituuden alle 400 riviä, v(G) alle 15 ja B alle 2. Nämä tulokset ovat vain suuntaa-antavia. [Kau96]



Kuva 15. Ohjelmaa kuvaava graafi [McC76]

7 APUVÄLINEEN VALINTA JA KÄYTTÖÖNOTTO

7.1 APUVÄLINEEN VALINTA AUTOMATISOITAESSA TESTAUSTA

Testauksen työkalun valintaan kuuluvat käyttäjän tarpeiden tunnistaminen, välineen arviointimenetelmän luominen, sopivien ehdokkaiden löytäminen ja työkalun valinta ja arviointi siitä, mikä on investoinnin takaisinmaksuaika. [PoS92]

Luulo, että automatisoinnin työkalun rakentaminen on halvempaa kuin ostaminen, on väärä. Yleensä rakentaminen on aivan yhtä kallista, ellei jopa kalliimpaa kuin ostaminen. Samoin ajatus rakentamisen helppoudesta (oletetaan, että välineeseen ei tarvita mitään hienouksia) on virheellinen. Minkään työkalun rakentaminen ei ole helppoa. Uskomus, että pystytään rakentamaan parempi työkalu kuin markkinoilla on tarjolla, ei aina pidä paikkaansa. Kaupallisia tuotteita valmistavat yritykset ovat käyttäneet suunnattoman määrän aikaa ja rahaa tuotekehittelyyn. On sula mahdottomuus tehdä muutamassa viikossa vastaava työkalu [Hen00]. Pelkästään valmiin välineen käytön opettelu kestää kokeneeltakin ammattilaiselta ainakin kaksi viikkoa [FeG99]. Monet ajattelevat, että rakentaminen antaa enemmän joustavuutta, mikäli prosessi muuttuu työn aikana. Mikäli ei tiedetä, mihin tarkoitukseen työkalua tullaan lopulta käyttämään, ei ole järkevää ostaa, eikä ruveta rakentamaan [Hen00].

Katsotaanpa asioita päinvastaisesta näkökulmasta: Oletus, että ostaminen on helpompaa kuin rakentaminen, ei pidä paikkaansa. Uuden tuotteen käyttöönotto aiheuttaa aina muutoksia, ja niitä useimmat ihmiset vastustavat luonnostaan. Vaikka ei tarvittaisi ohjelmointia yrityksen sisällä, uuden välineen tuominen organisaatioon ei ole helppoa. Entä saadaanko tuote nopeammin käyttöön, jos se ostetaan? Ostettaessa säästyy suunnittelu- ja ohjelmointiaikaa, mutta muut toiminnot (välineen valinta, käsiteltävän datan siirtäminen uuteen ympäristöön, käyttöönotto ja käyttäjien kouluttaminen) vievät paljon aikaa. Viimeinen kumottava uskomus on se, että ostamalla testausväline ei tarvita ylläpitoa. On aivan sama rakennetaanko vai ostetaanko – ylläpitoa tarvitaan aina. [Hen00]

Päätöksentekoon vaikuttavat kriteerit ovat seuraavat: Voimmeko saada työkalun jostain muualta? Saammeko joitakin merkittäviä etuja valitsemalla ostamisen tai rakentamisen?

Kuinka päätös vaikuttaa sovellusten kehityksen joustavuuteen pidemmällä aikavälillä? [Hen00]

Marick antaa työkalun arviointiin muutamia kriteereitä: Onko työkalu parempi, jos se löytää enemmän virheitä kuin joku toinen työkalu? Näin ei välttämättä ole. Täytyy kiinnittää huomiota myös siihen, minkälaisia virheitä löytyi. Millainen on työkalun ratkaisumalli? Onko tärkeät piirteet toteutettu? Onko epäoleelliset piirteet karsittu pois? Onko ylläpidettävyys hyvä? Mitä todella tapahtuu, kun ohjelmistoa kehitetään? Tarkastele testausta yksityiskohdittain. Voitko löytää testaamattomia kohtia? Mitä väline ei tee? Esimerkiksi monet koodin kattavuuden työkalut ovat huonoja toiminnallisilta ja laadullisilta ominaisuuksiltaan. Mitä välineen myötä menetetään? Jokaiselta työkalulta jää huomaamatta joitakin virheitä, jotka mahdollisesti paljastuvat systeemin käyttövaiheessa. Täytyykin arvioida, miten kriittisiä nämä virheet ovat. [MaB00]

7.1.1 Ostaminen

Hendricsson esittää työkalun arviointiin viiden kohdan ohjelman: Ensimmäiseksi tehdään esiselvitys siitä, mitä ominaisuuksia tuotteelta odotetaan. Toiseksi tutkitaan markkinoilla olevia tuotteita mahdollisimman laajasti – ei kuitenkaan vielä demonstroimalla niitä. Kolmannessa vaiheessa tarkennetaan määrittelyjä niiden tietojen avulla, mitä tähän mennessä on saatu. Nyt voidaan ottaa mukaan tuotteen lopullisia käyttäjiä. Seuraavaksi valitaan ehdolla olevista kaksi tai kolme tarkempiin tutkimuksiin. Enempää ei kannata ottaa, koska työkalun käytön opettelu ja kokeileminen vie aikaa niin paljon, ettei sitä ehditä suorittaa suuremmalle määrälle. Lopuksi pyydetään tuotteiden toimittajilta kokeiluversiot. Mikäli toimittaja tarjoaa omaa henkilöään kokeiluun, mukana on ehdottomasti oltava myös ostajan oma edustaja. Tässä vaiheessa paljastuu, onko tuotteen asennus ja käyttö hankalaa. Parasta olisi käyttää jotain todellista testitilannetta, ei mitään keksittyä. Tuotteen valinnan jälkeen on vielä suunniteltava koulutus ja käyttöönotto, jotka ovat hyvin merkittäviä osatekijöitä onnistunen automatisoinnin kannalta. [Hen99a]

Jotkut yritykset onnistuvat jatkuvasti ostamaan sellaisia testauksen apuvälineitä, jotka palvelevat heidän testaajiaan ja kehittäjiään hyvin. Eräänä vaikuttavana tekijänä on varmasti se, että he käyttävät jotakin systemaattista tiedonkeruumenetelmää arvioidessaan

eri työkaluja. Se voi olla moniosainen tarkastuslista, jossa analysoidaan käyttäjän tarpeita, selvitetään valintakriteerit, kartoitetaan välineet, suoritetaan valinta ja tehdään jälkiarviointi. [PoS92]

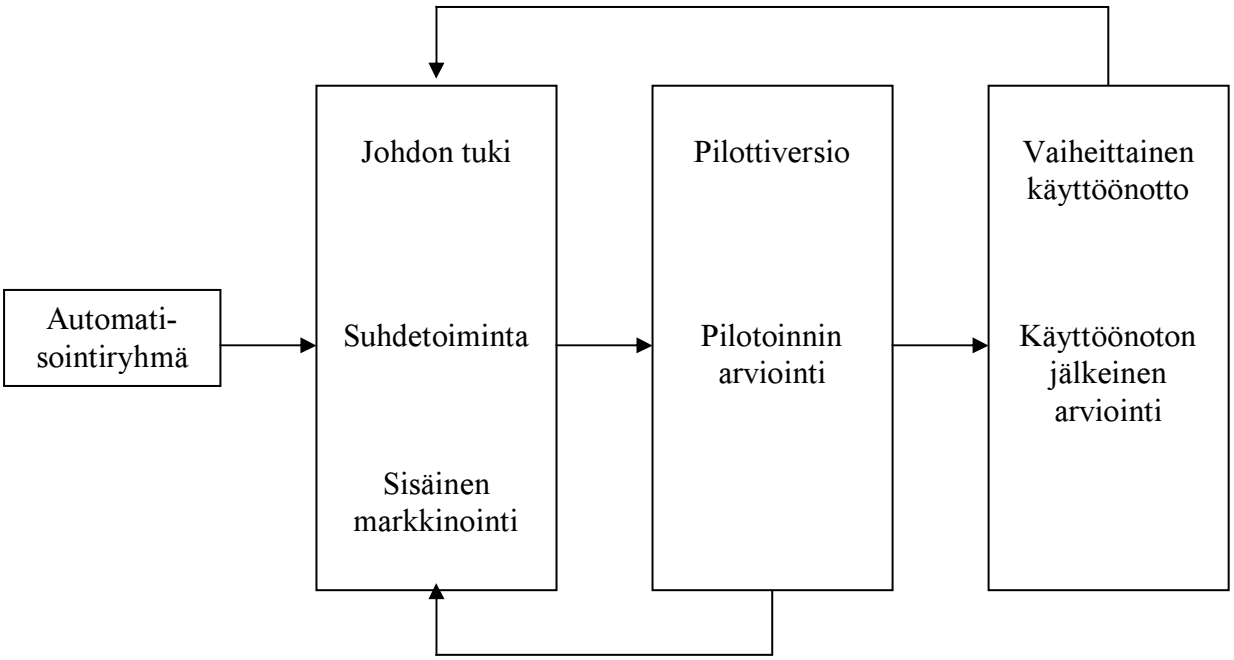
7.1.2 Rakentaminen

Mikäli lähdetään rakentamaan itse, työkalun täytyy olla mahdollisimman tarkoituksenmukainen omiin tarpeisiin. Työkalulla itsellään täytyy olla mahdollista parantaa testattavuutta ohjelmistossa, jota testataan. Työkalu tulee sisältämään sovellustuntemusta, joten se vähentää automatisoidun testauksen käyttöönottotyötä. Dokumentointi, avustus (help) ja koulutus eivät todennäköisesti ole kovin hyvin tuettuja itse rakennetussa apuvälineessä. Omassa talossa tehtyä ei mielletä yhtä hyväksi ja arvokkaaksi kuin kaupalliselta valmistajalta ostettua eli syntyy niin sanottu imago-ongelma. Käyttöliittymässä voi olla toivomisen varaa, koska tekniikan ihmiset eivät juurikaan pidä käyttämisen helppoutta tärkeänä. [FeG99]

7.2 APUVÄLINEEN KÄYTTÖÖNOTTO YRITYKSEN SISÄLLÄ

Kun apuvälineen valinta on takanapäin, alkaa todellinen työ. Vaikka valinta olisi tehty kuinka huolellisesti tahansa, ei ole mitään takeita, että työkalun käyttämisessä onnistutaan. On paljon yrityksiä, jotka ovat menestyksellisesti valinneet ja hankkineet työkalun, mutta suunnilleen puolet näistä ei ole saanut siitä koskaan mitään hyötyä. Tämä sen takia, että apuvälinettä ei ole koskaan otettu käyttöön tai se on jäänyt pois käytöstä hyvin pian. [FeG99]

Apuvälineen valintaprosessissa vähennetään vaihtoehtojen lukumäärää asteittain. Käyttöönotto on päinvastainen prosessi. Työkalun hyväksymistä, käyttöä ja saavutettavia etuja laajennetaan asteittain, kuten kuva 16 osoittaa. [FeG99]



Kuva 16. Apuvälineen käyttöönottoprosessi [FeG99]

Työkalun ostamis- tai vuokrauskustannukset ovat hyvin pienet verrattuna käyttöönottokustannuksiin. Apuväline täytyy ”myydä” yrityksen sisällä. On järjestettävä tuki ja koulutus. On rakennettava infrastruktuuri tukemaan jatkuvaa testauksen automatisointijärjestelmää. Täytyy muistaa, että kun yritykseen tuodaan uusi testauksen työkalu, se muuttaa tapaa, jolla ihmiset työskentelevät. Muutosprosessia on mahdollista keventää muun muassa hyvällä koulutuksella. [FeG99]

8 TESTAUSKOKEMUKSIA

8.1 KIRJALLISUUDESTA

8.1.1 Epäonnistumisia

Amland kertoo artikkelissaan kokemuksistaan automatisoidusta testauksesta kahdessa eri projektissa. Projekti A oli pienen pankin palvelimen systeemitestaus ja projekti B kotipankkijärjestelmän graafisen käyttöliittymän hyväksymistestaus. [Aml99)

Testauksen työkalujen ongelmia olivat työkalun yhteensopimattomuus kehitysvälineen kanssa, ikkunan koon muuttaminen (lamaannutti testauksen) ja pääongelmana se, ettei apuväline tunnistanut näytön elementtejä olioiksi, vaan ne piti kuvata pikselitasolla. Testaajien ongelmana oli se, että he olivat tuotannon ihmisiä eli tulevia käyttäjiä, eikä heillä ollut ohjelmointitaitoa. Testiympäristössä vaikeuksia syntyi huonosta tiedonkulusta testaajien välillä ja testipetien kelvottomuudesta. [Aml99]

Tapauksessa B oli arviolta 100 erilaista ikkunaa. Kokonaissysteemissä oli yli 15 keskus-yksikköä, viisi laitteistoalustaa, neljä ohjelmistotoimittajaa ja kaksi eri Windows versiota. Systeemin rakentajia tarvittiin 25 kuuden kuukauden ja testaajia 25 kolmen kuukauden ajan. Testaajista kahdeksan keskittyi pelkästään testauksen hallintaan ja kaksi testauksen automatisointiin. Automatisointi oli osana loppukäyttäjien hyväksymistestausta. [Aml99]

”Kaiken kaikkiaan yritimme automatisoida liian paljon liian lyhyessä ajassa”, kirjoittaja toteaa. Esim. A tapauksessa saatiin testattua vain 15 % tapahtumista käyttäen 25 % saatavilla olevista resursseista ja ainoastaan 2.5 % virheistä löydettiin. Alkuperäisenä pyrkimyksenä oli päästä 100 % automatisointiin. Tämä vaatimus osoittautui testauksen edetessä todella turhauttavaksi. [Aml99]

Johtopäätöksenä kaikesta tästä olikin, että testiryhmän koulutukseen täytyy käyttää huomattavasti enemmän aikaa ja sen jäsenten pitää olla atk-ammattilaisia. Myös sopivan testityökalun valintaan on kiinnitettävä enemmän huomiota. Kannattaa harkita sitäkin, milloin lähdetään automatisoimaan testausta ja milloin tyydytään perinteisiin menetelmiin.

Amland kuitenkin uskoo edelleen automatisoituun testaukseen, vaikka se onkin vaikeaa ja vaatii paljon resursseja. [Aml99].

Dustin kertoo artikkelissaan monista kohtaamistaan vaikeuksista automatisointiurallaan. Käytettäessä V-mallin (Kuva 1, sivu 11) eri vaiheissa eri valmistajien työkaluja oli vaikeaa siirtää edellisen vaiheen tuloksia seuraavaan. Tarvittiin hyvin monimutkaisia ohjelmointitekniikoita ja paljon ylimääräistä työtä työkalujen integrointiin. Kuitenkaan aikaansaatu koodi ei ollut käytettävissä uudelleen, mikäli työvälineistä tuli käyttöön uudempi versio. Kannattaa analysoida, milloin on edullisempaa ostaa valmis työkalupaketti, kuin lähteä yhdistämään toisilleen vieraita apuvälineitä. Hankittaessa uusi testauksen hallinnan työkalu, jo olemassa olevien lisäksi, havaittiin, että samaa tietoa tallennettiin ja säilytettiin eri paikoissa ja tiedon ylläpito oli erittäin vaikeaa. Välineiden lisääminen on monesti johtanut suorastaan huonompaan tuottavuuteen. Uuden työkalun käyttöönotto teettää monesti niin paljon työtä testiskriptien kirjoittamisessa, että varsinaiseen testaukseen ei enää jää tarpeeksi aikaa. Aikaa automatisointiin pitää olla riittävästi. Työkaluun tutustuminen on aloitettava jo varhaisessa suunnitteluvaiheessa ja rahallisia säästöjä ei kannata odottaa välittömästi. Dustin haluaa vielä painottaa sitä seikkaa, että kaikkea ei ole mahdollista automatisoida. [Dus99]

Hendricksonilta löytyy kokemusta sekä epäonnistumisesta ja sen jälkeen onnistumisesta. Hän toimi projektin johtajana kummassakin tapauksessa. Kyseessä oli sama yritys ja työn kohteena kummassakin samantyyppinen projekti eli monimutkaisen serverijärjestelmän automatisointi avoimelle arkkitehtuurille. Vastaavasti erot projekteissa syntyivät seuraavista tekijöistä: päämäärien asettelu, tiimin kokemus, viestintä ja testauksen automatisoinnin merkityksen ymmärtäminen ja soveltaminen. Kaikki edellä mainitut seikat olivat ensimmäisessä projektissa käytännössä huonommin hoidettuja kuin toisessa. Päämäärät olivat ylirealistiset – kaikki piti automatisoida kerralla. Tiimi oli kokematon, etenkin sen johtaja, eikä kommunikointi toiminut – ei tiimin kesken, eikä varsinkaan ylemmän johdon kanssa. Johto ei tiennyt, mihin rahat käytetään ja mitä etuja automatisoinnista on. Testauksen automatisointia ei ymmärretty eikä toteutettu oikein. Organisaatiossa ei edes kunnolla tiedetty, miten tuote testataan manuaalisesti. Automatisoinnin rakentajat eivät arvostaneet manuaalisten testaajien työtä. Kaikki tämä johtikin projektin suhteellisen nopeaan lopettamiseen. [Hen98]

8.1.2 Onnistumisia

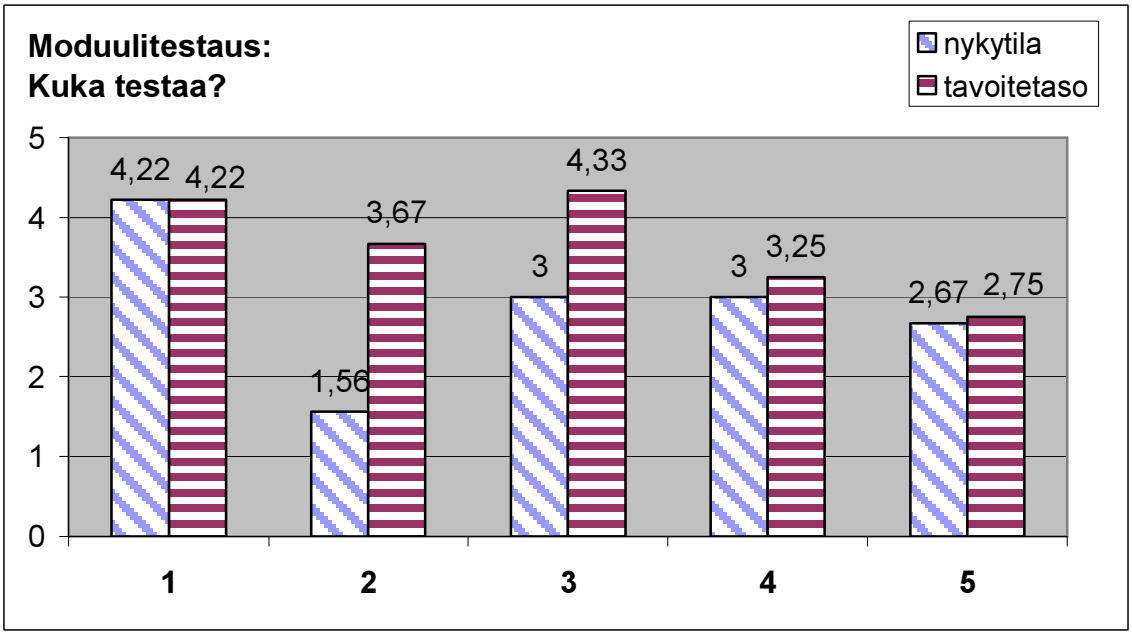
Parin vuoden kuluttua tehtiin uusi yritys. Nyt kaikki oli toisin. Yrityksen johto oli vaihtunut. Projektipäällikkö oli edelleen sama mutta hän oli ehtinyt hankkia vankkaa kokemusta sekä johtamisesta että teknisestä osaamisesta. Projektille ei tällä kerralla annettu tavoitteita, vaan ryhmäläiset saivat määritellä ne itse ja esittää johdolle. Tavoitteena oli säästää manuaalisten testaajien aikaa ja parantaa testauksen kattavuutta. Kommunikointi oli jatkuvaa ja säännöllistä. Yrityksessä oli tiedostettu testauksen tärkeys ja testaus oli järjestelmällistä. Projektiryhmä koki tekevänsä jotakin hyödyllistä ja he työskentelivät yhdessä manuaalisten testaajien kanssa. Elisabeth Hendrickson sanoo oppineensa paljon näistä kahdesta projektista. Esimerkiksi sen, miten tärkeää on realististen tavoitteiden asettaminen ja niihin pyrkiminen, oikeanlaisen välineen käyttö ja uudelleen käytettävään, ylläpidettävään koodiin keskittyminen. Myöskään ihmissuhteiden merkitystä ei ole syytä unohtaa. [Hen98].

8.2 YRITYSKYSELYN PERUSTEELLA

PlugIT-projektissa tehtiin alkuvuodesta 2003 yrityskysely ohjelmistotuotannon nykytilasta hankkeessa mukana olevissa yrityksissä, sairaaloissa, terveyskeskuksissa, sairaanhoitopiireissä ja kunnissa [Plu03]. Osallistuin kyselyn testauksen ja tarkastuksen osion toteuttamiseen. Varteenotettavia vastaajia tällä osa-alueella oli 13, joista kymmenen vastasi ja näistä yhdeksän oli käyttökelpoisia tilastolliseen analyysiin. Kyselyssä oli mukana yhdeksän ohjelmistotaloa ja loput sairaaloita, sairaanhoitopiirejä jne. Käyttökelpoisista vastauksista kuusi oli ohjelmistoyrityksistä, yksi sairaalan ATK-osastolta ja yksi erikoissairaanhoidosta. Yksi vastaaja ei ilmoittanut yksikkönsä tyyppiä. Vaikka otos oli pieni, tuloksia voidaan pitää vähintään suuntaa antavina terveydenhuollon parissa työskentelevien ATK-ammattilaisten käyttämistä testausmenetelmistä ja kehittämistarpeista. Nykytilan ja tavoitetason vertailu, sekä käytettävyyden arviointi ovat luotettavia, koska vastausfrekvenssi pysyi koko ajan lähes samana. Nykytilan ja tavoitetason vertailussa asteikko oli seuraava: 0 = ei koskaan, 1 = harvoin, 2 = joskus, 3 = usein, 4 = lähes aina, 5 = aina. Kyselyssä käytetty kyselylomake on liitteenä (Liite 1).

8.2.1 Moduulitestaus

Moduulitestauksessa oli tällä hetkellä tilanne se, että enimmäkseen testauksen suorittivat ohjelmoijat itse. Selvästi tarvetta oli lisätä ohjelmointiparien ja toimittajan testausryhmän tekemää testausta (Kuva 17). Menetelmien käytettävyydessä parhaaksi arvioitiin toimittajan testausryhmän testaus, joka myös tavoitetasoltaan nousi korkeimmalle



Kuva 17. Kuka testaa moduulitestauksessa [Kär03]

1. Ohjelmoija itse, nykytila 4,22 - Tavoitetaso 4,22

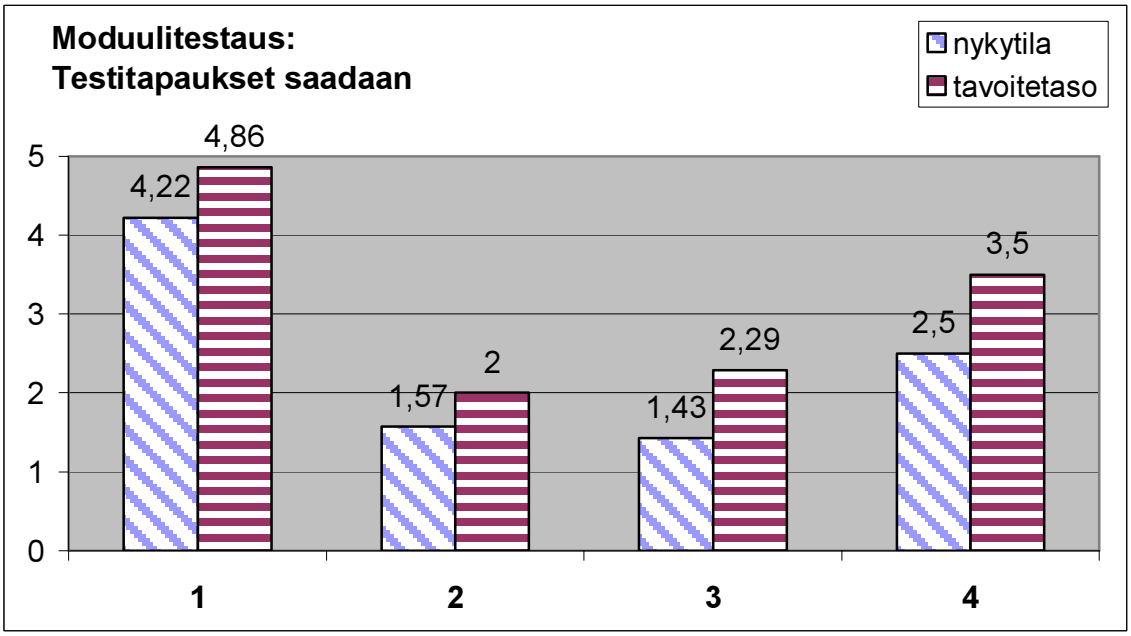
2. Ohjelmoijat vuorotellen toistensa moduulit, nykytila 1,56 - Tavoitetaso 3,67

3. Toimittajan testausryhmä, nykytila 3,00 - Tavoitetaso 4,33

4. Asiakkaan testausryhmä, nykytila 3,00 - Tavoitetaso 3,25

5. Loppukäyttäjä, nykytila 2,67 - Tavoitetaso 2,75

Moduulitestauksen testitapaukset saadaan pääasiallisesti käyttötapauksista. Pienimmät ryhmät ovat aktiviteetti- ja tilakaavioista saadut testitapaukset. Toiseksi suurin osuus on jostakin muualta saadut. Kaikkien ryhmien tavoitetaso on korkeampi kuin nykytila (Kuva 18). Käytettävyydeltään ylivoimaisesti paras oli menetelmä, jossa testitapaukset saatiin käyttötapauksista.



Kuva 18. Miten testitapaukset saadaan [Kär03]

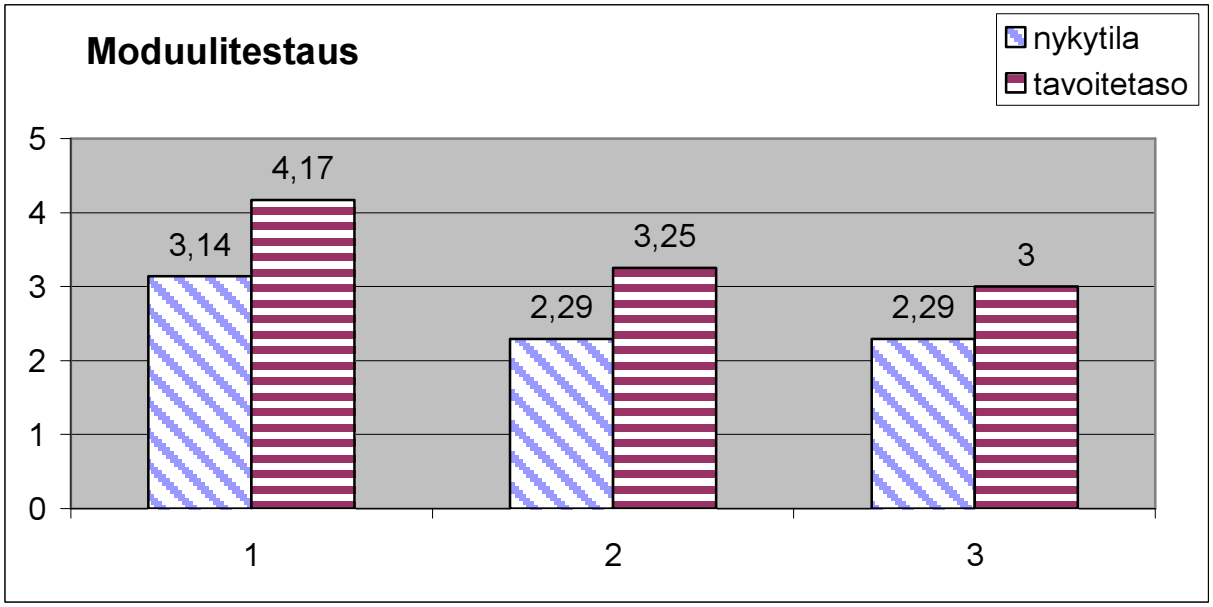
1. Käyttötapauksina, nykytila 4,22 - Tavoitetaso 4,86

2. Aktiviteettikaaviosta, nykytila 1,57 - Tavoitetaso 2,00

3. Tilakaaviosta, nykytila 1,43 - Tavoitetaso 2,29

4. Muualta, nykytila 2,50 - Tavoitetaso 3,50

Testausmenetelmistä, eli musta-, lasi- ja harmaalaatikosta, suosituin oli mustalaatikkotestaus, jonka tavoitetaso oli myös korkein (Kuva 19). Ekvivalenssiluokituksen hyväksikäyttö oli vaatimatonta, mutta tavoitetasomittauksessa sen odotettiin kehittyvän eniten. Käytettävyydeltään kaikkia kolmea edellä mainittua menetelmää pidettiin likimain yhtä hyvinä – lähes kakkosen arvoisina asteikolla 0 – 3.



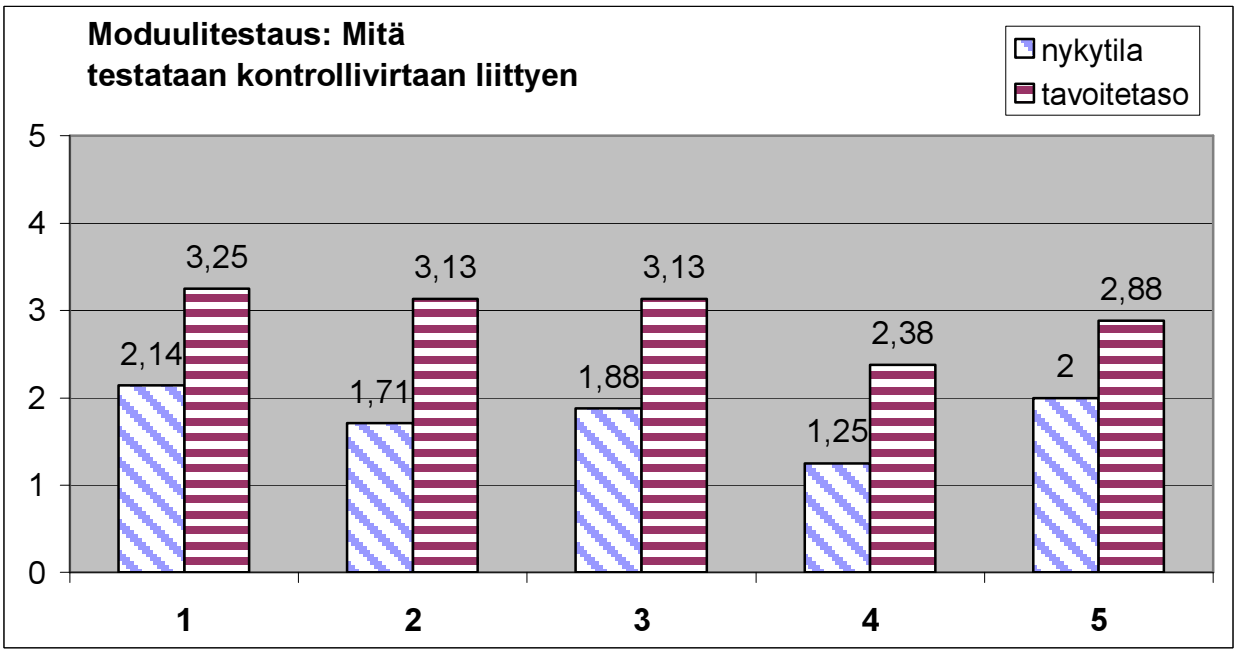
Kuva 19. Musta-, lasi- ja harmaalaatikon käyttöaste moduulitestauksessa [Kär03]

1. Mustalaatikko, nykytila 3,14 - Tavoitetaso 4,17

2. Lasilaatikko, nykytila 2,29 - Tavoitetaso 3,25

3. Harmaalaatikko, nykytila 2,29 - Tavoitetaso 3,00

Lasilaatikkotestausta suoritettiin joko kontrollivirtaan tai tietovirtaan perustuen. Kontrollivirtatestauksen eri menetelmiä käytettiin suhteellisen tasaisesti. Ainoastaan moniehtokattavuuteen perustuva testaus erottautui muista vähäisemmällä käytöllään. Kaikkien menetelmien tavoitetaso oli huomattavasti korkeammalla nykyistä tasoa. Suhteellisesti eniten lisäystä toivottiin päätöskattavuuden ja moniehtokattavuuden käyttöön (Kuva 20). Menetelmistä käytettävimpinä pidettiin lause- ja päätöskattavuutta.



Kuva 20. Kontrollivirtaan liittyvä testaus [Kär03]

1. Lausekattavuus= kaikkien lauseiden läpikäynti, nykytila 2,14 - Tavoitetaso 3,25

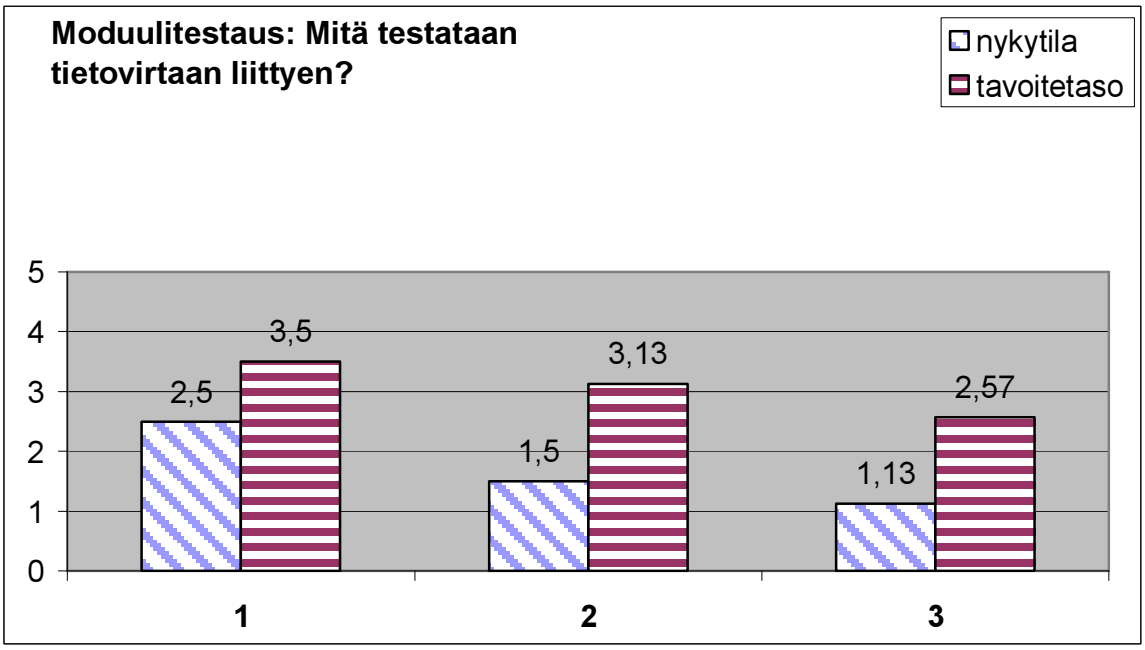
2. Päätöskattavuus= kaikkien ohjelman vuokaavion kaarien läpikäynti, nykytila 1,71 - Tavoitetaso 3,13

3. Ehtokattavuus= kaikkien ehtolauseiden 'tosi' ja 'epätosi' arvojen läpikäynti, nykytila 1,88 - Tavoitetaso 3,13

4. Moniehtokattavuus= kaikkien ehtolauseiden 'tosi' ja 'epätosi' arvojen eri kombinaatioiden läpikäynti, nykytila 1,25 - Tavoitetaso 2,38

5. Polkukattavuus= mahdollisimman monien ohjelman eri polkujen läpikäynti, nykytila 2,00 - Tavoitetaso 2,88

Tietovirtaan perustuvassa testauksessa suosituin oli menetelmä, jossa kaikki määrittelyt saavuttavat jonkin käyttönsä. Tavoitteena oli kaikkien menetelmien käytön lisääminen (Kuva 21). Käytettävyydeltään parhaiksi arvioitiin menetelmät: "kaikki määrittelyt saavuttavat jonkin käyttönsä" ja "kaikki määrittelyt/käytöt saavuttavat toisensa kaikkia mahdollisia polkuja pitkin".



Kuva 21. Tietovirtaan liittyvä testaus [Kär03]

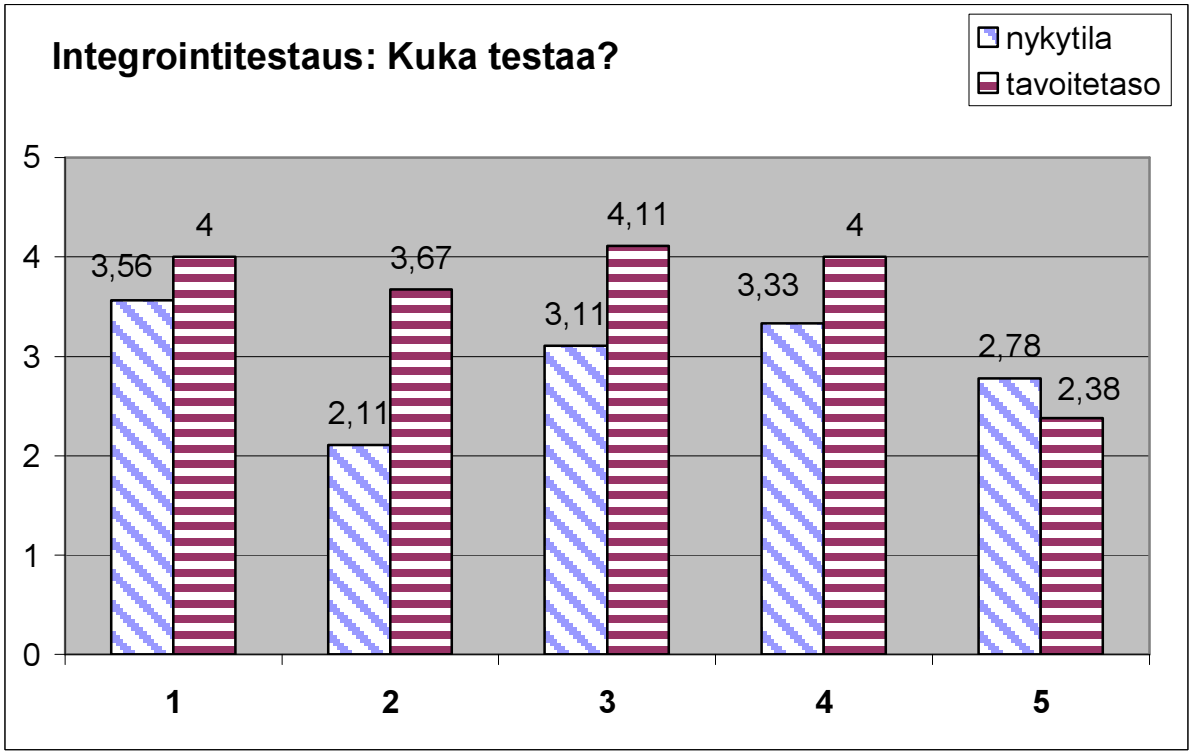
1. Kaikki määrittelyt saavuttavat jonkin käyttönsä, nykytila 2,50 - Tavoitetaso 3,50

2. Kaikki määrittelyt saavuttavat kaikki käyttönsä, nykytila 1,50 - Tavoitetaso 3,13

3. Kaikki määrittelyt/käytöt saavuttavat toisensa kaikkia mahdollisia polkuja pitkin, nykytila 1,13 - Tavoitetaso 2,57

8.2.2 Integrointitestaus

Integrointitestausta (Kuva 22) suorittavat tällä hetkellä eniten ohjelmoijat itse. Selvästi tavoitteena on lisätä toisten tekijöiden töiden testaamista ja toimittajan ja asiakkaan testausryhmän suorittamaa testausta. Loppukäyttäjän testausosuutta haluttiin pienentää. Käytettävyydeltään parhaana pidettiin toimittajan testausryhmää.



Kuva 22. Kuka testaa integrointitestauksessa [Kär03]

1. Ohjelmoija itse, nykytila 3,56 - Tavoitetaso 4,00

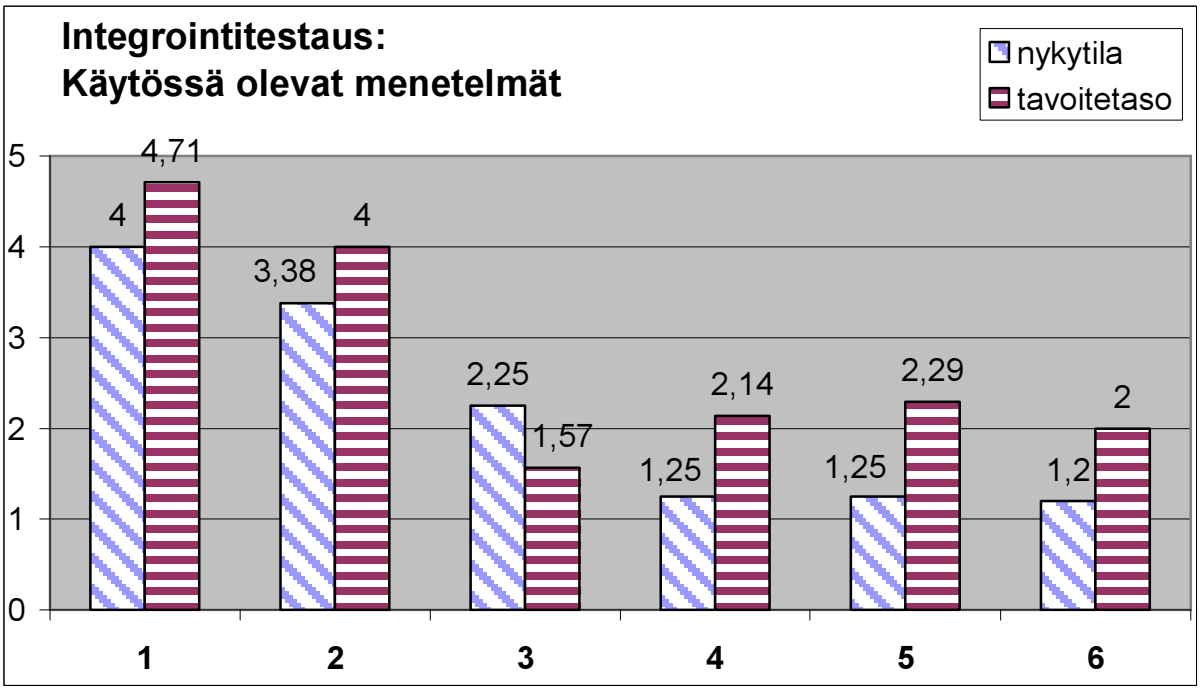
2. Tekijät testaavat vuorotellen toistensa työt, nykytila 2,11 - Tavoitetaso 3,67

3. Toimittajan testausryhmä, nykytila 3,11 - Tavoitetaso 4,11

4. Asiakkaan testausryhmä, nykytila 3,33 - Tavoitetaso 4,00

5. Loppukäyttäjä, nykytila 2,78 - Tavoitetaso 2,38

Integrointitestauksessa käytetyistä testausmenetelmistä suosituin oli mustalaatikko ja varsinaisista yhteenliittämismenetelmistä kertarysäys. Kertarysäyksen osuutta haluttiin kuitenkin pienentää. Tavoitetasossa voimakkain kasvutarve oli jäsentävällä, kokoavalla ja kerrosvoileipämenetelmällä (Kuva 23). Käytettävyydeltään parhaat olivat mustalaatikko, jäsentävä ja kokoava menetelmä.



Kuva 23. Integrointitestauksessa käytössä olevat menetelmät [Kär03]

1. Mustalaatikko, nykytila 4,00 - Tavoitetaso 4,71

2. Virheenjäljitys (debuggaus), nykytila 3,38 - Tavoitetaso 4,00

3. Kertarysäys (big-bang), nykytila 2,25 - Tavoitetaso 1,57

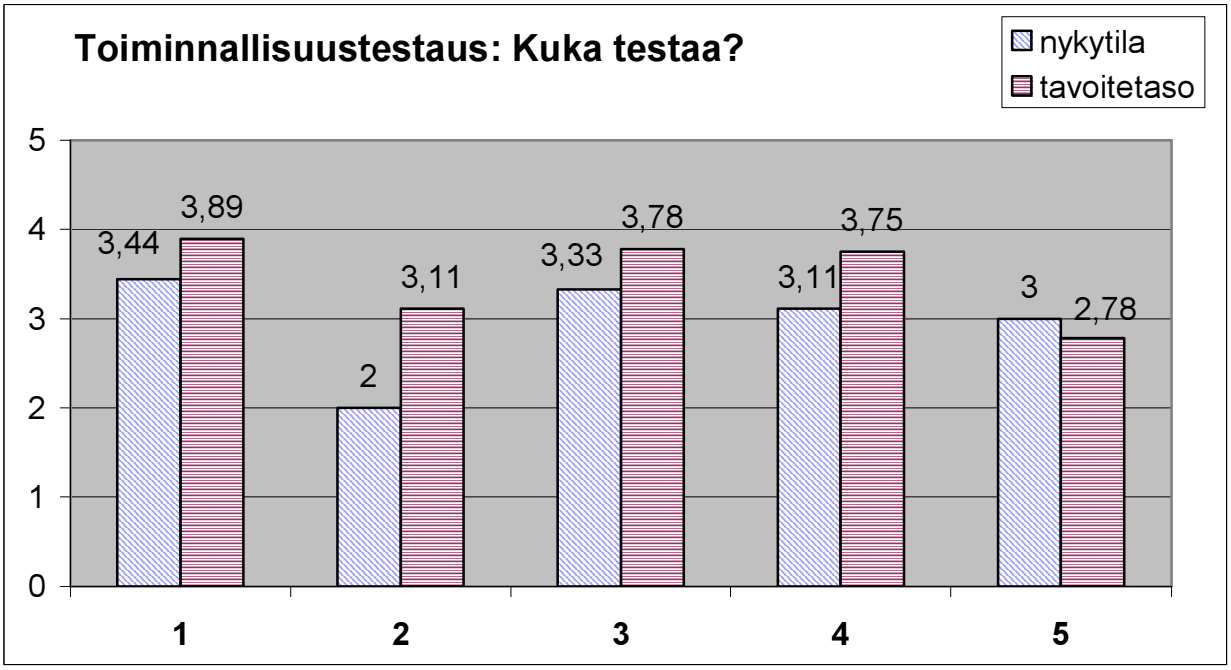
4. Jäsentävä (top-down), nykytila 1,25 - Tavoitetaso 2,14

5. Kokoava (bottom-up), nykytila 1,25 - Tavoitetaso 2,29

6. Kerrosvoileipä (sandwich), nykytila 1,20 - Tavoitetaso 2,00

8.2.3 Toiminnallisuustestaus

Toiminnallisuustestauksessa ohjelmoijan itsensä, toimittajan tai asiakkaan testausryhmän suorittamalla testauksella oli lähes yhtä merkittävä rooli. Voimakkain kasvutarve oli toisten tekijöiden töiden vuorotellen testaamisella (Kuva 24). Käytettävyydessä ei ollut merkittäviä eroja, joskin hiukan muita paremmaksi arvioitiin asiakkaan testausryhmä.



Kuva 24. Kuka testaa toiminnallisuustestauksessa [Kär03]

1. Ohjelmoija itse, nykytila 3,44 - Tavoitetaso 3,89

2. Tekijät testaavat vuorotellen toistensa työt, nykytila 2,00 - Tavoitetaso 3,11

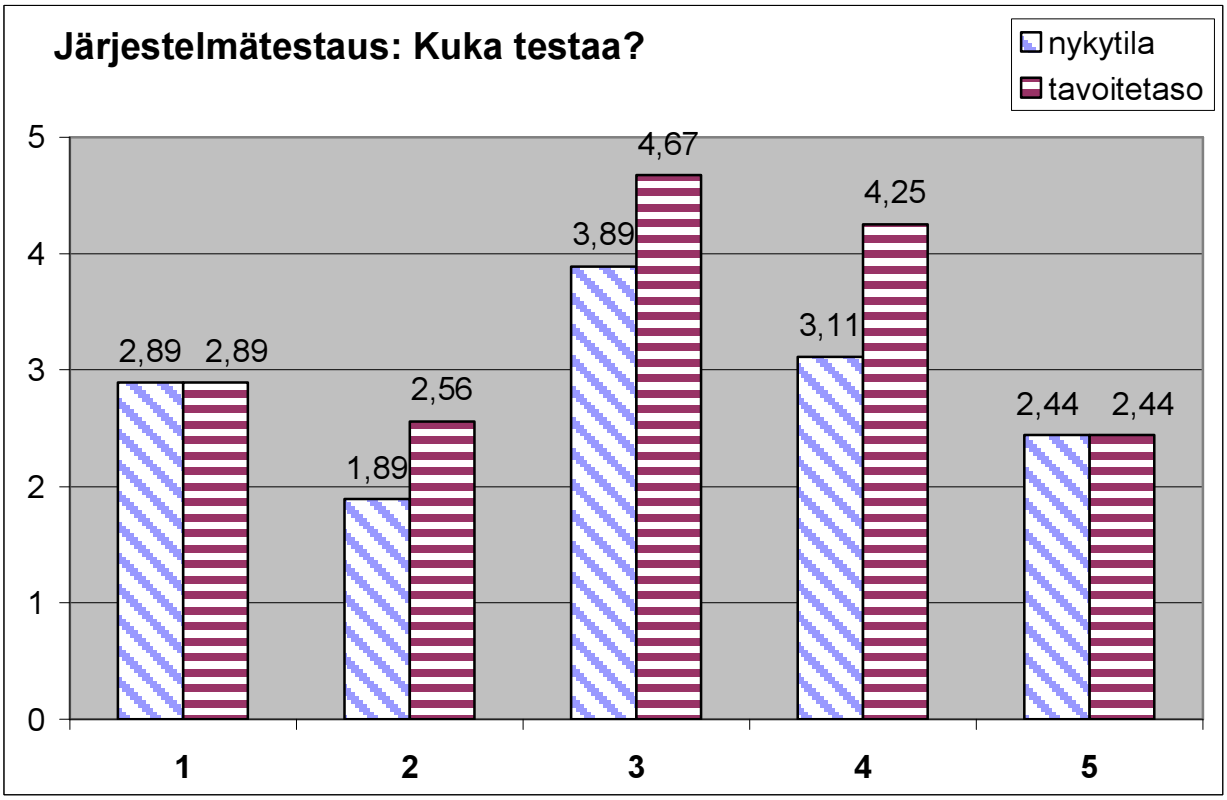
3. Toimittajan testausryhmä, nykytila 3,33 - Tavoitetaso 3,78

4. Asiakkaan testausryhmä, nykytila 3,11 - Tavoitetaso 3,75

5. Loppukäyttäjä, nykytila 3,00 - Tavoitetaso 2,78

8.2.4 Järjestelmätestaus

Selvästi suurin järjestelmätestauksen suorittajista oli toimittajan testausryhmä, jolle myös odotettiin kasvua niin paljon, että se pysyy jatkossakin suurimpana (Kuva 25). Käytettävyydeltään asiakkaan ja toimittajan testausryhmät olivat tasavertaiset.



Kuva 25. Järjestelmätestauksen suorittajat [Kär03]

1. Ohjelmoija itse, nykytila 2,89 - Tavoitetaso 2,89

2. Tekijät testaavat vuorotellen toistensa työt, nykytila 1,89 - Tavoitetaso 2,56

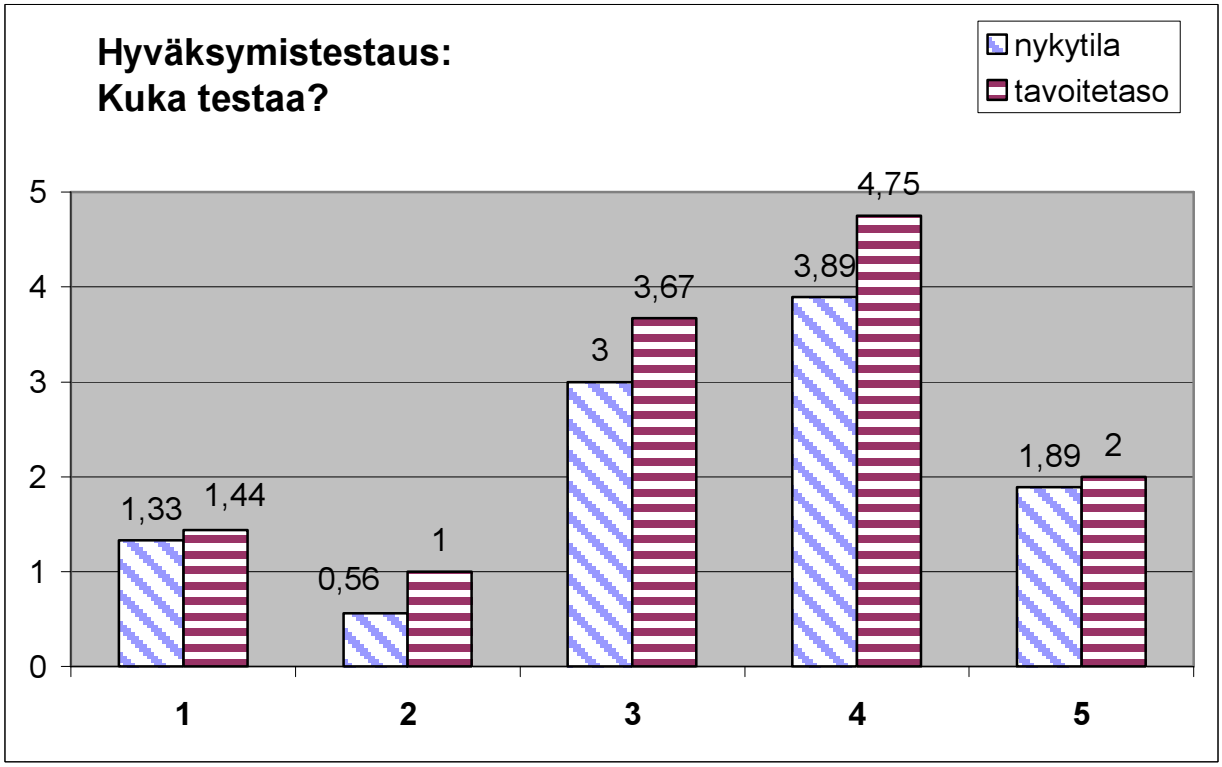
3. Toimittajan testausryhmä, nykytila 3,89 - Tavoitetaso 4,67

4. Asiakkaan testausryhmä, nykytila 3,11 - Tavoitetaso 4,25

5. Loppukäyttäjä, nykytila 2,44 - Tavoitetaso 2,44

8.2.5 Hyväksymistestaus

Hyväksymistestauksen suorittaa useimmiten asiakkaan testausryhmä, mutta suhteellisen usein myös toimittaja. Molemmilla on tulevaisuudessa kasvutarvetta. Loppukäyttäjän ja ohjelmoijan testausosuuden haluttiin kasvavan vain vähän (Kuva 26). Käytettävyydeltään asiakkaan testausryhmä ja toimittaja olivat selvästi kaksi parasta.



Kuva 26. Hyväksymistestauksen suorittajat [Kär03]

1. Ohjelmoija itse, nykytila 1.33 - Tavoitetaso 1,44

2. Tekijät testaavat vuorotellen toistensa työt, nykytila 0,56 - Tavoitetaso 1,00

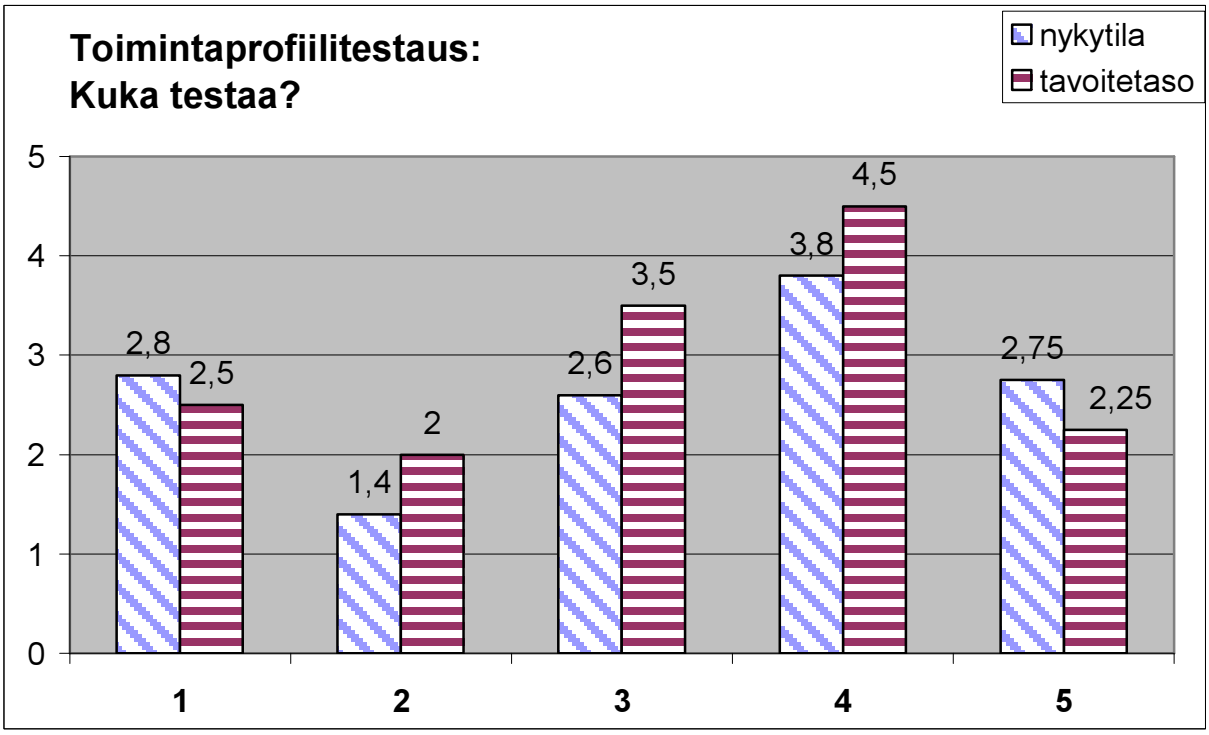
3. Toimittajan testausryhmä, nykytila 3,00 - Tavoitetaso 3,67

4. Asiakkaan testausryhmä, nykytila 3,89 - Tavoitetaso 4,75

5. Loppukäyttäjä, nykytila 1,89 - Tavoitetaso 2,00

8.2.6 Toimintaprofiilitestaus

Toimintaprofiili (tilastollinen) -testaus suoritetaan useimmiten asiakkaan testausryhmän toimesta. Kasvutarve osoittaa sen olevan tulevaisuudessakin suosituin (Kuva 27). Käytettävyydeltään se oli myös selvästi paras.



Kuva 27. Toimintaprofiilitestauksen suorittajat [Kär03]

1. Ohjelmoija itse, nykytila 2,80 - Tavoitetaso 2,50

2. Työpari vuorotellen toistensa työt, nykytila 1,40 - Tavoitetaso 2,00

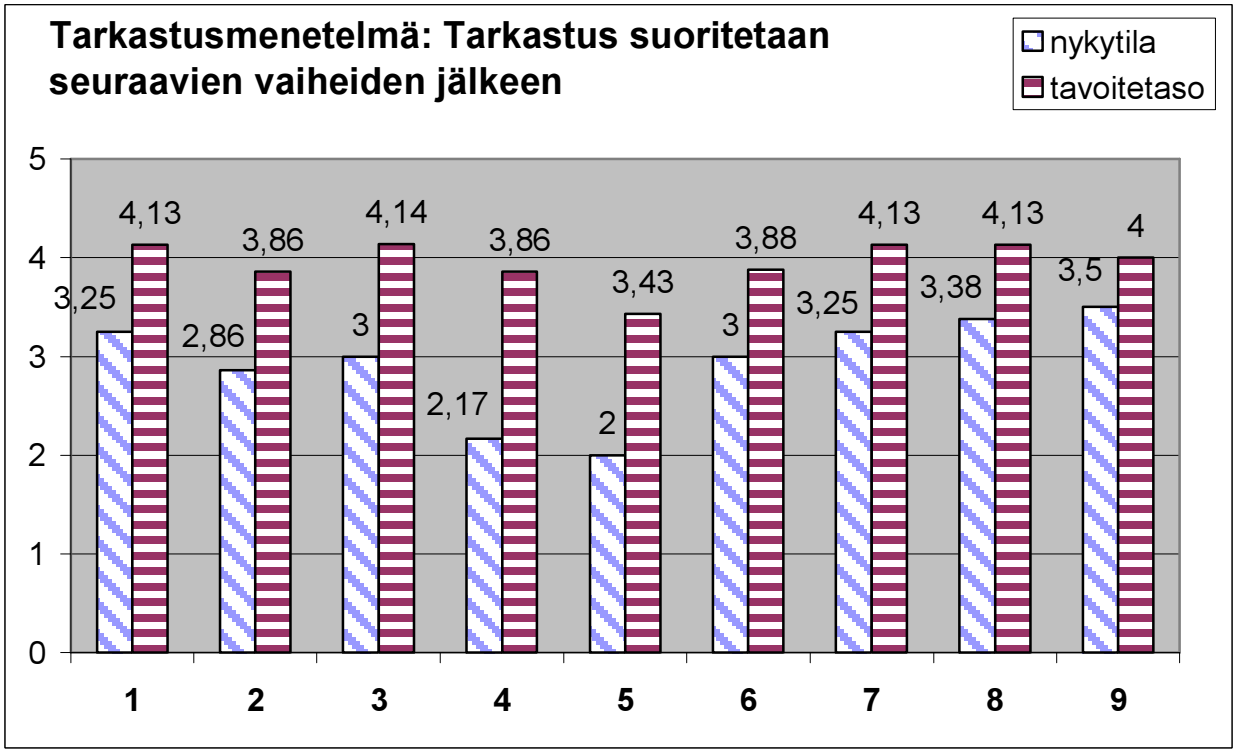
3. Toimittajan testausryhmä, nykytila 2,60 - Tavoitetaso 3,50

4. Asiakkaan testausryhmä, nykytila 3,80 - Tavoitetaso 4,50

5. Loppukäyttäjä, nykytila 2,75 - Tavoitetaso 2,25

8.2.7 Tarkastusmenetelmä

Tarkastusmenetelmää käytetään nykyisin jokaisen V-mallin vaiheen jälkeen ja jatkossa pyritään käyttöä lisäämään kaikissa vaiheissa (Kuva 28). Tarkastusta suorittavat eniten tarkastusryhmät, joiden rooli katsotaan tärkeimmäksi myös tulevaisuudessa (Kuva 29, sivu 78). Käytettävyydeltään parhaana ja yhtä hyvänä pidetään sekä tarkastusryhmän että asiakkaan suorittamaa tarkastusta.



Kuva 28. Tarkastuksen suorittaminen eri vaiheiden jälkeen [Kär03]

1. Vaatimusten määrittely, nykytila 3,25 - Tavoitetaso 4,13

2. Arkkitehtuurisuunnittelu, nykytila 2,86 - Tavoitetaso 3,86

3. Ohjelmistosuunnittelu, nykytila 3,00 - Tavoitetaso 4,14

4. Koodaus, nykytila 2,17 - Tavoitetaso 3,86

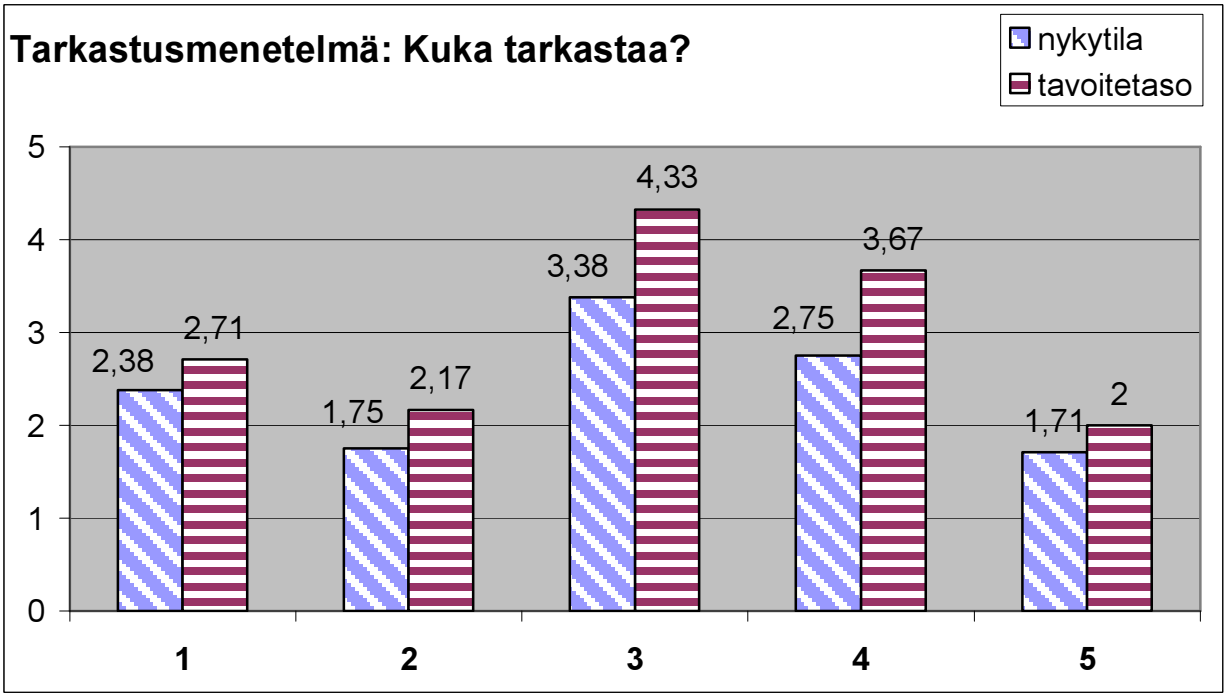
5. Yksikkötestaus, nykytila 2,00 - Tavoitetaso 3,43

6. Integrointitestaus, nykytila 3,00 - Tavoitetaso 3,88

7. Toiminnallisuuden testaus, nykytila 3,25 - Tavoitetaso 4,13

8. Järjestelmätestaus, nykytila 3,38 - Tavoitetaso 4,13

9. Hyväksymistestaus, nykytila 3,50 - Tavoitetaso 4,00



Kuva 29. Tarkastuksen suorittajat [Kär03]

1. Suunnittelija itse, nykytila 2,38 - Tavoitetaso 2,71

2. Työpari vuorotellen toistensa työt, nykytila 1, 75 - Tavoitetaso 2,17

3. Tarkastusryhmä, nykytila 3,38 - Tavoitetaso 4,33

4. Asiakas, nykytila 2,75 - Tavoitetaso 3,67

5. Loppukäyttäjä, nykytila 1,71 - Tavoitetaso 2,00

8.2.8 Automatisointityökalujen käyttö

Tilastollisessa analyysissä mukana olleista yhdeksästä vastaajasta neljä ilmoitti käyttävänsä jotakin työkalua. Kahdella vastaajista oli käytössä kaksi erilaista ja kahdella yksi työkalu. Kaikkiaan erilaisia työkaluja löytyi kuusi kappaletta. Keskimääräinen työkalun käyttöaika oli kaksi vuotta. Käytön oppimiseen oli kulunut aikaa hieman vajaat kolme viikkoa. Työkalun käytettävyyttä, tulosten tulkintaa ja sijoituksen kannattavuutta pidettiin hyvinä. Ohjelmistojen toimivuuden paranemista ja työkalun käyttöastetta pidettiin keskinkertaisena (Kuva 30).

'Käytettävyyden', 'tulosten tulkinnan' ja 'sijoituksen kannattavuuden' asteikko: 0 = kelvoton 1 = menettelee 2 = hyvä 3 = erinomainen - = en osaa sanoa tai puuttuva tieto	'Käyttöasteen' ja 'ohjelmistojen toimivuuden parantuminen työkalujen myötä' asteikko: 0 = ei ollenkaan 1 = jonkin verran 2 = usein 3 = aina - = en osaa sanoa tai puuttuva tieto

Työkalun nimi	käyttö-aika vuosina	oppi-misaika viikkoi-na	käytet-tävyys	tulos-ten-tulkin-ta	sijoituk-sen kan-nattavuus	käyttö-aste	toimivuu-den parantu-minen
WebStress	1	0-1	3	2	3	1	3
Rational TeamTest	4	-	1	1	(en osaa sanoa)	2	(en osaa sanoa)
Compuware QARun	2	-	2	2	(en osaa sanoa)	1	(en osaa sanoa)
TestDirector	2	4	2	2	2	2	2
WinRunner	2	3	2	2	2	2	2
Jtest	1	3	1	3	2	1	3
Keskiarvot	2,00	2,75	1,83	2,00	2,25	1,5	1,5

Kuva 30. Automatisoitujen työkalujen nykyinen käyttö [Kär03]

8.2.9 Kyselyn vastausten analysointi

Tarkasteltaessa kyselyn tuloksia kokonaisuutena oli ilahduttavaa, että lähes jokaisella osa-alueella oli pyrkimys parempaan tulevaisuudessa, vaikka nykytilannekaan ei suinkaan ollut huono.

Yksityiskohtaisemmassa tarkastelussa kiinnitti ensimmäiseksi huomiota se, että moduulitestauksen suosituin menetelmä oli mustalaatikkotestaus, vaikka kaikki teoriakirjat pitävät sitä käyttökelpoisena aikaisintaan integrointitestausvaiheessa ja siitä ylöspäin V-mallin oikeassa haarassa. Saattaa olla, että vääristymää aiheutti virheiden jäljityksen sisällyttäminen mustalaatikkoon koska virheiden jäljitys ei ole testausta. Erittäin positiivista oli halu lisätä ohjelmointiparien ja toimittajan testausryhmän suorittamaa testausta, sekä ekvivalenssiluokituksen tunteminen ja käytön lisääminen tulevaisuudessa. Lasilaatikkotestauksessa tunnettiin hyvin kontrollivirtaan ja tietovirtaan perustuvat menetelmät nimeltä ja kaikkien käytön lisäämistä pidettiin tarpeellisena. Teorioitten mukaanhan lasilaatikko on nimenomaan moduulitestauksen menetelmä.

Integrointitestauksessa oli havaittavissa voimakas tarve kehittää vuorottelutestausta tekijöiden kesken, sekä toimittajan ja asiakkaan testausryhmän työtä. Käytetyissä menetelmissä kehityssuunta oli aivan oikea. Nykyisin suurimmassa käytössä olevan kertarysäyksen osuutta haluttiin pienentää, ja vastaavasti jäsentävän, kokoavan ja kerrosvoileipä menetelmän käyttöä kasvattaa. Kertarysäyshän soveltuu hyvin vain erittäin pienien ohjelmistojen testaukseen.

Toiminnallisuus-, järjestelmä- ja hyväksymistestauksissa suunta oli aivan oikea, kun haluttiin kehittää toimittajan, asiakkaan ja tekijöiden vuorottelun rooleja testauksessa. Selvästi havaittava tavoite oli toimittajan testauksen lisääminen.

Miellyttävä yllätys oli tarkastusmenetelmän laaja käyttö läpi koko V-mallin. Lisäksi menetelmän käyttöä haluttiin lisätä kaikilla osa-alueilla.

Automatisointityökalujen käyttöä olisi syytä lisätä. Nyt vastanneista vain alle puolella oli jokin työkalu käytössä. Huomion arvoista oli, että kokemukset olivat enemmän myönteisiä kuin kielteisiä.

9 POHDINTA

Kun katsotaan, mikä oli Wilkesin [Wil85] havainto vuonna 1949, voidaan todeta, että ongelmia on yhä olemassa, mutta edistystä on tapahtunut.

Tutkielman alkuosassa esiteltiin testauksen historian lisäksi testauksen vaihejakoa ja erilaisia testausmenetelmiä. Joukossa oli muutamia vähemmän tunnettuja menetelmiäkin. Esimerkiksi muuttumattomuus, tutkiva ja äärimmilleen viety testaus.

Automatisointiosuudessa tarkasteltiin testausta kahdesta näkökulmasta. Automatisoidun testauksen elinkaaren metodiikka kävi vaiheittain läpi prosessin päätöksestä automatisoida testaus testauksen toteuttamiseen ja prosessin arviointiin. Toinen näkökulma oli enemmän kokemusperäinen. Siinä esiteltiin tunnettuja automatisoinnin etuja ja ongelmia ja ratkaisuehdotuksia ongelmiin. Tässä osassa havaittiin, että on mahdollista automatisoida myös testauksen suunnittelu, suoritus ja tulosten vertailu.

Havaittiin myös, että työvälineitä löytyy jokaiseen V-mallin (Kuva 1, sivu 11) vaiheeseen. Tutkielmassa on esitelty eräs mahdollisuus niiden luokittelemiseen. Tämä ei ole ainut mahdollinen, sillä välineillä on niin paljon päällekkäisiä ominaisuuksia, että on luokittelijasta kiinni, mitä ominaisuutta hän painottaa ja mihin luokkaan välineen sijoittaa.

Mielenkiintoista oli tutustua erilaisiin automatisointiyrityksiin. Näissä havaittiin, että ensimmäisellä kerralla puuttui yleensä kokemusta, ja kommunikointi johdon ja automatisoijien välillä ei pelannut. Varsin merkittävä huomio oli myös se, että yritettiin automatisoida kerralla liian paljon: Yleensä ensimmäinen yritys epäonnistui.

Yrityskyselynä suoritettu kokeellinen osuus antoi viittauksia siihen, että oikeaan suuntaan ollaan menossa (Kappale 8.2.9).

Tutkielmassa esitettyjen tutkimusten perusteella voin todeta, että paljon ovat testaus ja sen menetelmät kehittyneet vuosien varrella, mutta kyllä tekemistäkin vielä riittää. Toivottavaa olisi, että testaus saisi sille kuuluvan arvostetun aseman yhtenä ohjelmistotuotannon osana. Poikkeuksetta kaikissa löytämissäni lähteissä testauksen

automatisointia pidettiin erittäin vaikeana ja haastavana tehtävänä, jossa oli suuret mahdollisuudet epäonnistua. Kuitenkin hyvin toteutettuna on automatisointi kannattavaa ja ehdoton edellytys testauksen tulevalle kehitykselle.

Lopuksi haluan todeta hieman Jörn Donneria mukaillen:

”Testaaminen kannattaa aina.”

LÄHTEET

[AdB82] Adrion, W. R., Branstad, M. A., Cherniavsky, J. C.: *Validation, Verification and Testing of Computer Software*. ACM Computing Surveys, Volume14, Issue2, 159-192, June 1982.

[Aml99] Amland, S.: *Test automation failures: lessons to be learned*. In Fewster, M. (Ed.), Graham, D.: *Software Test Automation. Effective use of test execution tools.* ACM Press, New York, 1999.

[Asb00] Asböck, S.: *Load testing for eConfidence*. Segue Software, Inc., USA, October 2000.

[Bac01] Bach, J.*: James Bach on Explaining Testing to Them*. The Software Testing & Quality Engineering Magazine, Volume 3, Issue 6, 28-32, November/December 2001.

[Bac99] Bach, J.: *Test Automation Snake Oil*. V2.1 6/13/99, haettu 13.02.02, URL: http://www.satisfice.com/articles/test_automation_snake_oil.pdf.

[Bei84] Beizer, B.: *Software System Testing and Quality Assurance*. Van Nostrand Reinhold Company, USA, 1984.

[BuO00] Buy, U., Orso, A., Pezze, M.: *Automated Testing of Classes*. Proceedings of ISSTA, Portland, Oregon, 2000.

[McC76] McCabe, T.: *A Complexity Measure*. IEEE Transactions on Software Engineering, SE-2, 308-320, 1976.

[Chi99] Chillarege, R.: *Software Testing Best Practices*. Center for Software Engineering, IBM Research, 1999, haettu 20.02.2002, http://www.chillarege.com/authwork/TestingBestPractice.pdf.

[DaD72] Dahl, O. J., Dijkstra, E. W., Hoare C. A. R.: *Structured Programming*. Academic Press, 1972.

[DuR99] Dustin, E., Rashka, J., Paul, J.: *Automated Software Testing. Introduction, Management and Performance*. Addison-Wesley, 1999.

[Dus99] Dustin, E.: *Lessons in Test Automation*. The Software Testing & Quality Engineering Magazine, Volume 1, Issue 5, 16-23, September/October 1999.

[FeG99] Fewster, M., Graham, D.: *Software Test Automation. Effective use of test execution tools*. ACM Press, New York, 1999.

[GrH00] Gronau, I., Hartman, A., Kirshin, A., Nagin, K., Olvovsky, S.: *A Methodology and Architecture for Automated Software Testing*. IBM Research Laboratory in Haifa, Technical Report, 2000, haettu 12.02.2002, URL: http://www.geocities.com/model_based_testing/gtcbmanda.pdf.

[Hal77] Halstead, M.: *Elements of Software Science*. Elsevier North-Holland, New York, 1977.

[HaM01] Haikala, I., Märijärvi, J.*: Ohjelmistotuotanto*. 7. painos, Talentum Media Oy, Helsinki, 2001.

[Hay95] Hayes, L.: *Extracts from The Automated Testing Handbook*. In Fewster, M. (Ed.), Graham, D.: *Software Test Automation. Effective Use of test execution* tools. ACM Press, New York, 1999.

[HeH84] Hennell, M. A., Hedley, D., Riddell, I. J.: *Assessing a Class of Software Tools*. Proceedings of the IEEE 7th International Conference on Software Engineering, IEEE, 266-277, March 1984.

[Hen00] Hendricsson, E.: *Build It or Buy It*? The Software Testing & Quality Engineering Magazine, May/June 2000, haettu 7.02.2002, URL: http://www.qualitytree.com/feature/biobi.pdf.

[Hen01] Hendricsson, E.*: Better Testing – Worse Quality?* Presentation in International Conference On Software Management & Applications of Software Measurement, February 12-16, 2001, San Diego, CA, USA, haettu 7.02.2002, URL: http://www.qualitytree.com/feature/btwq.pdf.

[Hen98] Hendricsson, E.: *The Differences Between Test Automation Success and Failure*. Presentation in the International Conference On Software Testing Analysis & Review, October 26-30, San Diego, CA, USA, haettu 7.02.2002, URL: http://www.qualitytree.com/feature/dbtasaf.pdf.

[Hen99a] Hendricsson, E.: *Evaluating Tools.* The Software Testing & Quality Engineering Magazine, January/February 1999, haettu 7.02.2002, URL: http://www.qualitytree.com/feature/et.pdf.

[Hen99b] Hendricsson, E.: *Making the Right Choice.* The Software Testing & Quality Engineering Magazine, May/June 1999, haettu 7.02.2002, URL: http://www.qualitytree.com/feature/mtrc.pdf.

[Het85] Hetzel, W. C.: *The Complete Guide to Software Testing*. Collins, 1985.

[Jef99] Jeffries, R. E.: *Extreme Testing.* The Software Testing & Quality Engineering Magazine, Volume 1, Issue 2, March/April, 1999, 22-27.

[Jän03] Jäntti M.: *Testitapausten suunnittelu UML-mallinnuksen avulla*. Pro gradu -tutkielma. Kuopion Yliopisto, Tietojenkäsittelytieteen laitos, elokuu 2003, URL: http://www.cs.uku.fi/research/Teho/julkaisut.html.

[KaB02] Kaner, C., Bach, J., Pettichord, B.: *Lessons Learned in Software Testing.* John Wiley & Sons, Inc., New York, 2002.

[KaF99] Kaner, C., Falk, J., Nquyen, H. Q.: *Testing Computer Software*. Second Edition, John Wiley & Sons, Inc., USA, 1999.

[KaG96] Kasik, D. J., George H G.: *Toward Automatic Generation of Novice User Test Scripts.* Conference Proceedings on Human factors in computing systems: Common Ground, pages 244--251, New York, ACM Press,13-18 April 1996.

[Kan96] Kaner, C.: *Software Negligence and Testing Coverage*. Haettu 12.02.2002, URL: http://www.kaner.com/negotiate.htm.

[Kan97] Kaner, C.: *Improving the Maintainability of Automated Test Suites*. Paper Presented at Quality Week, 1997, haettu 12.02.2002, URL: http://www.testing.com/writings/classic/mistakes.html.

[Kau96] Kautto, T.: *Ohjelmistotestaus ja siinä käytettävät työkalut*. Ohjelmistotekniikan seminaariesitelmä, Tietojenkäsittelytieteen laitos, Jyväskylän Yliopisto, tammikuu 1996.

[Kit95] Kit, E.: *Software Testing in The Real World*, Addison Wesley, 1995.

[Kär03] Kärkkäinen Tarja-Liisa.:*PlugIT-yrityskyselyn analysointi*. Kuopion yliopisto, 2003.

[Len01] Lengel, K.*: Look Before You Test*. The Software Testing & Quality Engineering Magazine, Volume 3, Issue 2, 60-64, March/April 2001.

[MaB00] Marick, B., Bach, J., Kaner, C.: *A Manager's Guide to Evaluating Test Suites*. Haettu 4.02.2002, URL: http://www.testing.com/writings/evaluating-test-suites-paper.pdf.

[Mar95] Marick, B.: *The Craft of Software Testing*. PTR Prentice Hall, Englewood Cliffs, New Jersey, USA, 1995.

[Mar97] Marick, B.: *How to Misuse Code Coverage*. Haettu 12.02.2002, URL: http://www.testing.com/writings/coverage.pdf.

[Mar98] Marick, B.: *When Should a Test Be Automated*. Haettu 12.02.2002, URL: http://www.testing.com/writings/automate.pdf.

[Mye79] Myers, G.: *The Art of Software Testing*. John Wiley & Sons, New York,1979.

[Paa00] Paakki, J.: *581361-6 Ohjelmistojen testaus*. Lecture notes, University of Helsinki Department of Computer Science, Autumn 2000.

[Par03] Partanen A.*: Olio-ohjelmien testaus*. Pro gradu -tutkielma, Kuopion yliopisto, Tietojenkäsittelytieteen laitos, heinäkuu 2003, URL: http://www.cs.uku.fi/research/Teho/julkaisut.html.

[Pet00] Pettichord, B.: *Testers and Developers Think Differently*. The Software Testing & Quality Engineering Magazine, Volume 2, Issue 1, 42-47, January/February 2000.

[Pet01a] Pettichord, B.: *Seven Steps to Test Automation Success*. Originally presented at STAR West, San Jose, November 1999, version of 26 June 2001, haettu 20.02.2002, http://www.io.com/~wazmo/papers/seven_steps.html

[Pet01b] Pettichord, B.: *Success with Test Automation*. Version of 28 June 2001, Originally presented at Quality Week, San Francisco, May 1996, haettu 12.02.2002, URL: http://www.io.com/~wazmo/succpap.htm.

[Plu03] *PlugIT –projektin esittely*, haettu 24.04.2003. URL: http://www.uku.fi/atkk/plugit/

[Poh02] Pohjolainen, P.: *Software Testing Tools*. Erikoistyö, Tietojenkäsittelytieteen ja sovelletun matematiikan laitos, Kuopion Yliopisto, helmikuu 2002, URL: http://www.cs.uku.fi/research/Teho/julkaisut.html.

[PoS92] Poston, R. M., Sexton, M. P.: *Evaluating and Selecting Testing Tools*, IEEE Software, Volume 9, Issue 3, 33-42, May 1992.

[Rob00] Robinson, H.: *Intelligent Test Automation*. The Software Testing & Quality Engineering Magazine, September/October 2000, haettu 12.02.2002, URL: http://www.geocities.com/harry_robinson_testing/robinson.pdf.

[Rol99] Rothman, J., Lawrence, B.: *Testing in the Dark*. The Software Testing & Quality Engineering Magazine, Volume 1, Issue 2, 34-40, March/April 1999.

[Rop94] Roper, M.: *Software testing*. McGraw-Hill, 1994.

[Rot02] Rothman J.: *Release Criteria: Is This Software Done?* The Software Testing & Quality Engineering Magazine, January/February 2002, haettu 20.08.2002, URL: http://www.stqemagazine.com/index.asp?frame=CORE&content=FEATURED

[Sma99] Smale, A.: *Test automation experience at Microsoft*. In Fewster, M. (Ed.), Graham, D.: *Software Test Automation. Effective use of test execution tools*. ACM Press, New York, 1999.

[Ter00] Tervonen, I.*: Katselmointi ja testaus*. Lecture notes in University of Oulu, 2000.

[Ter01] TerMaat, P.: *Adventures in Automated Testing*. The Software Testing & Quality Engineering Magazine, Volume 3, Issue 3, 22-28, May/June 2001.

[ThH02] Thomas, D., Hunt, A.: *Learning to Love Unit Testing*. The Software Testing & Quality Engineering Magazine, January/February 2002, haettu 12.02.2002, URL: http://www.stqemagazine.com/featured.asp?stamp=1129125440.

[Vir02] Virkanen H.: *Ohjelmistojen testaus ja virheen jäljitys*. Pro gradu -tutkielma, Kuopion yliopisto, Tietojenkäsittelytieteen laitos, joulukuu 2002, URL: http://www.cs.uku.fi/research/Teho/julkaisut.html.

[Wei01] Weinberg, G. M.: *Idempotency: Design for Testability*. The Software Testing & Quality Engineering Magazine, Volume 3, Issue 5, 16-18, September/October 2001.

[Whi00] Whittaker, J. A.: *What Is Software Testing? And Why Is It So Hard?* IEEE SOFTWARE, January/February 2000, haettu 20.02.2002, URL: http://www.computer.org/software/so2000/pdf/s1070.pdf.

[Wil85] Wilkes, M.: *Memoirs of a Computer Pioneer*. MIT Press, 1985.

[Zam98] Zambelich, K.: *Totally Data-Driven Automated Testing*. A White Paper, haettu 12.02.2002, URL: http://www.sqa-test.com/w_paper1.html.

LIITE 1

TESTAUS JA TARKASTUS

TAUSTATIEDOT

Organisaationne/yrityksenne koko: ……henkilöä

Yksikkönne koko: ……henkilöä

Yksikkönne tyyppi:

1 erikoissairaanhoito
2 perusterveydenhuolto
3 sairaalan atk-osasto
4 ohjelmistoyritys
5 muu, mikä?……

Koulutuksenne: ……

Kokemuksenne eri tehtävissä:

1 sovellussuunnittelu	…… vuotta
2 sovelluskehitys	…… vuotta
3 testaus	…… vuotta
4 projektinohjaus (esim. projektipäällikkö)	…… vuotta
5 hoitotyö perusterveydenhuollossa	…… vuotta
6 hoitotyö erikoisterveydenhuollossa	…… vuotta
7 lääkäri	…… vuotta
8 osastosihteeri	…… vuotta
9 hallinnolliset johtotehtävät	…… vuotta
10 muu:	
……	…… vuotta

Nykyinen työtehtävänne:……

Lyhyt kuvaus testausmenetelmistä:

Moduulitestaus l. yksikkötestaus on yksittäisen ohjelmamoduulin testaamista.
- Mustalaatikossa ei ohjelmakoodi ole nähtävissä. Testitapaukset suunnitellaan vaatimusmäärittelyjen pohjalta.
- Lasilaatikossa ohjelmakoodi on nähtävissä ja testitapaukset laaditaan siten, että koodi tulisi mahdollisimman kattavasti läpikäydyksi.
- Harmaa laatikko on kahden edellisen yhdistelmä, eli hyödynnetään sekä vaatimuksia, että ohjelmakoodia.

Integrointitestauksessa testataan toisiinsa liittyvien ohjelmamoduulien toimivuus yhdessä.
- Kertarysäyksessä linkitetään kaikki moduulit kerralla yhteen ja testataan yhtenä kokonaisuutena.
- Jäsentävässä menetelmässä lähdetään liikkeelle ensimmäisestä (ylimmästä) moduulista testaten se. Testaus etenee kutsuttujen moduulien mukaisesti alaspäin.
- Kokoavassa menetelmässä lähdetään alimman tason moduuleista ja edetään ylöspäin kutsuvien moduulien mukaan.
- Kerrosvoileipämenetelmässä edetään yhtä aikaa ylhäältä alas ja alhaalta ylös. Se on siis jäsentävän ja kokoavan sekoitus.

Toiminnallisuustestaus l. suorituskykytestaus mittaa, kuinka hyvin systeemi suoriutuu vaatimusmäärittelyissä sille asetetuista tavoitteista.

Järjestelmätestauksessa testataan koko systeemi (kaikki moduulit) yhtenä kokonaisuutena.

Hyväksymistestaus on asiakkaan ja valmistajan yhdessä suorittama viimeinen tarkistus ennen järjestelmän käyttöönottoa.

Järjestelmän toimintaprofiilin tutkimisen perusteella keskitytään testaamaan niitä osa-alueita, joita todennäköisesti käytetään eniten.

Ekvivalenssiluokitus on toiminnallisen testauksen (black-box) menetelmä, jossa jokainen arvoalue jaetaan ekvivalenssiluokkiin (=kelvollisten syötteiden osajoukot sekä epäkelpojen syötteiden joukko). Näille luokille pätee, että jokainen tiettyyn ekvivalenssiluokkaan kuuluva arvo käyttäytyy testauksen kannalta ‘samalla tavalla’. Jokaisesta luokasta otetaan yksi tai useampi edustaja testitapauksiksi. Ekvivalenssiluokituksella vähennetään tarvittavien testitapausten määrää.

Tarkastusmenetelmä on kokoustekniikka, jota voidaan käyttää minkä tahansa kirjallisen tuotteen laadun varmistamiseen V-mallin määrittely- ja suunnitteluvaiheista aina ohjelmakoodiin saakka. Menetelmän vaiheet ovat suunnittelu, esittely, valmistautuminen, tarkastustilaisuus, korjaukset ja seuranta.

KYSYMYS:

Mitkä seuraavista menetelmistä ja kuvaustavoista ovat käytössä yksikössänne?

Arvioikaa ensimmäisessä sarakkeessa esitetyn menetelmän tämän hetken käyttötilannetta (0= Ei koskaan, 1= Harvoin, 2= Joskus, 3= Usein, 4= Lähes aina, 5= Aina, 6= En osaa sanoa) sekä käytettävyyttä (0=kelvoton, 1=menettelee, 2=hyvä, 3=erinomainen) ja ympyröikää mielestänne sopivin vaihtoehto. Merkitkää myös menetelmän käytön tavoitetaso ympyröimällä se numero, jossa toivoisitte olevanne kolmen vuoden päästä (asteikko 0-6). Huomatkaa, että käytössä on tavallisesti useampia menetelmiä.

Esimerkiksi

Testaus ja tarkastus	Ei koskaan	Harvoin	Joskus	Usein	Lähes aina	Aina	En osaa sanoa	Tavoitetaso	Käytettävyys: Kelvoton	Menettelee	Hyvä	Erinomainen
Moduulitestaus												
Kuka testaa ?												
- ohjelmoija itse	0	1	2	(3)	4	5	6	0 1 (2) 3 4 5 6	0	(1)	2	3
- ohjelmoijat testaavat vuorotellen toistensa moduulit (esim. työpareina)	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3

VASTAUSLOMAKE

Testaus ja tarkastus	Ei koskaan	Harvoin	Joskus	Usein	Lähes aina	Aina	En osaa sanoa	Tavoitetaso	Käytettävyys			
									Kelvoton	Menettelee	Hyvä	Erin-omainen
Moduulitestaus												
Kuka testaa ?												
- ohjelmoija itse	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- ohjelmoijat testaavat vuorotellen toistensa moduulit (esim. työpareina)	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- toimittajan testausryhmä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- asiakkaan testausryhmä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- loppukäyttäjä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- joku muu, kuka?	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Testitapaukset saadaan												
- käyttötapauksista	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- aktiviteettikaaviosta	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- tilakaaviosta	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- muualta, mistä?	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3

Testaus ja tarkastus	Ei koskaan	Harvoin	Joskus	Usein	Lähes aina	Aina	En osaa sanoa	Tavoitetaso	Käytettävyys			
									Kelvoton	Menettelee	Hyvä	Erin-omainen
Mustalaatikko (black-box)	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Testaus suoritetaan virheitä jäljittämällä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Testitapausten määrittelyssä käytetään ekvivalenssiluokitusta	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Lasilaatikko (white-box)	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Mitä testataan (kontrollivirtaan liittyen)?												
- lausekattavuus = kaikkien lauseiden läpikäynti	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- päätöskattavuus = kaikkien ohjelman vuokaavion kaarien läpikäynti	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- ehtokattavuus = kaikkien ehtolauseiden ’tosi’ ja ’epätosi’ arvojen läpikäynti	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- moniehtokattavuus = ehtolauseiden kaikkien ’tosi’ ja ’epätosi’ arvojen eri kombinaatioiden läpikäynti	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- polkukattavuus = mahdollisimman monien ohjelman eri polkujen läpikäynti	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Mitä testataan (tietovirtaan liittyen)?												
- kaikki määrittelyt saavuttavat jonkin käyttönsä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3

Testaus ja tarkastus	Ei koskaan	Harvoin	Joskus	Usein	Lähes aina	Aina	En osaa sanoa	Tavoitetaso	Käytettävyys			
									Kelvoton	Menettelee	Hyvä	Erin-omainen
- kaikki määrittelyt saavuttavat kaikki käyttönsä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- kaikki määrittelyt/käytöt saavuttavat toisensa kaikkia mahdollisia polkuja pitkin	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Harmaa laatikko (gray-box)	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Joku muu menetelmä, mikä? Kuvailkaa menetelmä:	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3

Testaus ja tarkastus	Ei koskaan	Harvoin	Joskus	Usein	Lähes aina	Aina	En osaa sanoa	Tavoitetaso	Käytettävyys			
									Kelvoton	Menettelee	Hyvä	Erin-omainen
Mitä ongelmia teillä on ollut moduulitestauksessa?												
Mitä moduulitestauksessa pitäisi kehittää?												
Integrointitestaus												
Kuka testaa?												
- ohjelmoija itse	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- tekijät testaavat vuorotellen toistensa työt	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3

Testaus ja tarkastus	Ei koskaan	Harvoin	Joskus	Usein	Lähes aina	Aina	En osaa sanoa	Tavoitetaso	Käytettävyys			
									Kelvoton	Menettelee	Hyvä	Erin-omainen
- toimittajan testausryhmä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- asiakkaan testausryhmä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- loppukäyttäjä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- joku muu, kuka?	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Käytössä olevat menetelmät												
- mustalaatikko	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- virheenjäljitys (debuggaus)	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- kertarysäys (big-bang)	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- jäsentävä (top-down)	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- kokoava (bottom-up)	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- kerrosvoileipä (sandwich)	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- joku muu, mikä? Kuvailkaa menetelmä:	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Toiminnallisuustestaus												
Kuka testaa?												
- ohjelmoija itse	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3

Testaus ja tarkastus	Ei koskaan	Harvoin	Joskus	Usein	Lähes aina	Aina	En osaa sanoa	Tavoitetaso	Käytettävyys			
									Kelvoton	Menettelee	Hyvä	Erin-omainen
- tekijät testaavat vuorotellen toistensa työt	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- toimittajan testausryhmä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- asiakkaan testausryhmä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- loppukäyttäjä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- joku muu, kuka?	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Käytössä olevat menetelmät												
- mustalaatikko	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- virheenjäljitys (debuggaus)	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- joku muu, mikä? Kuvailkaa menetelmä:	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Järjestelmätestaus												
Kuka testaa?												
- ohjelmoija itse	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- tekijät testaavat vuorotellen toistensa työt	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- toimittajan testausryhmä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3

Testaus ja tarkastus	Ei koskaan	Harvoin	Joskus	Usein	Lähes aina	Aina	En osaa sanoa	Tavoitetaso	Käytettävyys			
									Kelvoton	Menettelee	Hyvä	Erin-omainen
- asiakkaan testausryhmä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- loppukäyttäjä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- joku muu, kuka?	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Käytössä olevat menetelmät												
- mustalaatikko	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- virheenjäljitys (debuggaus)	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- joku muu, mikä? Kuvailkaa menetelmä:	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Hyväksymistestaus												
Kuka testaa?												
- ohjelmoija itse	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- tekijät testaavat vuorotellen toistensa työt	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- toimittajan testausryhmä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- asiakkaan testausryhmä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- loppukäyttäjä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3

Testaus ja tarkastus	Ei koskaan	Harvoin	Joskus	Usein	Lähes aina	Aina	En osaa sanoa	Tavoitetaso	Käytettävyys			
									Kelvoton	Menettelee	Hyvä	Erin-omainen
- joku muu, kuka?	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Käytössä olevat menetelmät												
- mustalaatikko	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- virheenjäljitys (debuggaus)	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- joku muu, mikä? Kuvailkaa menetelmä:	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Ennalta laaditut hyväksymiskriteerit toteutuvat testauksessa	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Toimintaprofiilitestaus												
Kuka testaa?												
- ohjelmoija itse	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- työpari vuorotellen toistensa työt	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- toimittajan testausryhmä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- asiakkaan testausryhmä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- loppukäyttäjä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3

Testaus ja tarkastus	Ei koskaan	Harvoin	Joskus	Usein	Lähes aina	Aina	En osaa sanoa	Tavoitetaso	Käytettävyys			
									Kelvoton	Menettelee	Hyvä	Erin-omainen
- joku muu, kuka?	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Tarkastusmenetelmä												
Tarkastus suoritetaan seuraavien vaiheiden jälkeen												
- Vaatimusten määrittely	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- Arkkitehtuurisuunnittelu	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- Ohjelmistosuunnittelu	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- Koodaus	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- Yksikkötestaus	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- Integrointitestaus	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- Toiminnallisuuden testaus	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- Järjestelmätestaus	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- Hyväksymistestaus	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Kuka tarkastaa?												
- suunnittelija itse	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- työpari vuorotellen toistensa työt	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3

Testaus ja tarkastus	Ei koskaan	Harvoin	Joskus	Usein	Lähes aina	Aina	En osaa sanoa	Tavoitetaso	Käytettävyys			
									Kelvoton	Menettelee	Hyvä	Erin-omainen
- tarkastusryhmä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- asiakas	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
Kuka tarkastaa (jatkuu)?												
- loppukäyttäjä	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3
- joku muu, kuka?	0	1	2	3	4	5	6	0 1 2 3 4 5 6	0	1	2	3

TEHTÄVÄ: Arvioikaa käytössänne olevia automatisoidun testauksen työkaluja.

Työkalun nimi	Kuinka monta vuotta käytetty	Käytön oppiminen kesti (viikkoina)	0= Kelvoton, 1= Menettelee, 2= Hyvä, 3= Erinomainen, 4= En osaa sanoa			0= Ei ollenkaan, 1= Jonkin verran, 2= Usein 3= Aina, 4= En osaa sanoa	
			Käytettävyys	Tulosten tulkinta	Sijoituksen kannattavuus	Käyttöaste	Ohjelmistojen toimivuus on parantunut työkalujen käytön myötä
			0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4
			0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4
			0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4
			0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4
			0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4

Mitä kehitystä toivoisitte tapahtuvan testauksen työkaluissa?

Testauksen ongelmia:

Testauksen kehittämiskohteita ja –ideoita:

Tarkastuksen ongelmia:

Tarkastuksen kehittämiskohteita ja –ideoita:

Kertokaa mielenkiintoisia kokemuksianne testauksesta. (Ilmaiskaa myös, voiko niitä julkaista anonyymisti):

Viesti PluglT-hankkeelle: