Testausprojektien työmääräarviointi ja työmäärään vaikuttavat tekijät

Katja Kuusisto

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Erkki Mäkinen
Toukokuu 2010

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Katja Kuusisto
Pro gradu -tutkielma, 59 sivua
Toukokuu 2010

Tutkielma käsittelee ohjelmistoprojektista eriytetyn testausprojektin työmääräarviointia. Tutkielmassa arvioidaan koko ohjelmistoprojektin kattavien työmääräarviomallien soveltumista testausprojektien työmääräarviointiin. Lisäksi perehdytään joihinkin testauksen työmäärään huomattavasti vaikuttaviin ohjelmistoprosesseihin ja -tekniikoihin.

Testauksen automatisointi on eräs keino vaikuttaa merkittävästi testauksen työmäärään. Tutkielmassa pohditaan, milloin automatisointi on kannattavaa ja millä eri tavoilla sitä on mahdollista suorittaa. Lisäksi tutkielmassa esitellään joitakin automatisointiin liittyviä työkaluja.

Avainsanat ja -sanonnat: Ohjelmistotuotanto, ohjelmistotestaus, työmääräarviointi, testauksen automatisointi

Sisällys

1. Johdanto ... 1
2. Testaus ohjelmistoprojekteissa ... 2
   2.1. Suunnittelu ja valvonta ... 2
   2.2. Määrittelyiden katselmoinnit ja kooditarkastukset ... 5
   2.3. Testitapausten luominen ... 7
   2.4. Testitapausten suorittaminen ... 8
   2.5. Prosessin parantaminen ... 10
   2.6. Yhteenveto ohjelmistoprojektin testauksesta ... 11
3. Testausprojektien ulkoistaminen ... 13
   3.1. Syitä testausprojektien ulkoistamiseen ... 13
   3.2. Erilaisten testausprojektien ulkoistamismahdollisuudet ... 14
4. Työmääräarviointi ohjelmisto- ja testausprojekteissa ... 17
   4.1. Ohjelmiston koko ... 17
      4.1.1. Ohjelmiston koon vaikutus työmääräarvioon ... 18
      4.1.2. Ohjelmiston koon käyttäminen työmääräarviomalleissa ... 18
      4.1.3. Ohjelmiston koon merkitys testauksen työmääräarviointiin ... 19
      4.1.4. Ohjelmiston koon soveltuvuus työmääräarvioihin ... 20
   4.2. Ohjelmiston kompleksisuus ... 21
      4.2.1. Ohjelmiston kompleksisuuden vaikutus työmääräarvioon ... 21
      4.2.2. Ohjelmiston kompleksisuuden käyttäminen työmääräarviomalleissa ... 22
      4.2.3. Ohjelmiston kompleksisuuden merkitys testausprojekteihin ... 23
      4.2.4. Ohjelmiston kompleksisuuden soveltuvuus työmääräarvioihin ... 25
5. Muita työmääräarvioissa huomioitavia tekijöitä ... 27
   5.1. Historiatieto ... 27
   5.2. Projektin henkilöstö ... 28
   5.3. Ympäristötekijät ... 29
   5.4. Liiketoimintaosaaminen ... 30
   5.5. Ohjelmistoprosessien vaikutus testauksen työmäärään ... 30
      5.5.1. Ketterän ohjelmistoprosessin erityispiirteet testaukseen ... 31
      5.5.2. Yhtenäistetyn ohjelmistokehitysprosessin erityispiirteet testaukseen ... 33
   5.6. Ohjelmistotekniikan vaikutus testaukseen ... 34
6. Automatisointi ... 37
   6.1. Testitapausten luominen automaattisesti ... 38
   6.2. Automatisoitu yksikkötestaus ... 41
   6.3. Automatisoitu toiminnallinen testaus ... 44
   6.4. Automatisoitu suorituskyky- ja kuormitustestaus ... 48

6.5. Yhteenveto testauksen automatisoinnista 50
7. Yhteenveto 52

Viiteluettelo 54

1. Johdanto

Tietotekniset sovellukset tavoittavat koko ajan enemmän loppukäyttäjiä yhä laajemmin. Tämän pitäisi vaikuttaa siihen, että ohjelmistojen laadun merkitys korostuu. Aiemmin ohjelmistoja käyttivät lähinnä asiantuntijat omassa työssään, mutta 2000-luvulla ohjelmistot ovat levinneet peruskäyttäjien arkeen ja leviäminen jatkuu edelleen. Matkapuhelin on yhä useammin älypuhelin, veroilmoituksen voi täyttää verkossa ja pian kansallinen terveysarkisto tuo omat terveystiedot jokaisen nähtäville. Ohjelmistojen käyttäjäkunnan muuttumisen pitäisi nostaa tuotteen laatukriteerejä, tavalliselle peruskäyttäjälle asti ei saisi päästä virheitä, jotka haittaavat ohjelmiston käyttämistä. [Mitro, 2009].

Eräänä ratkaisuna ohjelmistojen laadunkohotukseen voi toimia viime vuosina yleistynyt testauksen eriyttäminen omaksi projektikseen tuotekehitysprosessin sisältä. Tällöin ohjelmiston testaus voidaan esimerkiksi ostaa kolmannelta osapuolelta ja antaa samalla ulospäin signaali siitä, että ohjelmistotuottaja haluaa itsekin varmistua tuotteensa laadusta. Projektikohtaisesti voidaan määritellä kolmannen osapuolen suorittaman validoinnin laajuus.

Testauksen eriyttäminen omaksi projektiksi tuo uusia haasteita ohjelmistotuotantoon. Perinteinen malli on laskea työmääräarvio koko ohjelmistoprojektille, johon siis sisältyy ohjelmiston koko kehityskaari: määrittely, suunnittelu, toteutus ja testaus. Tätä varten on olemassa monia valmiita malleja ja laskentatapoja. Pelkille testausprojekteille vastaavanlaisia malleja ei löydy, mutta kuitenkin testausprojekteillekin tarvitaan vastaavasti työmääräarvioita aikataulun ja kustannusten arvioimiseksi. Tämän tutkielman tarkoitus on tarkastella ohjelmistoprojektien työmääräarviomallien sopivuutta pelkille testausprojekteille, nostaa esiin joitakin merkittävimpiä testausprojektien työmäärään vaikuttavia tekijöitä sekä esitellä ratkaisuja, joilla testauksen työmäärään voidaan vaikuttaa.

Toisessa luvussa esitellään, mitä ohjelmistotestaus on, millaisia osa-alueita siihen kuuluu ja miten se yleensä suhteutuu koko ohjelmistoprojektiin. Kolmannessa luvussa käsitellään testausprojektien ulkoistamista ja syitä siihen. Neljännessä luvussa esitellään yleisimpiä perinteisiä työmääräarviomalleja ja arvioidaan niiden sopivuutta testausprojektien työmääräarviointiin. Viidennessä luvussa käydään läpi merkittävimpiä ohjelmistoprojektien työmääriin vaikuttavia tekijöitä niin toteutuksen kuin testauksenkin näkökulmasta. Kuudennessa luvussa esitellään testauksen automatisointia. Siinä pohditaan, milloin automatisointi on työmäärän kannalta järkevää ja milloin ei. Luvussa myös esitellään joitain testaustyökaluja, joilla työmäärään voidaan vaikuttaa. Seitsemäs luku on tutkielman yhteenveto.

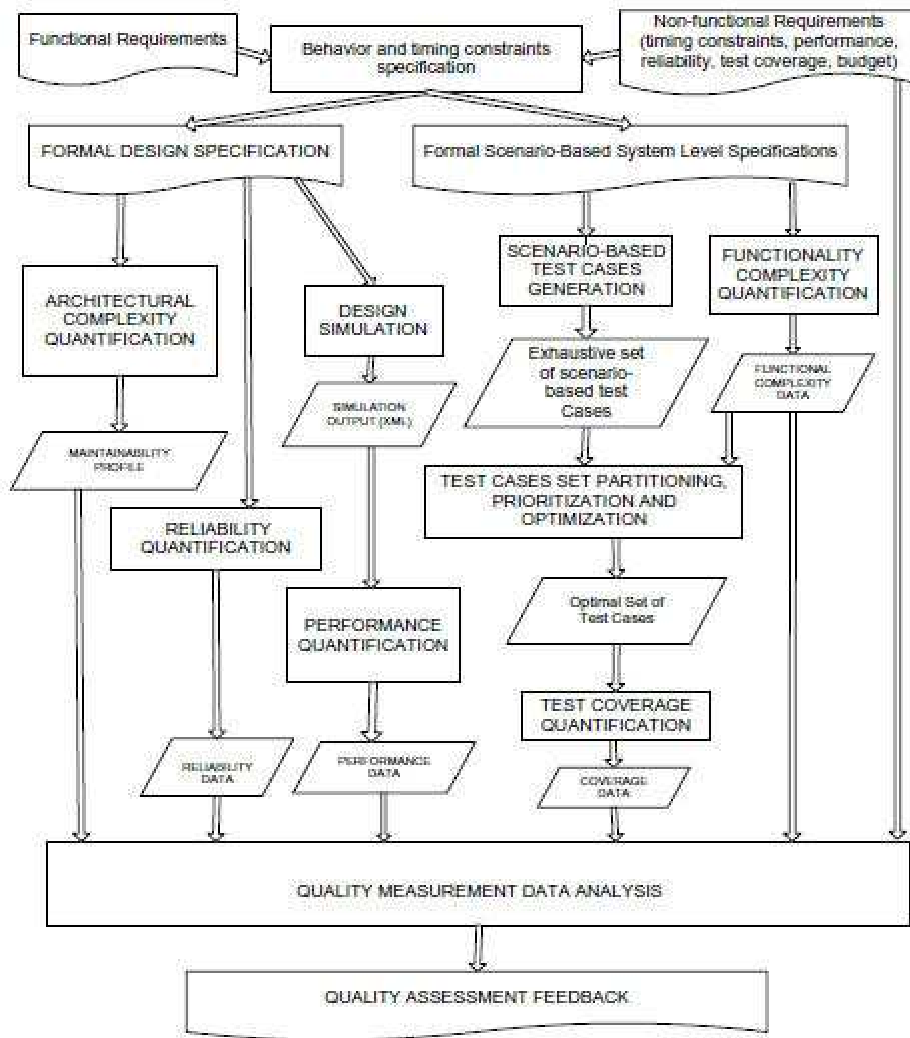
2. Testaus ohjelmistoprojekteissa

Testaaminen on tärkeä osa ohjelmistokehitystä. Sen tavoite on varmistaa ohjelmiston toimivuus halutulla tavalla ilman virheitä. Testaukseen voidaan katsoa kuuluvan viisi osa-aluetta: suunnittelu ja valvonta, määrittelyiden tarkastaminen, testitapausten luominen, testitapausten suorittaminen ja prosessin parantaminen. [Baresi and Bezze, 2006]

2.1. Suunnittelu ja valvonta

Testaukseen kuuluu laatuvaatimusten huomiointi koko projektin ajan. Tämä voidaan aloittaa heti, kun tiedetään, mitä projekti tulee sisältämään. Testaajien on tarkoitus löytää mahdolliset haavoittuvuudet ja riskialttiit ratkaisut jo suunnitteluvaiheessa, jolloin on mahdollista löytää vaihtoehtoisia ratkaisuja riskialttiiden tilalle.

Mitä kriittisempi järjestelmä on kyseessä, sitä tärkeämpää on laadunvalvonta. Kuva 1 havainnollistaa Ormandjievan ja muiden [2008] esittämää metodia, jolla näiden erityisen kriittisten hankkeiden laatua voidaan valvoa projektin alusta lähtien. Esitetty metodi ottaa laadullisesti kantaa toimintojen monimutkaisuuteen, suorituskykyyn, luotettavuuteen, arkkitehtuurin monimutkaisuuteen, ylläpidettävyyteen ja testauksen kattavuuteen matemaattisten mallien avulla. Nämä formaalit mittarit tuottavat toteuttajille lähes reaaliaikaista tietoa ohjelmiston laadusta. Kyseinen prosessi on kohtalaisen raskas, eikä vähemmän kriittisissä hankkeissa yleensä haluta panostaa laatuun aivan tässä laajuudessa.



Kuva 1. Laadunvalvontamallin työnkulku [Ormandjieva et al., 2008]

Testaussuunnitelma on yleisesti käytössä pienemmissäkin ohjelmistohankkeissa. Sen tarkoituksena on määritellä kyseisen hankkeen testausstrategia, ja siinä huomioidaan sekä hankkeen että organisaation erityispiirteet. Testaussuunnitelma toimii linjauksena projektin testauksesta. Testaussuunnitelmasta löytyy yleensä taulukossa 1 kuvatut asiat.

Testattavat asiat	Tarkoittaa esimerkiksi erilaisia asennustyyppejä, joita ohjelmistolla on.
Testattavat ominaisuudet	Määritellyt ominaisuudet, joiden toiminnallisuus tulee testata.
Ominaisuudet, joita ei testata	Ohjelmistossa voi olla asioita, joita ei kuitenkaan testata. Esimerkiksi jokin kolmannen osapuolen tuottama viestipalvelu, jonka oletetaan olevan testattu.
Lähestymistapa	Valittu laatustrategia, esimerkiksi mitä katselmoidaan ja missä eri vaiheissa testausta suoritetaan.
Hyväksymiskriteerit	Kriteerit, jotka läpäistyä ohjelmiston voidaan katsoa olevan julkaisukelpoinen. Esimerkiksi, ettei jäljellä ole yhtään vakavaa vikaa, mutta kosmeettisia virheitä voi olla.
Keskeytys ja uudelleenjatkaminen	Määritellään reunaehdot sille, että testausta voidaan suorittaa., ja ehdot sille, milloin testausta voidaan jälleen jatkaa, jos se on jouduttu keskeyttämään. Tällä voidaan tarkoittaa esimerkiksi virheiden määrää tai testausympäristön keskeneräisyyttä.
Riskit ja epävarmuustekijät	Tavanomainen riskianalyysi, jossa voidaan huomioida esimerkiksi henkilöstön menetys, tekniset ongelmat tai toteutuksen huono laatu.
Toimitettavat osat	Mitä toimitettavia osia projektista on tarkoitus syntyä.
Tehtävät ja aikataulu	Projektin testauksen aikataulutus, jota voidaan tarkentaa projektin edetessä. Kuitenkin yleensä testaus aikataulutetaan toteutus huomioiden. Jos toteutusvaihe venyy, pitää testausvaiheenkin aikataulua venyttää, eikä vain testata lyhyemmässä ajassa.
Henkilöstö ja vastuualueet	Testaukseen kuuluvat henkilöt ja heidän vastuualueensa projektissa. On myös hyvä huomioida henkilöiden työkuorma, eli ovatko he projektissa esimerkiksi 100% tai 50% työajasta.
Ympäristötarpeet	Testiympäristöä koskevat vaatimukset. Tällaisia voivat olla esimerkiksi tietyt matkapuhelinmallit, jos kyseessä on matkapuhelimen ohjelmiston testaaminen.

Taulukko 1. Keskeisimmät asiat, joihin testaussuunnitelmassa yleensä otetaan kantaa. [Baresi and Bezze, 2006]

Testaussuunnitelmaa tehdessä kohdataan usein ongelmia. Testaussuunnitelma pohjautuu määrittelyihin ja määrittelyt ovat hyvin usein virheellisiä, mistä johtuen niitä pitää tarkentaa ja korjata projektin edessä. Edellä mainitusta syistä testaussuunnitelmaa on hankala pitää täysin ajantasaisena, sillä määrittelyt voivat muuttua jatkuvasti. Kun määrittelyt vihdoin saadaan riittävän kypsiksi, on ohjelmointityö ehditty jo aloittaa. Näistä ongelmista huolimatta testaussuunnitelma kannattaa tehdä, sillä sitä tehdessä pitää miettiä ja suunnitella projektin testaukseen liittyviä asioita, jotka on hyvä tunnistaa ja kirjata ylös ennen varsinaisen testauksen aloittamista. [Mosley, 2000]

2.2. Määrittelyiden katselmoinnit ja kooditarkastukset

Katselmoinnit voidaan suorittaa heti, kun ominaisuus on määritelty. Tässä vaiheessa virheenkorjaus on vielä edullista, sillä toteutustyötä ei pitäisi vielä olla aloitettu. Katselmointien tarkoitus on löytää dokumentista puutteita ja virheitä. Yleensä katselmoitava dokumentti annetaan ajoissa luettavaksi tietylle joukolle ihmisiä. Ihmiset kokoontuvat yhdessä kommentoimaan dokumenttia ja yksi etukäteen sovittu henkilö toimii puheenjohtajana jakaen kommentointivuoroja. Havaitut virheet ja puutteet kirjataan ylös ja dokumentin tekijän tehtävä on korjata nämä dokumenttiin. Tarvittaessa voidaan järjestää useampiakin katselmointitilaisuuksia.

Kollanus [2006] on etsinyt syitä siihen, miksi katselmointeja käytetään melko vähän, vaikka niiden on todettu vaikuttavan huomattavasti ohjelmiston laatuun. Yritysten kokemusten perusteella suurimpia ongelmia katselmoinneissa ovat palaverin aiheuttama viivästys aikatauluun, keskustelun ajautuminen epäolennaisiin asioihin ja huono valmistautuminen katselmointitilaisuuteen. Aikataulun viivästymistä ei kuitenkaan koettu vakavana ongelmana. Huono valmistautuminen oli tutkimuksessa koettu suurimmaksi ongelmaksi ja siitä toki voi seurata se, että katselmoinnissa ei löydetä riittävästi virheitä suhteessa kulutettuun aikaan. Valmistautumiseen käytettävää aikaa tulisi painottaa ja resursseja siihen tulisi varata riittävästi. Keskustelun ajautumiselle sivuraiteille Kollanus [2006] esitti erilaisia syitä: osa koki ongelmanratkaisukeskustelun asiaankuuluvana, toisten mielestä se kulutti turhaan koko ryhmän resursseja. Kokonaisuutena suurin ongelma kiteytyy työntekijöiden asenteeseen ja motivaatioon. Toisten työn lukeminen ei ole välttämättä kiinnostavaa, omat työt koetaan tärkeämpinä kuin toisten töiden arviointi eikä resursoinnissa ole huomioitu katselmointiin käytettävää aikaa. Näihin puuttumalla katselmoinneista saatava hyöty paranisi ja sitä kautta todennäköisesti motivaatiokin.

Kooditarkastuksissa käydään läpi tuotettua ohjelmakoodia. Tarkoitus ei ole suorittaa koodia, vaan sitä luetaan kuten tekstiä ja yritetään löytää huonoja ratkaisuja tai virheitä. Kooditarkastustilaisuudet ovat muodollisesti samankaltaisia kuin edellä kuvattu dokumenttien katselmointitilaisuus. Kooditarkastuksen virhelöydökset verrattuna ohjelmistotestaukseen on esitetty taulukossa 2. [Kumari et al., 2009]

Nro	Virhetyyppi	Kooditarkastus	Ohjelmistotestaus
1	Yksikkötason rajapintavirheet	X	-
2	Erittäin monimutkaisesti tehty koodi	X	-
3	Ominaisuudet, joita ei ole vaatimuksissa	X	-
4	Käytettävyysongelmat	-	X
5	Suorituskykyongelmat	X	X
6	Huonorakenteinen koodi	X	-
7	Puute vaadituissa ominaisuuksissa	X	X
8	Raja-arvovirheet	X	X

Taulukko 2. Erityyppisten virheiden löytäminen kooditarkastuksissa verrattuna testaukseen [Kumari et al., 2009]

Vaatimusmäärittelyjen ja suunnittelujen katselmointien sekä kooditarkastusten on havaittu laskevan ohjelmistotuotantokustannuksia huolimatta niihin kuluvasta ajasta. Itsestäänselvää on, että edellä mainitut toimenpiteet nostavat ohjelmiston laatua. Taulukossa 3 esitetään keskimääräisiä työmääriä virheiden löytämiseen erilaisilla tekniikoilla. [Kumari et al., 2009]

Nro	Tekniikka virheen löytämiseen	Minimiarvo	Todennäköisin arvo	Maksimiarvo
1	Suunnitteludokumentin katselmointi	0.58	1.58	2.9
2	Kooditarkastus	0.67	1.46	2.7
3	Ohjelmistotestaus	4.5	6.0	17

Taulukko 3. Keskimääräinen työmäärä virheen löytämiseen virhettä kohden tunteina ilmoitettuna [Kumari et al., 2009]

Toisaalta, kun huomioidaan Kollanuksen [2006] esille tuomat ongelmat katselmointikäytännöissä, voidaan olettaa, että Kumarin ja muiden [2009] hyvin katselmointimyönteisiin lukemiin ei välttämättä jokaisessa ohjelmistoprojektissa päästä. Esitettyjen lukemien pitäisi toki kannustaa ja motivoida käyttämään katselmointeja ohjelmistoprojekteissa enemmänkin.

Jones [2007] esittää mielenkiintoisia lukuja suunnitteludokumenttien katselmoinnista ja kooditarkastusten hyödystä. Hänen mukaansa suunnitteludokumenttien katselmoinnilla ja kooditarkastuksilla voidaan poistaa 60 prosenttia olemassa olevista virheistä. Luku on huomattavasti paljon suurempi kuin millään muulla testausmuodolla. Jos projektissa on käytössä sekä dokumenttien että toteutuksen katselmointeja, on virheitä 80 % vähemmän siinä vaiheessa, kun varsinainen ohjelmiston testaus alkaa.

2.3. Testitapausten luominen

Testitapaukset tulee suunnitella siten, että niillä voidaan olettaa löydettävän mahdolliset virheet ohjelmistosta. Suunniteltaessa tulee pitää mielessä, että jokainen testitapaus on voitava yhdistää jollakin tavalla ohjelmiston vaatimuksiin. Testitapausten suunnittelu tulee aloittaa heti, kun vaatimusmäärittely on valmis. Ohjelmistoa ei tarvitse olla toteutettuna testitapauksia varten; testitapauksia voidaan kyllä päivittää, kun vaatimukset ja määrittelyt päivittyvät. Suunnitteluvaiheessa tulisi löytää ohjelmiston haavoittuvimmat ja kriittisimmät kohdat. Pareto-periaatteen mukaan 80 % virheistä tulee noin 20 %:sta ohjelmakoodia. Eli useimmiten ohjelmistolla on erityisen haavoittuvia kohtia ja näiden paikallistaminen on tärkeä osa testauksen suunnittelua. Testitapausten sisältöön vaikuttaa kuitenkin merkittävästi se, mitä niillä on tarkoitus testata ja kuinka laajaa kokonaisuutta. [Pressman, 1997]

Testitapauksissa kerrotaan alkuehto, joka voi liittyä esimerkiksi joidenkin muiden testitapausten läpäisyyn tai systeemin konfigurointiin. Testistä kerrotaan, miten se on tarkoitus suorittaa ja minkälaisilla arvoilla. Erilaisia arvokombinaatioita on yleensä useita yhtä testitapausta kohden. Testitapauksessa on myös kerrottava, mikä on sen odotettu tulos ja hyväksymiskriteeri. Sen lisäksi, että testitapaukset peilaavat ohjelmiston määrittelyä, kannattaa testaajan testitapausta suunnitellessaan käyttää luovuutta, tutustua koodin rakenteeseen ja lisätä testitapauksia, joilla olettaa ohjelman toiminnan rikkoontuvan. [Myers, 2004]

Testitapauksilla olisi tarkoitus kattaa toteutettava ohjelmisto mahdollisimman laajasti. Toisaalta resurssit useimmiten pakottavat jonkinlaiseen priorisointiin ja silloin testaajan tulisi osata kohdentaa testitapauksia haavoittuvimmille ja kriittisimmille alueille. Yleisesti kriittisinä alueina voidaan pitää systeemin rajapintoja sekä syötteiden ja ulostulojen raja-arvoja. [Mosley, 2000]

Tyypillisiä testitapausten suunnittelumenetelmiä ovat white box -testaus ja black box -testaus. White box -testauksen kohdalla keskitytään lähinnä kooditasolle tutkimalla esimerkiksi metodien, olioiden, rajapintojen yms. reagointia erilaisiin syötteisiin, sekä sallittuihin että virheellisiin. Nimitys tulee siitä, että tällä tekniikalla "nähdään ohjelmakoodin sisälle". Black box -testaus perustuu useimmiten ohjelman toiminnalliseen määrittelyyn. Nimityksellä tarkoitetaan sitä, että ohjelma toimii ikää kuin mustana laatikkona, jolle voidaan antaa tiettyjä syötteitä ja se vastaa niihin toiminnoilla. Se, mitä laatikon sisällä tapahtuu, ei kiinnosta tämän tason testaamisessa. Tällä tasolla ei enää "nähdä koodia", vaan ohjelmaa testataan vain toiminnallisesta näkökulmasta. Tällöin testitapauksetkin määritellään korkeammalla tasolla ja niillä yleensä suoritetaan ohjelman toimintoja. [Pressman, 1997]

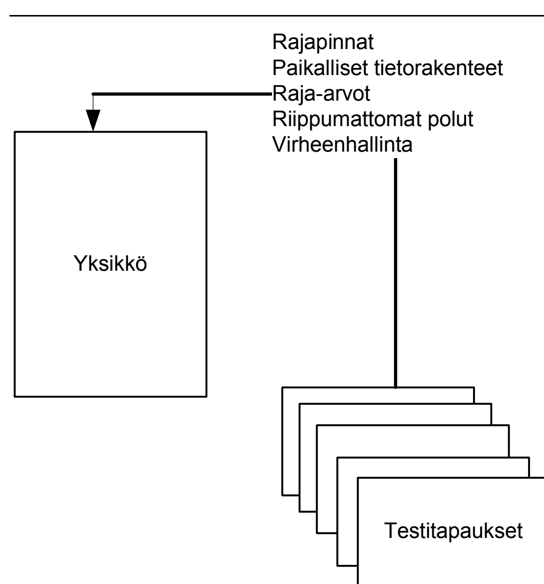
Testitapaukset tulee dokumentoida vähintään neljästä syystä. Testaajia on usein enemmän kuin yksi ja tiedon tulee kulkea testaajien välillä. Ei ole resurssien käytön kannalta järkevää, että moni testaa samoja asioita, joten kirjattujen testitapausten avulla

voidaan vastuualueita jakaa selkeästi. Toisaalta henkilöstö saattaa vaihtua projektin aikana, joten uusi henkilö saa testitapausten kautta tietoa projektin luonteesta. Usein samoja testitapauksia joudutaan toistamaan useita kertoja projektin aikana, joten on hyödyllistä, että tiedetään tarkat arvot, joilla jotakin toiminallisuutta on aiemmin testattu ja joilla se ei mahdollisesti ole toiminut. Toisaalta kirjatut testitapaukset helpottavat mahdollisen virheraportin kirjoittamista, sillä niistä saadaan suoraan tieto siitä, millä arvoilla ohjelma on toiminut ja millä ei. Testitapauksia voidaan käyttää myös raportointimielessä osoittamaan, mitä on testattu ja mitä ei. Kriittisissä sovelluksissa saattaa lisäksi tulla eteen tilanne, jossa on erittäin tärkeää voida todistaa jotakin testatuksi. [Patton, 2005]

2.4. Testitapausten suorittaminen

Jos testitapaukset on hyvin dokumentoitu, lähes kenen tahansa pitäisi kyetä suorittamaan testit niiden avulla. Testausta tulisi suorittaa monessa eri vaiheessa projektia ja testaustavan tulisi riippua siitä, mikä vaihe on meneillään. Projektista riippuen suoritetaan erilaisia testaustyyppejä. Esimerkiksi, jos ohjelmistolla ei ole suorituskykyvaatimuksia, sille ei välttämättä tarvitse erillistä suorituskykytestausta suorittaa.

Yksikkötestaus on useimmiten ohjelmoijan suorittama ensimmäinen testivaihe, ja se lasketaan kuuluvaksi toteutukseen. Ohjelmoijan tehtävä on testata oma koodinsa ja varmistaa sen toimivan kuten hän on ajatellutkin. Yksikkötestaus on usein hyvin koodiläheistä ja siinä testataan esimerkiksi yksittäisten metodien, olioiden tai rajapintojen virheetön toiminta. Yksikkötestausta voidaan suorittaa manuaalisesti tai automaattisesti. [Mosley, 2000]



Kuva 2. Yksikkötestaukseen kuuluvat osiot [Pressman, 1997]

Integraatiotestauksessa testataan eri moduulien yhdistäminen toisiinsa. Kun yksikkötestauksessa on testattu kunkin moduulin ottamat ja antamat syötteet, integraatiotes-

tauksessa testataan syötteiden liikkuvuus moduulien välillä oikein. On suositeltavaa suorittaa integraatiotestaus inkrementaalisesti, eli lisätä yksi moduuli kerrallaan muiden joukkoon. Tällöin mahdollisessa virhetilanteessa on helpompi löytää virheen aiheuttaja. [Baresi and Bezze, 2006]

Toiminnallisen testauksen on nimensä mukaisesti tarkoitus testata ohjelman toimintoja. Toiminnallinen testaus suoritetaan yleensä toiminnallista määrittelyä vasten, josta seuraa se, että vika ei aina välttämässä ole ohjelmassa, vaan se voi löytyä myös määrittelystä. Mitä huonommin määrittely on tarkastettu, esimerkiksi katselmoimalla, sitä todennäköisempää on, että siitäkin löytyy vikoja. [Baresi and Bezze, 2006]

Systeemitestauksessa testataan kokonaisuuden näkökulmasta, eli käytetään ohjelmistoa sen oikeassa ympäristössä ja varmistutaan siitä, että käyttötapaukset toimivat kuten on määritelty. Jos vaatimuksiin on näin määritelty, systeemitestaukseen kuuluu myös virhetilanteista toipumisen testaaminen, turvallisuuden testaaminen, stressitestaus ja suorituskykytestaus. [Pressman, 1997]

Käytettävyystestauksessa on tarkoitus testata ohjelman käytettävyyttä loppukäyttäjän näkökulmasta. Käytettävyys voidaan määritellä kolmella termillä. Tuloksellisuudella tarkoitetaan sitä, että käyttäjä onnistuu suorittamaan halutun toiminnon. Tehokkuus tarkoittaa, että käyttäjä pystyy suorittamaan halutun toiminnon ns. minimityöllä. Tyytyväisyydellä tarkoitetaan sitä, että käyttäjälle tulisi jäädä luottavainen olo ohjelman toiminnasta. [Kortum, 2008]

Käytettävyystestaus on ollut melko aliarvostettu testausmuoto viime vuosiin asti. Useimmissa ohjelmistotaloissa tunnustetaan käytettävyyden tärkeys, mutta silti sen parantamiseksi ei allokoida resursseja eikä varsinaisia käytettävyystestauksia useinkaan suoriteta. Asenne käytettävyyttä kohti on kuitenkin muuttumassa ja ohjelmistotalot suhtautuvat toiveikkaasti perinteisten ohjelmistotuotannon metodien ja käytettävyyden yhdistämiseen. [Bygstad et al., 2008]

Regressiotestauksella tarkoitetaan testien uudelleenajamista. Vaikka testit on jo kertaalleen läpäisty, niin sovelluksen muututtua ainakin osa testeistä pitää ajaa uudelleen. Muuttuminen voi tarkoittaa mm. virheenkorjausta. Muuttumisen jälkeen tulee varmistaa, ettei mikään muu kohta ole mennyt rikki tai ettei toiminnallisuus ole muuttunut virheelliseksi. Regressiotesteihin tulee valita osajoukko kaikista testeistä ja osajoukon valinnassa tulee huomioida ohjelmiston kriittisimmät kohdat. [Pressman, 1997]

Regressiotestauksen laajuus on työmäärällisesti merkittävä tekijä. Jos resurssit olisivat rajattomat, olisi tietenkin kannattavinta ajaa kaikki testit uudelleen. Näin ei kuitenkaan yleensä ole, ja on kehitetty erilaisia tekniikoita regressiotestisetin valitsemiseksi. Regressiotestitapausten valintaan on olemassa kolme yleistä tekniikkaa. Testisetin minimointi, joka perustuu testauksen kattavuuteen, eli pyritään mahdollisimman pienellä testitapausmäärällä mahdollisimman suureen testikattavuuteen. Testitapausten valinta niiden tapauskohtaisen tärkeyden perusteella, eli huomioidaan ohjelmistoon tehdyt

muutokset ja valitaan regressiotestiin ne tapaukset, joihin tehdyt muutokset todennäköisimmin vaikuttavat. Testitapausten priorisointitekniikka laittaa testitapaukset oletettuun tärkeysjärjestykseen, jolloin testitapauksia suoritetaan järjestyksessä tärkeimmistä vähemmän tärkeisiin. [Yoo and Harman, 2010]

Hyväksymistestaus on yleensä ohjelmiston tilaajan suorittama viimeinen testi ennen kuin ohjelmisto luovutetaan. Hyväksymistestauksen tarkoitus on varmistaa ohjelmiston sopivuus siihen tarkoitukseen, jota varten se on tilattu. Hyväksymistestaus voidaan jakaa kahteen osaan: alfa-testaus suoritetaan yleensä suppeammalla käyttäjäjoukolla toteuttajan testiympäristössä; beeta-testaukseen osallistuu suurempi joukko käyttäjiä ja siinä ohjelmistoa käytetään sen lopullisessa käyttöympäristössä. [Kruchten, 2003]

Testitapausten suorittaminen yleensä dokumentoidaan. Oman kokemukseni mukaan nykyään on usein käytössä jokin ohjelmisto testitapausten säilyttämiselle ja samaa ohjelmistoa voidaan käyttää myös testauksen raportointiin. Eräs tällainen ohjelma on HP:n Quality Center. Siihen voidaan syöttää testitapaukset suoritusohjeineen ja näistä voidaan kopioida aina suoritettava testiotos. Tällaiseen testiotokseen voidaan sitten merkitä, läpäistiinkö testitapaus vai ei. Ajetuista testikierroksista on mahdollista saada yhteenvetoja ja tällaisten avulla projektin vastuuhenkilöiden on helppo seurata projektin etenemistä ja virheiden määrää.

On myös kehitetty malleja, joilla testaustilanteen perusteella voidaan ennustaa ohjelmistoprojektin virheenkorjaamiseen kuluvaa aikaa. Tämä edellyttää, että testauksen tilanne on reaaliaikaisesti raportoituna siten, että tietoa voidaan käsitellä ja sen perusteella laskea arviointeja. Tämä ennustaminen vaatii jonkin verran manuaalista ylläpitotyötä, mutta sen on todettu toimivan kohtuullisen tarkkana ennusteena projektista. [Staron and Meding, 2008]

2.5. Prosessin parantaminen

Prosessin parantaminen liittyy lähinnä siihen, että vastaavankaltaisia projekteja on tulevaisuudessakin. Jos projektissa havaitaan jotain erityisiä ongelmia, esimerkiksi kehitysympäristön tai joidenkin henkilöiden työnlaadun suhteen, nämä asiat voidaan huomioida sitten seuraavassa projektissa asianmukaisella tavalla. Ohjelmiston laadusta vastaavien tulee seurata projektin mahdollisia toistuvia ongelmia ja yrittää minimoida näiden vaikutusta. Korjausliikkeitä ongelmakohtiin voidaan tehdä jo meneillään olevaan projektiinkin, mutta useimmin ne kuitenkin keskittyvät vasta tulevaisuuteen. [Baresi and Bezze, 2006]

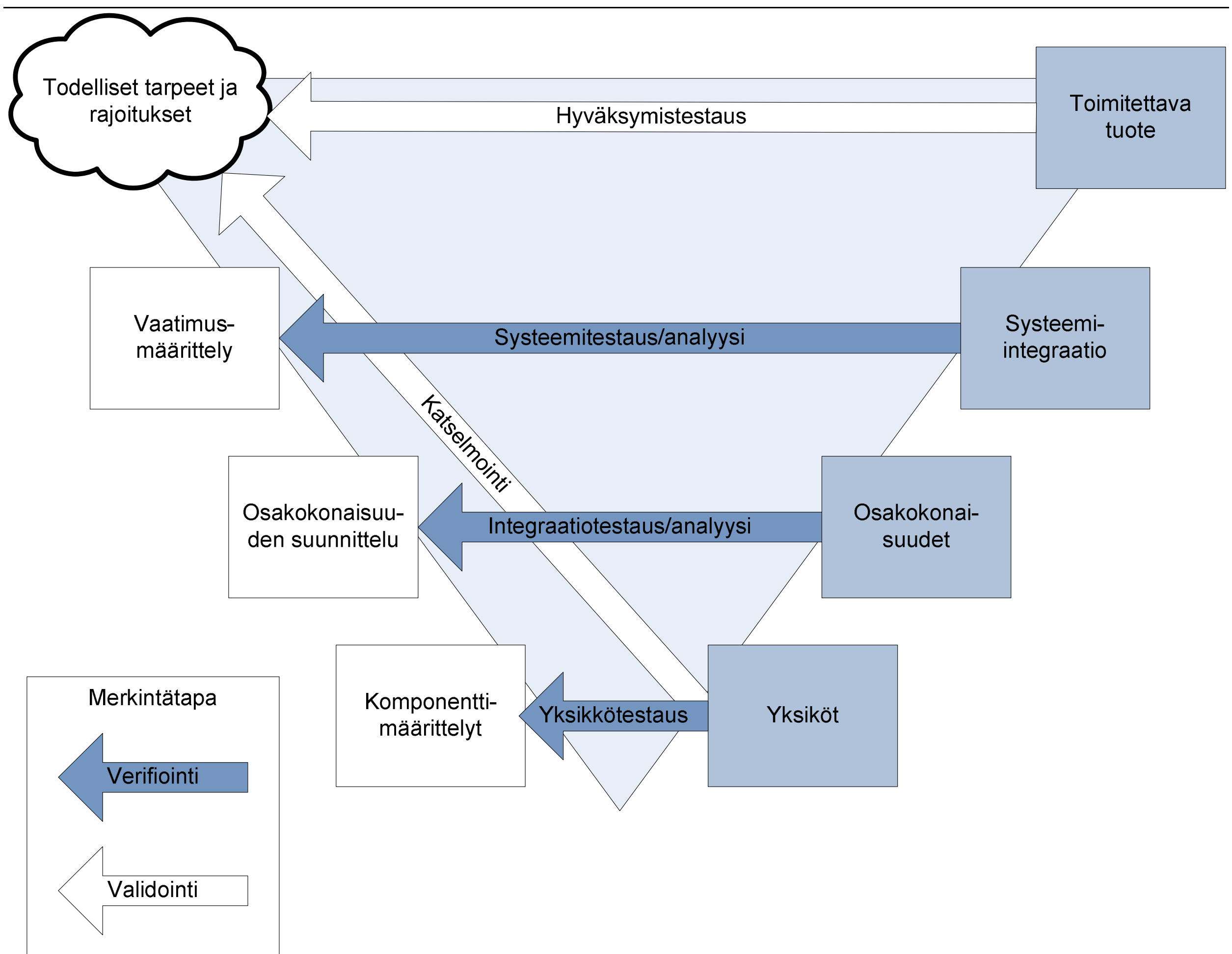
Oman kokemukseni perusteella laadukas testauksenhallintaohjelmisto, kuten esimerkiksi aiemmin mainittu Quality Center, on tärkeä työväline prosessin parantamisessa. Testauksenhallintaohjelmistot mahdollistavat laatuseurannan jälkikäteen. Voidaan esimerkiksi seurata, miten usein virheenkorjaus aiheutti uuden virheen tai miten paljon testisettien ajaminen nopeutui projektin loppua kohden, kun ne tulivat testaajille tutuik-

si. Kun ongelmakohdat pystytään huomioimaan, voidaan selvittää syy ja siten korjata ongelmakohta seuraavassa projektissa.

Yleisiä syitä prosessin ja projektien mittaamiseen ovat mm. asiakastyytyväisyys, aikataulussa pysyminen, työkuorma, systeemin koko, budjetti ja julkaisun jälkeiset viat. Mittareiden avulla ylempi johto voi muodostaa käsityksensä projektin onnistumisesta ja prosessin soveltuvuudesta. [Soini et al. 2006]

2.6. Yhteenveto ohjelmistoprojektin testauksesta

Testaus voidaan jakaa validointiin ja verifiointiin. Validoinnilla tarkoitetaan sitä, että ollaan tekemässä oikeita asioita. Validoinnilla varmistetaan se, että ohjelmisto sisältää ne asiat, joita sen on tilattu sisältävän. Verifioinnin tarkoitus on varmistaa, että ohjelmisto toimii oikein. Kuvassa 3 on esitetty validoinnin ja verifioinnin jakautuminen testausprosessissa. [Baresi and Bezze, 2006]



Kuva 3. Testauksen jakautuminen validointiin ja verifiointiin V-mallisessa ohjelmistokehitysprojektissa [Baresi and Bezze, 2006]

Ohjelmistoprojektin testaustavoista pitää päättää aina ohjelmistokohtaisesti. Ohjelmiston vaatimukset ja toteutustapa asettavat suunnan testaukselle, sekä suoritettavien testaustasojen että kriittisyyden suhteen. Tämän lisäksi jokaisella organisaatiolla on

omat toimintatapansa, joiden avulla organisaatio seuraa laatustrategiaansa. Testausprosessin tehtävä on tukea tätä laatustrategiaa. [Kyllönen, 2008]

3. Testausprojektien ulkoistaminen

Sen lisäksi, että testaus saatetaan ulkoistaa ns. uskottavuussyistä, voivat testausprosessin eriyttämiseen olla syynä myös testauksen ammattimainen profiloituminen, hankkeiden laajuus tai halvemman työvoiman käyttö. [Kakkonen, 2009] Jos ohjelmistolla on useita toimittajia, kokonaisuuden testaamisen suorittaa jonkin tahon näkökulmasta ulkopuolinen testaustiimi.

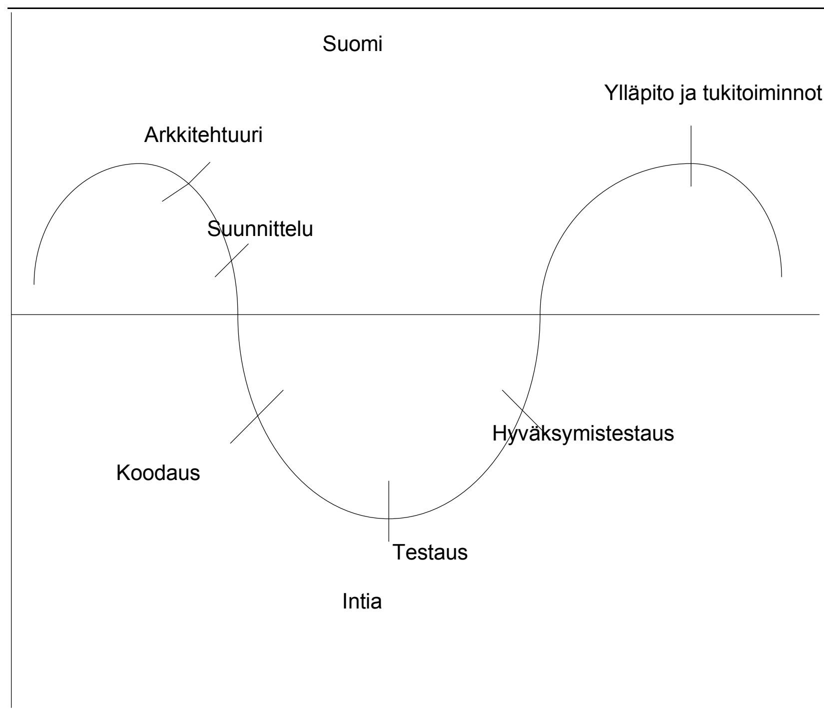
3.1. Syitä testausprojektien ulkoistamiseen

Ammattimaisella profiloitumisella tarkoitetaan sitä, että testauksen arvostus asiantuntijatehtävänä on kasvamassa. Testausta voi enenevässä määrin opiskella ja siitä voi suorittaa sertifiointeja (esimerkiksi ISTQB). Suomen yliopistoissa annettava testauksen opetus on alkanut 1990-luvulla, ja se on tällä hetkellä vakiintunut hyvälle tasolle. [Kyllönen, 2008] Testaus kuitenkin kehittyy koko ajan suurin harppauksin, joten eroja yliopistojen testausopetuksessa varmasti löytyy. Kymmenen vuotta sitten testausta ei ollut mahdollista Tampereen yliopistossa kovin paljoa opiskella. Onneksi tilanne on nyt muuttunut, ja jo opiskeluaikana voi halutessaan perehtyä testaamiseen ja sen ongelmakysymyksiin.

Ulkoistamispaine perustuu osittain jo mainittuun itse testaamisen asiantuntijuuteen, mutta toisaalta myös siihen, että nykyisin laajoissa hankkeissa on käytössä monenlaisia teknisiä ratkaisuja ja rajapintoja. Tällöin paras vaihtoehto lopputuloksen kannalta on se, että testaajalla on jo ennestään kokemusta ja tietämystä kyseisistä teknisistä ominaisuuksista. Tähän tarpeeseen ovat vastanneet Suomessakin toimivat eri alojen testausasiantuntijoita välittävät yritykset. [Kakkonen, 2009]

Toisaalta taas manuaalista ja yksinkertaista toistuvaa työtä ostetaan ns. halvemmista maista. [Ali-Yrkkö and Jain, 2005] Tällöin jo fyysinen etäisyys voi aiheuttaa testauksen erillisyyden muusta tuotekehityksestä. Toisaalta Ali-Yrkkö ja Jain toteavat keskustelunavauksessaan, että projektinhallinnasta ja ylimääräisestä kommunikaatiosta aiheutuvat kulut tasoittavat esimerkiksi Suomen ja Intian välistä työvoiman palkkaeroa huomattavasti.

Kustannusten lisäksi työtä ostetaan alihankintana myös resursoinnin takia. Omien työntekijöiden tietotaito halutaan hyödyntää ohjelmiston seuraavaan sukupolven suunnitteluun. Massamuotoisesti suoritettava kehitys- ja testaustyö ostetaan alihankintana. Kuvassa 4 on kuvattu, miten ohjelmistoprosessin eri työvaiheiden työ useimmiten jakautuu, kun työvoimaa ostetaan halvemman palkkatason maasta. Varsinainen ideointi ja alustava määrittely tuotetaan yrityksessä itse, mutta tekniseen suunnitteluun ja siitä eteenpäin alihankkijat osallistuvat prosessiin. [Ali-Yrkkö and Jain, 2005]



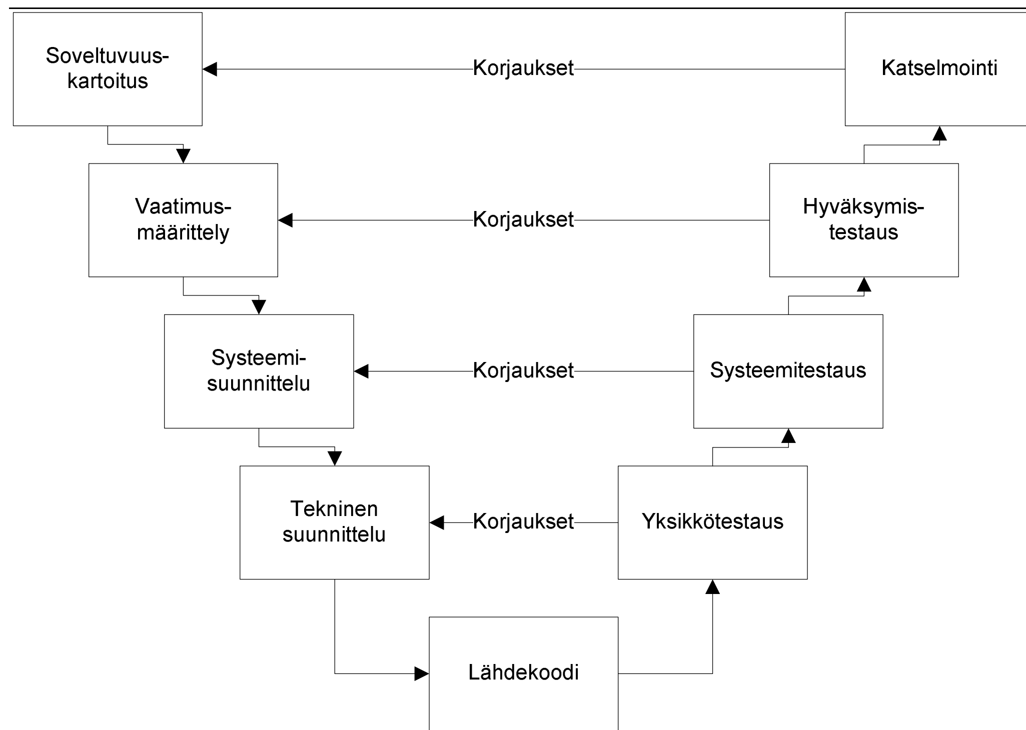
Kuva 4. Yksinkertaistettu kaavio kuvaa työmäärän ja työtehtävien jakautumista Suomen ja Intian organisaatioiden välillä alihankintaisissa ohjelmistoprojekteissa. [Ali-Yrkkö and Jain, 2005]

Tässä on toki ongelmana se, että ulkoiset resurssitkin pitää perehdyttää aiheeseen ja se vie resursseja omilta työntekijöiltä. Pitkässä alihankintasuhteessa tietotaitoa kertyy myös alihankkijalle ja näitä resursseja saadaan siten hyödynnettyä täysimääräisemmin. Ei kuitenkaan ole ennenkuulumatonta, että yritys haluaa palkata paljon tietoa ja kokemusta keränneitä alihankkihan henkilöitä omaksi työvoimakseen.

Erittäin laajat hankkeet ovat arkipäivää tietoyhteiskunnan kehittyessä. Eräänä esimerkkinä tästä toimii kansallinen terveysarkisto. Näihin laajoihin yhteiskunnallisiin hankkeisiin osallistuu useita eri tekijöitä ja kokonaisuus testataan laajemmin, kun useamman eri toimittajan tuottamat palaset yhdistetään.

3.2. Erilaisten testausprojektien ulkoistamismahdollisuudet

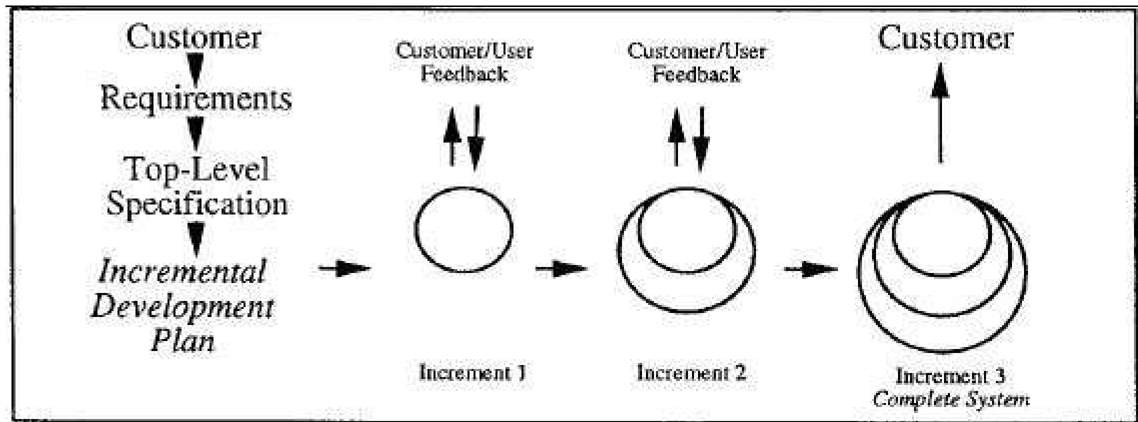
Prosessimallit antavat erilaisia mahdollisuuksia eriyttää testausprosessia. Esimerkiksi kuvassa 5 esitetty V-malli mahdollistaa testaustasojen suorittamisen alhaalta ylöspäin. Yksikkötestaus kuuluu yleensä ohjelmoijalle, hyväksymistestauksesta päättää tilaaja ja katselmoinnit usein pidetään organisaation sisäisinä tilaisuuksina. Tällöin mahdollisesti ulkoistettavaksi testausvaiheeksi jää toiminnallinen testaus.



Kuva 5. V-malli ohjelmistotuotannossa [Hughes and Cotterell, 2006]

Inkrementaalisessa prosessissa toistetaan käytännössä V-mallia useana pienenä kierroksena. Testauksen kannalta tästä aiheutuu se, että mitä lähemmäksi lopputuotetta päästään, sitä suurempi on testauksen työmäärä. Ensimmäisellä kierroksella voi testattavia ominaisuuksia olla viisi ja kymmenennellä kierroksella 150. Tämä aiheuttaa suuren resurssitarpeen kasvun ohjelmiston lähestyessä valmista. Jokaisella kierroksella tulisi varmistaa vanhojen toiminnallisuuksien säilyminen eheänä ja uusien ominaisuuksien toiminta määrittelyn kuvaamalla tavalla. [Kyllönen, 2008]

Oman kokemukseni perusteella inkrementaalisessa prosessissa usein ulkoistetaan edellisillä kierroksilla testatut ominaisuudet. Tällöin on usein tiedossa karkea arvio testattavien ominaisuuksien työmäärästä, eikä ole kovin todennäköistä, että regressiotestattavista ominaisuuksista löytyy virheitä. Laadukkaassa prosessissa regressiotestaus kuitenkin suoritetaan jokaisella kierroksella. Inkrementaalisesti tuotetun ohjelmiston koon kasvua havainnollistetaan kuvassa 6.



Kuva 6. Ohjelmiston kasvaminen inkrementaalisessa ohjelmistoprosessissa. [Trammell et al., 1996]

Tämä myös tukee Ali-Yrkön ja Jainin [2005] saamia tuloksia. Inkrementaalisessa prosessissa ohjelmiston uudet ja tietämyksen kannalta kriittiset elementit saadaan pidettyä oman yrityksen sisällä, mutta jo mahdollisesti julkaistujen toimintojen regressiotestaus voidaan ulkoistaa. Tällöin yrityksen sisällä testauksen työmäärä voidaan säilyttää vakiona ja ulkoistamalla saada lisäresursseja regressiotestaukseen.

4. Työmääräarviointi ohjelmisto- ja testausprojekteissa

Työmääräarviomenetelmiä on tutkittu paljon, mutta silti työmääräarvion tekeminen on edelleen erittäin haastavaa. Työmäärän arvioimiseksi kehitettyjä malleja on olemassa paljon, mutta silti suuri osa projekteista edelleen epäonnistuu. [Georgiadou, 2004] Jokaiseen projektiin vaikuttaa aina oma koostumuksensa näistä yleisesti vaikuttavista tekijöistä, vaihtelevin painotuksin. Työmääräarvioinnin tekijän osaamista mittaa se, miten hyvin hän osaa ottaa huomioon kunkin projektin erityispiirteet.

Testausprojektien työmääräarvioinnista erityisen haastavaa tekee se, että virheiden määrä ja laajuus selviää vasta testausvaiheessa. Jones [2007] tutki kollegojensa kanssa noin 12000 ohjelmistoprojektin virheiden esiintyvyyttä vuosina 1984–2007. Tämän pohjalta hän esittää taulukon 4 virheiden esiintyvyydestä ohjelmistotyypistä riippuen. Luvut on esitetty vikoina toimintopistettä kohden. Toimintopisteen käsite esitellään alakohdassa 4.1.2.

Lähde	Web	MIS	Ulkoinen	Kaupallinen	Systeemi	Armeija	KESKIARVO
Vaatimukset	1.00	1.00	1.10	1.25	1.30	1.70	1.23
Suunnittelu	1.00	1.25	1.20	1.30	1.50	1.75	1.33
Lähdekoodi	1.25	1.75	1.70	1.75	1.80	1.75	1.67
Dokumentointi	0.30	0.60	0.50	0.70	0.70	1.20	0.67
Huono korjaus	0.45	0.40	0.30	0.50	0.70	0.60	0.49
YHTEENSÄ	4.00	5.00	4.80	5.50	6.00	7.00	5.38

Taulukko 4. Ohjelmistotyypeittäin eri lähteistä peräisin olevien vikojen esiintyvyys toimintopistettä kohden [Jones, 2007]

Taulukossa 1 lyhenne MIS (Management information system) tarkoittaa erilaisia tiedonhallintasovelluksia. Voisi olettaa, että tiedonkeruun alkaminen jo 1980-luvulla vaikuttaa erityisesti web-sovellusten virhelukemiin. Vasta viimeisen kymmenen vuoden aikana on yleistynyt laajempien ohjelmistojen kehittäminen selainpohjaisiksi. Tästä syystä kyseiseen aineistoon on todennäköisesti huomioitu melko yksinkertaisia web-sovelluksia ja tämän vuoksi niissä näyttäisi esiintyviä vikoja vähemmän. Taulukon tuomalla datalla voidaan tietenkin karkeasti arvioida todennäköisesti löytyvää virheiden määrää ja siten arvioida, miten se vaikuttaa testauksen työmäärään.

4.1. Ohjelmiston koko

Ohjelmiston koko on varmasti selkein ja helposti ymmärrettävin työmäärään vaikuttava tekijä. On olemassa useita erilaisia tapoja muuntaa ohjelmiston koko matemaattiseksi arvoksi. Tämän ansiosta ohjelmiston kokoa voidaan käyttää työmääräarvioinnin apuna monessa eri vaiheessa projektia. Mitä suurempi ohjelmisto tulee olemaan, sitä enemmän

siinä on tehtävää ja siten kuluu enemmän aikaa sen tekemiseen. Kun ohjelmistotuotannon menetelmät koko ajan kehittyvät, myös ohjelmistokomponenttien uudelleenkäyttöön on kiinnitetty enemmän huomiota. Nykyaikaiset suunnittelumenetelmät ja modernit kehitysalustat mahdollistavat komponenttien uudelleenkäytön entistä helpommin ja tehokkaammin, joten pelkän koon perusteella tehtävä työmääräarviointi ei välttämättä enää anna kovin tarkkaa kuvaa työmäärästä.

4.1.1. Ohjelmiston koon vaikutus työmääräarvioon

Jotta kehittyneempiä ohjelmistoarkkitehtuureja voidaan suunnitteluvaiheessa käyttää, pitää suunnittelua tekevällä henkilöllä luonnollisesti olla riittävä osaaminen ja ymmärrys niiden käyttämiseen. Jos projektin henkilöstöltä löytyy tätä osaamista, on toteutuksen realistinen työmääräarvio pienempi kuin henkilöstöllä, jolta osaamista ei löydy. Täten voidaan päätellä, että henkilöstön osaamisen kautta yritys voi hyötyä tarjouskilpailuissa, joissa työmääräarvio, ja siten suoraan hinta, vaikuttaa.

Mahdollisuuksien mukaan erittäin laajat projektit kannattaa pilkkoa osiin, jolloin osaprojektit sisältävät vähemmän toteutettavaa ja niiden työmääräarviointi muun projektinhallinnan ohessa on helpompaa. Tätä toteutusmallia tukee mm. inkrementaalinen ohjelmistokehitys. Myös koko ajan yleistyvät ketterät ohjelmistotuotantomenetelmät painottavat projektien pilkkomista pieniin osakokonaisuuksiin, jolloin osakokonaisuudet ovat helpommin hallittavia.

Ketteriä ohjelmistokehitysmenetelmiä ovat esimerkiksi Extreme Programming ja Scrum. Yleisesti ottaen voidaan sanoa, että ketterät menetelmät eroavat perinteisistä ohjelmistokehitysmalleista siinä, että ohjelmistot tuotetaan huomattavasti perinteistä pienemmissä osissa koko ajan asiakkaan kanssa kommunikoiden. [Huttunen, 2006]

4.1.2. Ohjelmiston koon käyttäminen työmääräarviomalleissa

Koska ohjelmiston kokoa on käytetty apuna työmääräarvioissa jo kymmenien vuosien ajan [Georgiadou, 2004], on olemassa monia erilaisia tapoja muuntaa ohjelmiston koko matemaattiseksi arvoksi. Eräs vanhimmista on arvioida käyttää suoraan lähdekoodirivien lukumäärää. Tosin yleensä siinä vaiheessa projektia, kun työmääräarvioita tehdään, ei ole kovin realistista arvioida ohjelmistoon tulevan lähdekoodin rivimäärää. Tästä johtuen ohjelmiston koon arviointiin on kehitetty muitakin laskentamalleja.

On vaikea määritellä, mikä lasketaan lähdekoodiriviksi ja mikä ei. Tästä syystä pelkkien rivien laskemisesta kehittyneempi malli on laskea lähdekoodin lausemäärä. Lausemäärä on kuitenkin myös melko hankalasti arvioitava luku työmäärän arviointivaiheessa.

Kehittynein versio ohjelmiston koon arviointiin on laskea ohjelmiston toimintopisteitä. Toimintopisteellä tarkoitetaan yhtä ohjelmistolta edellytettävää toiminnallista ominaisuutta. Toimintopisteet määritellään loppukäyttäjän näkökulmasta, joten ohjel-

mistoarkkitehtuuri tai ympäristö ei vaikuta tämän arviointitavan käyttämiseen. [Pressman, 1997]

Ohjelmiston kokoa voidaan arvioida myös dokumenttisivujen tai takaisinlaskennan avulla. Dokumenttisivujen laskentaa voidaan hyödyntää silloin, kun ohjelmisto on jo määrittelyvaiheessa kuvattu tiettyjä kiinteitä standardeja käyttäen. Tällöin dokumenttien rakenne pysyy vakiona ja sivumäärää voidaan käyttää kokoarvioinnin perusteena. Takaisinlaskentaa voidaan pitää välimuotona lähdekoodirivien ja toimintopisteiden laskennasta. Sen hyöty tulee esille ns. konversiotaululla, jolla voidaan muuntaa historiatietojen perusteella toimintopisteet lähdekoodirivimääräksi ja toisinpäin. [Pressman, 1997]

Taulukossa 5 on esitetty ohjelmiston koon erilaisten laskentatapojen soveltuvuutta erilaisiin ohjelmistohankkeisiin.

Projektityyppi	Rivimäärien laskeminen	Lausemäärien laskeminen	Toimintopiste-laskennat	Takaisin-laskenta	Dokumenttien sivumäärät
Uuskehitysprojektit	-	-	++	-	-
Lisäävä ylläpito	-	-	++	-	-
Muuttava ylläpito	+	++	+	++	+
Konversioprojektit	o	o	+	++	o
Vuotuinen kunnossapito	o	o	+	++	+
Valmisohjelmiston hankinta	-	-	+	-	o
Ulkoistamisprojektit	o	o	+	++	o

Taulukko 5. Ohjelmiston koon soveltuminen työmääräarvioon erilaisissa ohjelmistohankkeissa

++ tarkoittaa, että menetelmä sopii erinomaisesti
+ tarkoittaa, että menetelmää voidaan käyttää
o tarkoittaa, että menetelmää voidaan käyttää rajatuissa tapauksissa
- tarkoittaa, että menetelmä ei sovellu [Forselius, 1999].

Kuten taulukon 5 perusteella voidaan päätellä, erityisesti uuskehityshankkeissa toimintojen tunnistaminen etukäteen on tärkeää työmäärän arvioimista varten.

4.1.3. Ohjelmiston koon merkitys testauksen työmääräarviointiin

Ohjelmiston koko vaikuttaa luonnollisesti suoraan testauksenkin työmäärään. Erilaisten ohjelmiston koon arviointiin käytettyjen mallien soveltuminen testausprojektin työmäärän arviointiin riippuu siitä, minkä tyyppisestä testausprojektista on kyse. Edellä esiteltyä merkintätapaa noudattaen esitän oman kokemukseni pohjalta koosteen erilaisten testaustyyppien osalta taulukossa 6.

Testauksen tyyppi	Rivimäärien laskeminen	Lausemäärien laskeminen	Toimintopiste-laskenta	Takaisin-laskenta	Dokumenttien sivumäärät
Yksikkötestaus	o	o	o	-	-
Toiminnallinen testaus	-	-	++	-	o
Integrointitestaus	-	o	o	-	-
Systeemitestaus	-	-	++	-	o
Kuormitustestaus	o	-	+	-	-

Taulukko 6. Suuntaa antava arvio ohjelmiston koon arvioinnin soveltumisesta työmääräarvioon joissakin erityyppisissä testausprojekteissa

Rivi- ja lausemäärien arviointi on hyvin epävarmaksi tavaksi todettu jo ohjelmistoprojekteissa, joten sen soveltuvuus testausprojekteihinkin on huono. Myöskään takaisinlaskenta ei tällä näkökulmalla tuo apua testauksen työmääräarviointiin sen vaatiman historiatiedon vuoksi.

Toimintopistelaskenta auttaa kokoperustaisista arviointimalleista parhaiten hahmottamaan projektia testausnäkökulmasta. Yksikkötestausta voidaan sen avulla arvioida, jos on tiedossa ohjelmiston arkkitehtuuri ja koodaustyyli. Toimintojen määrä vaikuttaa suoraan toiminnallisen testauksen työmäärään. Toiminnallisessa testauksessa tulee tarkastaa jokaisen toiminnon oikeellisuus, joten toimintojen määrä antaa hyvän kuvan testauksenkin laajuudesta.

Integrointitestauksen toimintopistelaskenta ei välttämättä auta, ellei samalla ole saatavilla tietoa siitä, miten erilaiset toiminnot tulevat arkkitehtuurisesti sijaitsemaan, toisin sanoen mitkä toteutettavista toiminnoista integroidaan toimimaan yhdessä. Systeemitestaus perustuu integroituihin toimintoihin, joten sitä varten voidaan saada arvio toimintopisteiden kautta. Kuormitustestauksessa usein kuormitetaan toimintoja jostakin näkökulmasta (tietokanta, verkko, rajapinta), joten toiminnot auttavat jonkin verran päättelemään tarvittavaa työmäärää.

Dokumenttisivujen kohdalla pätee sama kuin toteutusprojekteissakin: jos käytössä on tiukka standardi, jolla dokumentit luodaan, niin ainakin projekteja toisiinsa vertaamalla saa käsitystä projektin koosta.

4.1.4. Ohjelmiston koon soveltuvuus työmääräarvioihin

Pelkän koon arviointi ja sen pohjalta tehtyä työmääräarviointia ei enää nykypäivänä pidetä kovin luotettavana. Kuten kuitenkin edellä todettiin, toimintopisteiden lukumäärällä voidaan jo kohtuullisesti arvioida ohjelmiston työmäärää sekä perinteisissä ohjelmistoprojekteissa että pelkissä testausprojekteissa.

Toisaalta kaikki juuri testaukseen vaikuttavat ominaispiirteet eivät välttämättä tule ilmi pelkästään toimintopisteiden lukumäärästä, joten vähänkään monimutkaisemmassa projektissa sen perusteella ei kovin luotettavaa työmääräarviota voida antaa.

4.2. Ohjelmiston kompleksisuus

Kahdella samalla kielellä toteutetulla ohjelmistolla voi olla sama määrä lähdekoodirivejä, mutta silti niiden vaatima työmäärä voi erota suurestikin. Tämä johtuu ohjelmiston monimutkaisuudesta. Ohjelmistotuotanto ei ikinä ole niin suoraviivaista kuin liukuhihnatyöskentely tehtaalla. Abstraktien rakenteiden muotoilu vaatii luovuutta matemaattisen älyn lisäksi. Mitä monimutkaisempia rakenteita ohjelmisto vaatii, sitä enemmän tarvitaan sekä luovuutta että älyä. [Hughes and Cotterell, 2006]

Ohjelmiston monimutkaisuudella tarkoitetaan sitä, miten paljon siitä on erilaisia rajapintoja ulospäin ja miten paljon erilaisia komponentteja pitää luoda. Monimutkaisessa ohjelmistossa jokainen rajapinta voi vaatia oman erillisen toteutuksensa, eikä niissä siten voida hyödyntää komponenttien uudelleenkäyttöä.

Asiakkaan kanssa sovitut vaatimukset ohjelmiston toiminnallisuuden suhteen on täytettävä. Ohjelmiston toteuttaja voi yrittää ohjata asiakasta vaatimusmäärittelyssä, mikäli asiakkaalle ei ole täysin selkeää, mitä ohjelmistolta halutaan. On hyödyllistä käydä määrittelyvaiheessa huolellisesti läpi asiakkaan tuottamat vaatimukset, sillä kompleksisuus lisää riskiä projektin epäonnistumiseen. Voi myös olla niin, ettei joku työmäärällisesti suuri ja monimutkainen komponentti olekaan kovin tarpeellinen. Jos projektin aikataulussa pysyminen on kriittinen tekijä, on järkevintä jättää riskiä kasvattava ei-pakollinen komponentti myöhemmin toteutettavaksi.

4.2.1. Ohjelmiston kompleksisuuden vaikutus työmääräarvioon

Yksinkertainen on kaunista, ja siihen tulisi ohjelmistotuotannossakin pyrkiä. Sen lisäksi, että kompleksiset ohjelmistorakenteet lisäävät ohjelmiston tekemiseen tarvittavaa työmäärää, ne myös altistavat suuremmassa määrin virheille, ja ohjelmiston monimutkaisuus korreloi virhemäärän kanssa. Tästä syystä monimutkaisen ohjelmiston testaamiseen ja virheiden korjaamiseen tulee varata enemmän aikaan kuin yksinkertaisen ohjelmiston korjaamiseen. [Hughes and Cotterell, 2006]

Yksinkertaisessa ohjelmistossa voidaan käyttää samaa tietyn palvelun toteuttavaa moduulia useaan kertaan, ja kun se on kerran testattu toimivaksi, sen voidaan luottaa toimivan oikein. Monimutkaisessa ohjelmistossa uudelleenkäyttöä ei voida hyödyntää, joten jokainen tuotettu toiminto tulee testata erikseen. Kun jokaista palvelua kohti joudutaan testaamaan erikseen kaikki erilaiset tapaukset, testauksen vaatima työmäärä nousee rajusti.

Monimutkainen ohjelmisto sisältää usein useita erilaisia rajapintoja ulospäin. Näiden rajapintojen toteutus vaatii monesti paljon selvitystä ja yhteistyötä rajapinnan toisen osapuolen ylläpitäjän kanssa. Työmääräarvioinnissa on huomioitava tämä jokaisen rajapinnan määrittelyyn erikseen kuluva aika. On myös mahdollista, ettei suora rajapintayhteys aina onnistu, ja tällöin voidaan joutua tekemään ylimääräisiä komponentteja, jotta

data saadaan kulkemaan. Näiden monimutkaisempien ratkaisujen toteutus vaatii aikaa ja osaamista, ja sekin tulee huomioida työmääräarviota tehdessä.

Näistä syistä johtuen monimutkaisen ohjelmiston työmäärän arviointi on vaikeampaa kuin kooltaan vastaavan yksinkertaisen ohjelmiston. Jos yrityksellä on runsaasti korkealaatuista ohjelmisto-osaamista, erityisesti monimutkaisten ohjelmistojen työmääräarvioinneissa voidaan ottaa hieman riskejä projektin voittamiseksi. Jos korkealuokkaista osaamista ei ole saatavilla, on parempi varmuuden vuoksi pyöristää työmääräarviot reilusti ylöspäin.

4.2.2. Ohjelmiston kompleksisuuden käyttäminen työmääräarviomalleissa

Joissakin työmääräarviomalleissa ohjelmiston monimutkaisuuden käsittely sisältyy ohjelmiston koon arviointiin; dokumenttisivujen laskenta on tästä hyvä esimerkki. Jos on käytössä kiinteä standardi, jonka mukaan ohjelmisto määritellään, tulee monimutkaisesta ohjelmistosta luonnollisesti enemmän sivuja dokumenttiin.

Toimintopistelaskentaa voidaan käyttää myös hieman monimuotoisemmin kuin vain yksinkertaisesti laskemalla toimintopisteet. Siinä voidaan huomioida ohjelmiston kompleksisuus sekä erilaisten toimintojen vaikeusaste. Tällöin lasketaan ensin luku varsinaisten toimintojen määrästä niiden vaikeusasteella huomioituna. Saadusta luvusta muodostetaan kompleksisuuden huomioivan laskentakaavan avulla lopullinen arvo. Ohjelmiston kompleksisuutta voidaan arvioida taulukkoon 7 merkittyjen kohtien avulla. Jokainen kohta arvioidaan asteikolla 0-5 (0 = ei vaikutusta, … ,5 = olennainen) sen mukaan, miten merkittävästä tekijästä kyseisen ohjelmiston kohdalla on kyse. [Helminen, 2008]

	Ulkoiset tekijät	Arvo
1	Varmistus ja toipuminen	0-5
2	Tietoliikenneyhteydet	0-5
3	Hajautettu datankäsittely	0-5
4	Suorituskyky	0-5
5	Käyttöympäristön vaatimukset	0-5
6	Dynaaminen datankäsittely	0-5
7	Transaktiotiheys	0-5
8	Reaaliaikainen päivitys	0-5
9	Käyttöönotto	0-5
10	Ohjelmiston toimintojen kompleksisuus	0-5
11	Lähdekoodin uudelleenkäytettävyys	0-5
12	Loppukäyttäjätehokkuus	0-5
13	Useat asennusympäristöt	0-5
14	Ylläpidettävyys	0-5
	Ulkoisten tekijöiden yhteisvaikutus	∑1-14
	Kompleksisuuskerroin: **0.65 + 0.01 * (∑1-14)**	

Taulukko 7. Toimintopistelaskennassa käytetyt tekijät arvioitaessa ohjelmiston kompleksisuutta [Pressman, 1997]

4.2.3. Ohjelmiston kompleksisuuden merkitys testausprojekteihin

Toimintopistelaskennan kompleksisuuskertoimen lista soveltuu melko hyvin työmäärän arvioimiseen testausprojektissakin. Listaa tosin kannattaa katsoa hieman eri näkökulmasta kuin miten sitä käytetään koko ohjelmistoprojektin työmäärää varten.

Varmistuksen ja toipumisen testaus vaatii ajallisesti melko paljon resursseja. Mitä tärkeämpi tämä ominaisuus on, sitä laajemmin kaikki erilaiset toipumista aiheuttavat tilanteet tulee testata. Eli tämän kohdan arvon kasvu korreloi testauksen työmäärän kasvun kanssa, joskin monien erilaisten toipumisskenaarioiden testaaminen vie todennäköisesti suhteessa enemmän aikaa kun niiden toteuttaminen. [Pressman, 1997]

Tietoliikenneyhteydet lisäävät testauksen työmäärää samoin kuin ohjelmoinninkin. Erityinen haaste testaukselle tulee siitä, että usein verkkoresurssit on jaettu muiden ohjelmistojen kesken. Erityisesti on huomioitava se, että Internetin yli toimivien verkkoa käyttävien sovellusten testaus on työläämpää kuin vain lähiverkossa toimivien. [Mosley, 2000]

Hajautettu tiedonkäsittely lisää testauksen työmäärää siinä missä ohjelmoinninkin. Testauksessa tulee huomioida kaikki erilaiset variaatiot tiedon välittämiselle sekä testata erilaiset virhetilanteet.

Jos suorituskyky on ohjelmiston kannalta epäolennainen asia, sitä ei tarvitse myöskään testata. Jos suorituskyky on tärkeä, sitä varten pitää todennäköisesti luoda omat suorituskykytestit ja niitä varten testiympäristö. Hyvä ohjelmoija tekee suorituskykyistä koodia lähes automaattisesti. Jos ohjelmiston toimintaympäristö on haastava, suorituskykyisen koodin tekeminenkin on luonnollisesti työläämpää. Mitä tärkeämpi rooli suorituskyvyllä on, sitä suuremmaksi kasvaa testauksenkin työmäärä, koska sitä laajemmat ja monipuolisemmat suorituskykytestit ympäristöineen pitää rakentaa. [Pressman, 1997]

Käyttöympäristön vaatimukset ovat niin laitteistoläheisiä asioita, että ohjelmiston testauksessa ne eivät näyttele työmäärällisesti merkittävää roolia.

Dynaaminen tiedonkäsittely tuo runsaasti haasteita testaamiseen. Sen lisäksi, että ohjelman tulee toimia oikein oikeantyyppisillä syötteillä, sen tulee myös tarjota käyttäjälle apua oikeanlaisten syötteiden muodostamiseen. Toisien sanoen, ohjelma ei saa kaatua vääränlaisesta syötteestä, vaan sen tulee antaa käyttäjälle riittävän informatiivinen palaute, jotta käyttäjä osaa sen jälkeen korjata toimintaansa. Ohjelmassa ei saa myöskään sisältää haavoittuvia kohtia, eli turvallisuus tulee taata. Käytännössä tällä tarkoitetaan esimerkiksi sitä, että käyttäjä ei voi tekstikenttään syöttää suoraan SQL-kielistä lausetta, jolla tuhoaa jonkun rivin tietokannasta. Dynaamisuus tuo mukanaan paljon testattavaa ja testauksen työmäärä kasvaa tällaisten ominaisuuksien mukana enemmän kuin ohjelmoinnin. Testaaja ei voi luottaa ohjelmoijan tehneen turvallista koodia, vaan lähtökohtaisesti testaajan tulee yrittää saada ohjelmisto toimimaan tietoturvattomasti. Tämä kasvattaa testauksen työmäärää siitä huolimatta, että turva-aukkoja ei ohjelmistosta löytyisi. [Mosley, 2000]

Transaktiotiheys liittyy osaltaan suorituskykyyn. Jos transaktioilla ei ole mitään merkittäviä huippuja, ei ole myöskään erityisiä vaatimuksia testaamiseen. Osittain tämä aihe liittyy suorituskykytestaukseen, ja mahdolliset transaktiohuiput tulee huomioida testauksessa. Jos kyseessä on ohjelmisto, jossa transaktioiden määrä on erittäin suuri, siitä todennäköisesti päätetään jos suunnitteluvaiheessa tehdä kolmitasoinen, jolloin keskimmäinen taso tasaa kuormaa tietokantaan nähden. Tällaisessa tapauksessa testaukseen tulee yksi kerros lisää; tosin se vaikuttaa vastaavasti myös ohjelmointiin. [Mosley, 2000]

Reaaliaikainen päivitys lisää testauksen työmäärää siinä mielessä suhteessa enemmän, että mitä enemmän on erilaisia tapauksia, joiden ollessa kesken on päivitys voitava suorittaa, sitä enemmän on testitapauksia. Asennusten testaamiset ovat aina suhteellisen työläitä, sillä pitää useaan kertaan toistaa tilanteita, joissa on pohjalla jokin tietty versio, ja sitten siitä päivitetään johonkin toiseen versioon. Tämä pitää vielä mahdollisesti tois-

taa usean erilaisen komponentin ollessa käytössä erilaisin kombinaatioin, joten testitapausten määrän kasvaminen kasvattaa rajusti testaamisen työmäärää.

Käyttöönotto voi ohjelmiston kanssa tapahtua joko ns. puhtaalta pöydältä, jolloin tällä kohdalla ei ole merkitystä, tai toisessa ääripäässä vanha data pohjille konvertoiden. Jos ohjelmaa voi alkaa vain käyttää (esimerkiksi selaimessa toimiva ohjelmisto), ei tämä kohta juuri vaadi testaushuomiota. Jos ohjelmiston asennus tai käyttöönotto vaatii dokumentaation, dokumentaatio tulee testata. Näissä tapauksissa työmäärä pysyy vielä linjassa kehitystyön suhteen. Jos ohjelmiston käyttöönottaminen automaattisesti esimerkiksi konvertoi edellisen käytössä olleen ohjelmiston datan itselleen sopivaksi, tämä kasvattaa erityisesti testauksen työmäärää runsaasti. Kaikki mahdolliset kombinaatiot tulee konvertoinnin suhteen testata.

Ohjelmiston toimintojen kompleksisuus vaikuttaa kehitystä vastaavasti suoraan kasvattaen myös testauksen työmäärään. Monimutkaiset ulkopuoliset liitännät, suuri määrä erilaisia poikkeustilannekäsittelyjä ja muut vastaavat kompleksiset tekijät lisäävät samassa suhteessa työmäärää toteutuksessa ja testauksessa.

Lähdekoodin uudelleenkäytettävyys poikkeuksena aiempiin laskee testauksen työmäärää vaikka se nostaa toteutuksen työmäärää. Käyttöliittymäkontrollit ovat hyvä esimerkki tällaisesta. Kun kontrolli on testattu kertaalleen, sen voidaan luottaa toimivan samalla tavalla muuallakin. Jos uudenkäyttöä ei tehdä, jokainen kontrolli joudutaan testaamaan alusta lähtien kokonaan. Testauksen työmäärässä säästetään tekemällä ja suunnittelemalla mahdollisimman uudelleenkäytettävää koodia. Aina toki uudelleenkäyttö ei tapahdu saman ohjelmiston sisällä. [Pressman, 1997]

Loppukäyttäjätehokkuudella tarkoitetaan paljolti käytettävyyttä. Jos käytettävyyttä ei ole ajoissa huomioitu ohjelmiston arkkitehtuurisuunnittelussa, se saattaa merkittävästi lisätä toteutuksen työmäärää. Toisaalta taas toteutuksen ja testauksen työmäärät kasvavat lineaarisesti käytettävyysvaatimusten suhteen. [Folmer and Bosch, 2005]

Useat asennusympäristöt todennäköisesti kuormittavat suhteessa enemmän toteutusta kuin testausta. Jotkin tekniset ratkaisut saattavat aiheuttaa suuressakin mittakaavassa ongelmia ja siten työtä, jos ohjelmistoa pitää voida käyttää samanaikaisesti useammasta paikasta. Testaaminen tämän suhteen on kuitenkin suoraviivaista. Siten tämän kohdan merkittävyys työmäärän kasvamiseen on voimakkaammin toteutukseen liittyvä.

Ylläpidettävyys tuottaa testaukselle työmäärää samalla tavoin kuin kohta toipumisesta ja varmistuksista. Mitä automaattisemmin ohjelmiston tulee suoriutua erilaisissa tilanteissa, sitä enemmän erilaisia skenaarioita pitää testata.

4.2.4. Ohjelmiston kompleksisuuden soveltuvuus työmääräarvioihin

Ohjelmiston kompleksisuuden huomioiminen tuo luotettavampaa tietoa ohjelmiston työmäärästä. Siinä huomioidaan lähtökohtaisesti monet ohjelmiston mahdollisesti sisältävistä monimutkaisista ja raskaista rakenteista. Toisaalta tämäkin menetelmä on hyvin teoreettinen, sillä ei huomioida muuttuvien tekijöiden vaikutusta.

Kuten edellä tuli ilmi, kompleksisuuskertoimeen vaikuttavat tekijät saattavat vaikuttaa eri tavoilla toteutus- ja testausprojekteihin. Merkittäviä nämä erot ovat silloin, kun testauksessa tulee toistaa joitakin työläitä tilanteita monilla erilaisilla kombinaatioilla tai kun toteutuksessa panostetaan uudelleenkäytettävyyteen, Vastaavasti tämä alentaa testauksen työmäärää, kun samaa testattua moduulia käytetään useammassa paikassa.

5. Muita työmääräarvioissa huomioitavia tekijöitä

5.1. Historiatieto

Vaikka projektien sisältö ja ihmiset vaihtuvat, voidaan jo valmistuneiden projektien toteutuneita työmääriä käyttää apuna arvioitaessa uuden projektin työmäärää. Historiatietoa käytetään muuttujana matemaattisissa malleissa. [Helminen, 2008] On todettu, että yritykset hyötyvät myös toisten organisaatioiden toteuttamien projektien historiatiedosta [Wieczorek, 2002]. Mitä laajemmin historiatietoa on käytössä, sitä paremmin sen perusteella voidaan tehdä työmääräarvioita.

Usein historiatietoa muunnetaan numeerisiksi arvoiksi, joiden avulla hyvinkin erilaiset projektit saattavat tuoda lisäinformaatiota joidenkin tekijöiden suhteen. Tässäkin muunnoksessa inhimillisen virheen mahdollisuus on suuri. Mitä paremmin yrityksellä on standardoituna erilaiset työmääräarvion osatekijöiden muunto numeeriseen muotoon, sitä paremmin yksittäiset ihmiset osaavat arvottaa samantasoisia tekijöitä toisiaan vastaaviksi. [Wieczorek, 2002]

Historiatiedon hyväksikäyttäminen on ehdottoman kannattavaa pitkissä ylläpitoprojekteissa, joissa uudetkin tehtävät ovat osa vanhaa rakennetta. Tällöin on yleensä saatavilla historiatietoa edellisen vaiheen päivitysten työmääristä, joista saadaan lähtökohta uusille työmääräarvioille. Ylläpitoprojektit ovat hyvä tapa ajaa sisään uusia työntekijöitä. Niiden työmääräarviointi on kohtuullisen helppoa, joten uuden työntekijän käyttämän ajan perusteella voidaan muodostaa jonkinlainen kuva juuri tämän henkilön työtehosta.

Historiatiedon käyttäminen pitäisi olla periaatteessa erittäin helppoa yrityksille, koska yleensä on käytössä kuitenkin jonkinlainen raportointijärjestelmä, johon jokainen työntekijä syöttää tuntinsa. Tässä tulee esille ensimmäinen ongelma asian suhteen, sillä harvoin tuntienkirjausjärjestelmissä on riittävästi erilaisia vaihtoehtoja tuntien kirjaamiseksi. Käytännön esimerkkinä olisi järkevää erotella puhdas toteutustyö erilleen mm. asiakkaan kanssa sähköpostitse käytävästä tarkentavasta keskustelusta.

Toisaalta yrityksellä tulisi olla käytössä jonkin suoraviivainen metodi, jolla ajankäyttö saadaan suoraan henkilöimättä tuntienkirjausjärjestelmästä työmääräarvioinnin käytettäväksi. Työntekijöillä voisi olla tuntienkirjausjärjestelmässä luokitus, joka perustuisi työkokemukseen. Täten saataisiin raportteja, joiden avulla voitaisiin erotella eri luokituksen omaavien työntekijöiden tekemiä työmääriä jo toteutuneissa projekteissa. Tällaisen historiatiedon olettaisi olevan erittäin hyödyllistä tulevia projekteja ajatellen.

Historiatietoa voidaan oman kokemukseni mukaan hyödyntää testausprojekteissa samoin kuin toteutusprojekteissakin. Esimerkiksi selainpohjaisen sovelluksen testaamiseen eri selaimissa kuluu jokseenkin sama työmäärä. Pitkällä aikavälillä vastaavasti

saataisiin dataa siitä, löytyykö jostain tietystä selaimesta suhteessa enemmän vikoja kuin muista.

5.2. Projektin henkilöstö

Työmääräarvio ja etenkin aikataulu vasta antavat viitteitä siitä, kuinka paljon henkilöstöä projekti tarvitsee. Kuitenkin jo työmääräarviota varten olisi syytä olla tiedossa, millaisen kokemustason omaavia ihmisiä projektiin on laittaa, sillä tämä asia vaikuttaa oleellisesti työmääräarvion suuruuteen. Toinen mahdollisuus on tietenkin laskea keskimääräisen toteuttajahenkilön työpanoksella arvio, jota sitten päivitetään suuremmaksi tai pienemmäksi tarpeen mukaan, kun todellinen henkilöstö projektiin selviää.

Asiantuntijoita on rajallisesti ja heidän aikansa on kallista. Tästä syystä projekteissa joudutaan käyttämään myös vähemmän kokeneita henkilöitä. Työmäärää arvioitaessa on kyettävä arvioimaan, miten paljon tehokkuus kärsii vähemmän osaavan tekijän tapauksessa ja huomioitava tämä seikka aikataulussa Kokemattomien arvioijien työmääräarviot ovat todennäköisemmin pienempiä kuin kokeneiden arvioijien tekemät työmääräarviot [McDonald, 2005]. Tämä asia korostuu entisestään, jos joltain taholta tulee painetta aikataulun tai budjetin kiristämiseen.

Toisaalta projektin onnistumiseen vaikuttaa myös henkilökemiat. Toisten ihmisten välillä kommunikaatio sujuu paremmin ja tiedonjakaminen on nopeaa. Hyvin yhdessä toimivaa projektiryhmää ei kannata hajottaa. On myös todettu, että eri alueiden asiantuntijoista koostuvan ryhmän yhteinen tiedonjako saa yksilöiden työmääräarviot muuttumaan realistisempaan suuntaan. Tämä johtuu todennäköisesti siitä, että tietämys määriteltyjen ominaisuuksien todellisesta vaikeusasteesta selviää keskustelemalla muiden asiantuntijoiden kanssa. [Moløkken-Østvold and Jørgensen, 2004]

Myös asiakkaan yhteyshenkilöllä on merkitystä. Jos yhteyshenkilö on erittäin kiireinen, saattaa vasteajat kysymyksiin olla pitkiä eikä määrittelypalavereille riitä aikaa. Asiakkaan sitoutuminen projektiin huomioidaankin yleensä riskinä projektille, jos tällainen tilanne on mahdollinen. Hyvin tehty työmääräarvio ei yksin riitä, jos koko projektihenkilöstö joutuu odottamaan asiakkaalta saatavia selvennyksiä asioihin.

Projektipäällikkö on yksittäisistä henkilöistä tärkein projektille. Projektipäällikön kokemus ja vankka osaaminen tasapainottaa muun henkilöstön kokemattomuutta. Työmääräarviolaskelmissa kannattaakin pisteyttää projektipäällikön kokemuksesta enemmän, koska sen on todettu vaikuttavan kokemusperäisistä asioista eniten. [McDonald, 2005]

Työmääriä arvioitaessa on otettava huomioon myös se, että henkilöstöä voidaan joutua vaihtamaan projektin edetessä. Uuden ihmisen perehdyttämiseen kuluu aikaa. Tätä riskiä voidaan toki pienentää ennalta, jos ihmiset on saatu sitoutettua yritykseen hyvin. Jos yrityksessä henkilöstön vaihtuvuus on suurta, voidaan olettaa, että projektin aikana tulee henkilöstövaihdoksia tapahtumaan ja silloin tämäkin asia pitää huomioida työmääräarvioissa.

Projektin kokemustaso voidaan muuntaa matemaattiseen muotoon esimerkiksi muuttamalla jokaisen henkilön kokemus numeeriseksi arvoksi ja laskea niistä keskiarvo. Täten saadaan huomioitua työkokemuksen vaikutus työmääräarvioinnissa. [McDonald, 2005]

Toisaalta on esitetty, ettei millään matemaattisella kaavalla ole mahdollista erotella eri toteuttajien erilaista työtahtia tehdä sama asia. Collopy [2007] toteaa, että samaan ongelmaan tuotetut ratkaisut ja ratkaisuun kuluneet ajat eroavat niin moninkertaisesti toisistaan, ettei erotusta ole matemaattisen kertoimen avulla mahdollista päätellä.

Näistä voidaan päätellä, että työmääräarvioita ei tule koskaan antaa kokemattomien henkilöiden tehtäväksi. Arvion tekevään projektitiimiin tulisi kuulua vähintään yksi henkilö, jolla on kokemusta vastaavasta projektista, jotta työmääräarviota voidaan pitää realistisena ja käyttää arviota hyödyksi projektin suunnittelussa. Hyödyllisintä olisi, jos jokaiseen työmääräarvioon olisi käytettävissä edellä mainittu asiantuntijaryhmä, eikä siten mikään projektin näkökulma jäisi vaille huomiota.

Kokemus ja tietämys vaikuttavat myös testausprojekteissa. Toisaalta taas pitkään samaa ohjelmistoa testannut sokeutuu sen käytettävyydelle ja uusia virheitä saattaa tulla esiin uuden henkilön alkaessa testata.

Testauksen voidaan sanoa vaativan tietynlaista luonnetta. Testaajien toimenkuvaan kuuluu etsiä virheitä toisen työstä, joten kyky etsiä mahdolliset ongelmakohdat on hyödyksi. Paineensietokyky on testaajille eduksi, sillä työ on keskittymistä vaativaa ja aikaa on rajallisesti. Hyödyllisiä luonteenpiirteitä testaajalle ovat päättäväisyys, keskittymiskyky ja hyvä huomiokyky. [Capretz and Ahmed, 2010]

Kehitystiimin osaamistaso vaikuttaa kokemukseni perusteella hyvin voimakkaasti nimenomaan testauksen työmäärään. Kokemattomat ohjelmoijat tekevät enemmän virheitä ja heillä kuluu enemmän aikaa löytää virheiden alkulähde. Kokemattomille suunnittelijoille testaaja joutuu kirjoittamaan tarkemman virheenkuvauksen, ja jos virheraportteja pitää kirjoittaa paljon, tämä aika on pois varsinaisesta testaamisesta.

5.3. Ympäristötekijät

Ohjelmistoprojekti on kuitenkin aina myös paljon muuta kuin suoraviivaista koodinkirjoittamista. Ohjelmiston määrittelevät ihmiset, koodia kirjoittavat ihmiset ja lopputuloksen onnistumisesta tai epäonnistumisesta päättävät ihmiset.

Kaikkea ympäristöön liittyvää tietoa ei välttämättä ole saatavilla vielä siinä vaiheessa, kun projektille pitäisi tehdä ensimmäiset työmääräarviot. Ei ehkä ole tietoa siitä, koska projekti tulisi alkamaan, eikä tällöin voida tietää, ketä henkilöitä projektiin olisi saatavilla. Mitä vähemmän tietoa on tarjolla, sitä enemmän pitää arvioida työmäärää yläkanttiin. Toisaalta joskus on mahdollista solmia erittäin tärkeitä uusia asiakassopimuksia, jolloin yhden projektin voitollisuus ei ole pitkän aikavälin kannalta merkittävä tekijä.

Hankalat tekniset ympäristöt tuovat haastetta testausprojekteille. Etenkin jos ollaan tekemisissä uusien teknologioiden kanssa, voi testausympäristön toimintakuntoon saattaminen viedä huomattavasti oletettua enemmän aikaa. Jos testattava ohjelmisto tukee monia ympäristöjä, on huomioitava, että testaajalla menee aikaa jokaisen erilaisen ympäristön toimintakykyiseksi saattamiseen. Toisaalta haastava ympäristö voi olla matkapuhelinverkkoon liittyvä. Esimerkiksi testattaessa matkapuhelinverkon ominaisuuksia voi testaaja joutua vaihtamaan sijaintiaan 2G- ja 3G-verkon välillä tai mahdollisesti vaihtaa solua samassa verkossa. Tämä vaatii vähintään liikkumista sekä tutustumista siihen, miten matkapuhelinverkko todellisuuteen suhteutuu.

5.4. Liiketoimintaosaaminen

Jokaisella liiketoiminnan alueella on omat erityispiirteensä. Jos ohjelmistoa tekevillä ja suunnittelevilla henkilöillä on vahva tietämys kyseisestä liiketoiminta-alueesta, on todennäköistä, että projektin päämäärä on selkeä. Jos liiketoiminta-alue on tuntematon, kuluu oma aikansa siihen, että projektin päämäärä ja tarkoitus selviävät.

Työmääräarvioinnin kannalta tämä asia tulee huomioida etenkin monimutkaisten projektien kohdalla. Jos tekijöillä ei ole selkeää käsitystä siitä, mitä he ovat tuottamassa, voi tämä kostautua moninkertaisena työmääränä, kun asioita joudutaan tekemään uudelleen. Ketterät ohjelmistokehitysmallit tarjoavat tähänkin asiaan ratkaisua, sillä niissä kehityksen ohessa on koko ajan myös asiakkaan edustaja varmistamassa tuotteen kehityssuunnan oikeellisuutta. [Dubinsky and Hazzan, 2004]

Liiketoiminta-alue myös määrittelee usein joukon suorituskyky- ja luotettavuustekijöitä. Esimerkiksi pankkiasiointiin liittyvällä ohjelmistolla on suuret luotettavuusvaatimukset verrattuna mobiililaitteessa toimivaan peliin. Jos tietokantajärjestelmässä on paljon transaktioita, niin se vaatii panostusta suorituskykyyn toisella tavalla kuin yrityksen sisäinen vianhallintasovellus. Erilaisten projektille spesifisten ympäristövaatimusten täyttäminen lisää projektin työmäärää.

Testauksen näkökulmasta liiketoiminta-alueen tuntemus tulee esille etenkin katselmoinneissa. Jos määrittelyiden tekijä ymmärtää liiketoiminta-alueen, ovat määrittelyt todennäköisemmin lähempänä oikeaa. Tästä voisi päätellä, että jos määrittelyjen kirjoittaja ei ole liiketoiminta-alueen osaaja, olisivat katselmoinnit erityisen tärkeitä.

5.5. Ohjelmistoprosessien vaikutus testauksen työmäärään

Ohjelmistoprosessin kypsyys auttaa tehokkaampaan ajankäyttöön projektissa. Tällöin asiat sujuvat aina samojen standardien mukaisesti eikä asioita jää huomioimatta. Jos prosessia ei ole, todennäköisyys ajan hukkakäyttöön kasvaa. [Wieczorek, 2002]

Esimerkkinä ohjelmistoteollisuuden kypsyysmallista toimii CMMI. Malli sisältää avainprosessialueet, joiden tulisi kuulua ns. kypsän yrityksen normeihin. Mitä enemmän näitä mallin sisältämiä prosesseja ja vaatimia dokumentaatioita yrityksellä on todistettavasti käytössä, sitä korkeamman kypsyysluokituksen yritys voi saada. Eli mitä kypsem-

mäksi yritys katsotaan, sitä enemmän voidaan olettaa yrityksellä olevan standardoituja vakiomalleja erilaisiin prosesseihin yrityksen tuotekehitysalueella. Kun prosessit on standardoitu, ei pyörää tarvitse keksiä jokaisessa projektissa uudelleen, vaan voidaan ilman ajan hukkakäyttöä noudattaa prosessin ohjaamaa käytäntöä. [Chrissis et al., 2003]

Etenkin yrityksen kommunikaatiokulttuuri on oleellinen osa prosessia. Turhat palaverit vievät aikaa, mutta vajavainen kommunikaatio hidastaa tiedonkulkua ja siten työtehoa. Ketterät ohjelmistokehitysmallit korostavat projektiryhmän keskinäistä kommunikaatiota. Ketteriä menetelmiä noudatettaessa yleensä pyritään saattamaan koko projektiryhmä samaan tilaan, jossa kommunikaatio voidaan hoitaa välittömästi kasvotusten ja virheen mahdollisuus kirjoitetun tekstin väärinymmärtämisestä pienenee. [Cohen et al., 2004]

5.5.1. Ketterän ohjelmistoprosessin erityispiirteet testaukseen

Kuten edellä on esitelty, perinteisessä ohjelmistoprojektimallissa testaus usein perustuu dokumenteissa esiteltyihin vaatimuksiin ja testatessa validoidaan ohjelmiston toiminta näitä vaatimuksia vasten. Ketterän ohjelmistoprosessin toimintatapa on kuitenkin hieman toisenlainen, joten perinteistä jaottelua toteutuksen ja testauksen vastuualueista ei voida samalla tavalla tehdä.

Ketterät menetelmät perustuvat yksilöiden vuorovaikutukseen ja ohjelmiston iteraatioon enemmän kuin dokumentointiin ja jäykkiin prosesseihin. Ohjelmistotiimi kommunikoi paljon suullisesti, ja tilaajan puolelta on edustaja mukana seuraamassa ohjelmiston kehitystä. Eli ohjelmistoa toteutetaan pienissä paloissa, ja palaute sen edistymisestä ja suunnitelmat seuraavista toteutettavista osista on lähinnä suullista. [Martin, 2003]

Koska ketterässä ohjelmistoprojektissa ei voida perinteiseen tyyliin testata suoranaisesti määrittelydokumentaatiota vasten, eräs mahdollinen toimintatapa on testivetoinen kehitys. Testivetoisessa ohjelmistoprosessissa aloitetaan testitapausten kirjoittamisella, ja se kuuluu ohjelmoija työnkuvaan. Sen jälkeen ohjelmoija työstää varsinaista ohjelmistoa, kunnes testitapaukset menevät läpi. Viimeisessä vaiheessa ohjelmoija ikään kuin jälkikäteen suunnittelee toteuttamansa koodin. Viimeistä vaihetta kutsutaan refaktoroinniksi, ja sen on tarkoitus parantaa koodin laatua ja ylläpidettävyyttä. Ilman refaktorointia testivetoinen ohjelmointiprosessi olisi vain tapa tuottaa nopeasti koodia, jonka laadulla ei ole merkitystä. Testivetoisen kehityksen ohjenuora on, että yhtään ohjelmakoodia ei saisi sijaita testitapausten ulkopuolella. [Makkonen, 2008]

Kuva 7. Testivetoisen ohjelmistokehityksen vaiheet [Makkonen, 2008]

Ketterät menetelmät perustuvat vain lyhyen tähtäimen suunnitteluun. Tällä on tarkoitus pitää kokonaisuudet riittävän pieninä ja hallittavina, jolloin tekijät saavat koko ajan huomata edistymistä ja saavat tästä onnistumisen tunteita. [Makkonen, 2008]

Testivetoisessa kehityksessä yksikkötestaus siis suoritetaan korkea laatu päämääränä. Kuitenkin se kattaa vain koodin laadullisen verifioinnin. Jotta voidaan varmistua siitä, että kehitettävä ohjelmisto on se, minkä asiakas on tilannut, voidaan validointi huomioida hyväksymistestausvetoisella ohjelmistokehityksellä. Ketterissä metodeissa asiakkaan vastuuhenkilö on osallisena projektissa ja siten häntä voidaan käyttää resurssina suunniteltaessa hyväksymistestitapauksia. Anderssonin ja kumppaneiden [2003] tutkimuksessa asiakkaan vastuuhenkilö antoi ohjelmoijille suuntaviivoja siitä, mitä testitapausten tulisi sisältää. Sitten testitapauksia iteroitiin kunnes ne vastasivat asiakkaan mielikuvaa. Päinvastoin kuin testivetoisessa ohjelmistokehityksessä Anderssonin ja kumppaneiden mallissa ei välttämättä käytetä ollenkaan yksikkötason testausta, vaan luotetaan siihen, että virheet tulevat ilmi hyväksymistesteillä. [Andersson et al., 2003]

Toinen tapa hankkia käyttötapauskuvauksia asiakkaalta on käyttötapaustarinat. Käyttötapaustarinoiden avuilla kerrotaan, miten ohjelmiston tulisi toimia erilaisissa tilanteissa. Käyttötapaustarinat voivat olla kieleltään lähellä normaalia kommunikointia, ja niiden pohjalta voidaan suunnitella hyväksymistestitapaukset yhdessä ohjelmoijien, testaajien ja asiakkaan kanssa. [Makkonen, 2008]

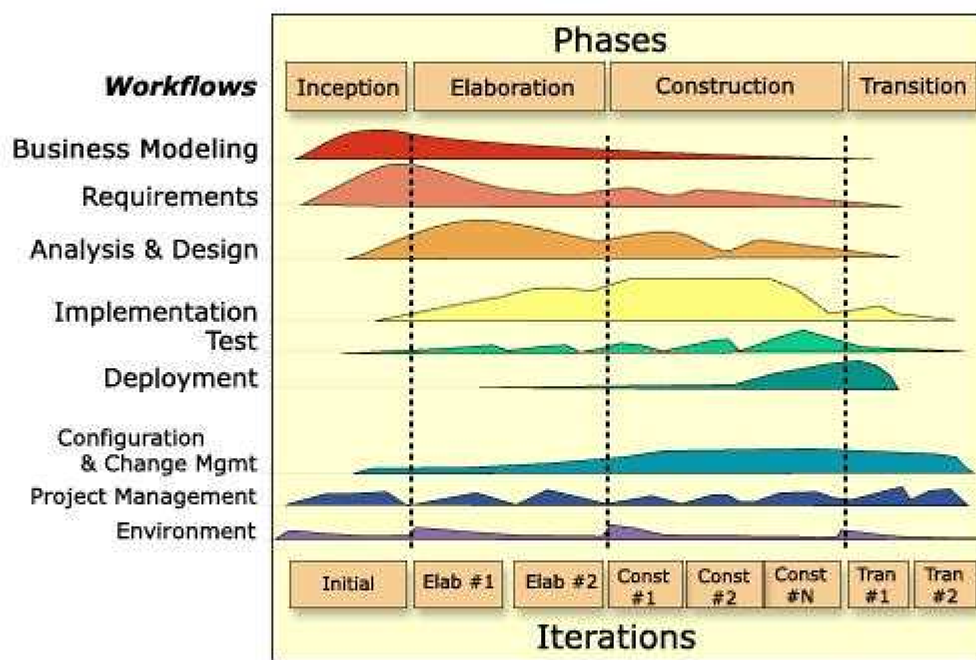
Ketterät menetelmät kannustavat ohjelmoijia testaamaan koodinsa hyvin ennen sen luovuttamista eteenpäin. Hyvänlaatuinen testattava koodi luonnollisesti vähentää testauksen työmäärää. Koska ketteriä menetelmiä käytettäessä dokumentointi on hyvinkin kevyttä, tulee testaajalle uusi velvollisuus kommunikoida asiakkaan kanssa ohjelmiston vaadituista toiminnoista. Tähän tehtävään kuluva työmäärä voi vaihdella suurestikin asiakkaasta ja projektista riippuen. On normaalia, että käyttäjät unohtavat kertoa joitakin oleellisia ehtoja ohjelman toiminnasta, joten osa virheistä todennäköisesti jää asiakkaan löydettäviksi. [Pyykkö, 2010]

Pyykön [2010] mukaan testaajan työkuorma ketterässä ohjelmistoprosessissa pysyy ennustettavana, sillä kerrallaan työn alla olevan iteraation läpimenoaika on lyhyt. Kuitenkin testaajien on oltava koko ajan valmiudessa vastata muihinkin kuin ennustettuihin testaustarpeisiin. Testaajan on erittäin tärkeä pysyä hyvin selvillä projektin tilanteesta,

jotta työmäärää ei kulu hukkaan keskeneräisiä ominaisuuksia testaamalla tai vääriä oletuksia vaatimuksista tekemällä. [Pyykkö, 2010] Näistä syistä johtuen ketterä ohjelmistoprosessi tekee testauksen ulkoistamisesta melko haastavaa.

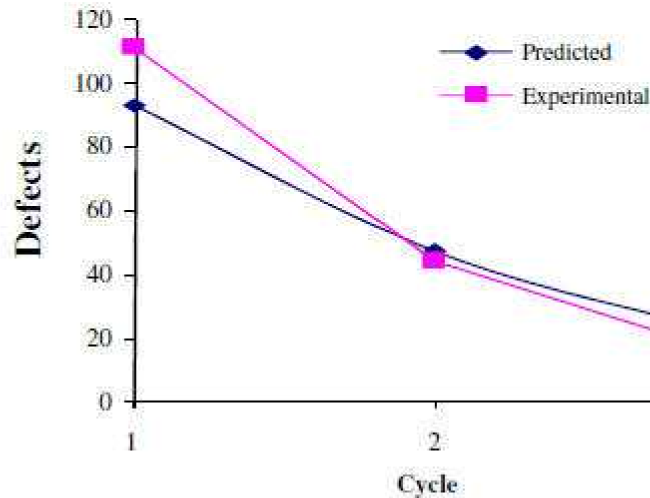
5.5.2. Yhtenäistetyn ohjelmistokehitysprosessin erityispiirteet testaukseen

Yhtenäistetty ohjelmistokehitysprosessi (engl. Rational Unified Process, RUP) perustuu neljään tarkoin dokumentoituun vaiheeseen, joita toistetaan iteratiivisesti. Alkuvaihe, kehitysvaihe, valmistusvaihe ja siirtymävaihe sisältävät eri asioita. Kaikki nämä vaiheet kuuluvat jokaiselle jaksolle, mutta painotus vaihtelee. Kuvassa 8 on havainnollistettu RUP:n kaksiulotteista prosessimallia. [Kruchten, 2003]



Kuva 8. RUP:n vaiheiden ja työvirtojen sijoittuminen projektissa, horisontaalisesti aika ja vertikaalisesti projektin sisältö [Kruchten, 2003]

Yksi tapa arvioida testauksen työmäärää on arvioida ohjelmistovirheiden määrä etukäteen. Mohan ja kumppanit [2008] tutkivat RUP:a noudattavan ohjelmistokehitysprosessin virheiden määrän arviointia sumean logiikan avulla ennen varsinaisen projektin aloittamista. RUP:n iteratiivinen kehitysmalli aiheuttaa virheiden selkeän vähenemisen projektin loppumista kohden. Ja toisaalta RUP-prosessiin kuuluva prosessin räätälöinti juuri kyseisen projektin tarpeisiin helpottaa ennustamista. Kuten kuvasta 9 nähdään, sumean logiikan yhdistäminen RUP:iin antaa melko osuvan kokonaiskuvan ohjelmiston virhemäärästä etukäteen arvioituna. [Mohan et al., 2008]



Kuva 9. Virheiden määrä sekä sumealla logiikalla arvioituna että kokeellisesti [Mohan et al., 2008]

RUP:ssa testauksen työnkuvaan kuuluu löytää ohjelmiston viat ja heikkoudet, ohjeistaa hallintoa laatutasosta, arvioida suunnittelu- ja määrittelydokumenteissa esitettyjä oletuksia konkreettisella kokeilulla, varmistaa ohjelmiston toimivuus siten kuin on määritelty ja varmistaa, että vaatimukset on toteutettu kunnolla. RUP sisältää testaukseenkin erilaisia rooleja, tunnustaa monia erilaisia testaustasoja ja kannustaa hyvään dokumentointiin niin ennen kuin jälkeen itse testaamisen. Testauksen työnkulku muodostaa RUP:ssa tärkeän palautemekanismin, jonka perusteella arvioidaan projektin tilaa. RUP myös tukee testausta lukuisilla testaustyökaluilla. [Kruchten, 2003]

RUP:n mahdollistama räätälöinti antaa hyvät työkalut spesifioida prosessia projektikohtaisesti. Testaus koetaan tärkeänä ja suunnittelun arvoisena osana prosessia ja sitä tukemaan on kehitetty laaja valikoima testaustyökaluja. Prosessi on tarkkaan dokumentoitu ja sitä hallinnoidaan huolellisesti. Näistä syistä RUP antaa tehokkaat mahdollisuudet myös testauksen työmäärän arvioimiseen ennalta ja siten mahdollistaa myös testauksen ulkoistamisen hallinnoidusti.

5.6. Ohjelmistotekniikan vaikutus testaukseen

Olioperustainen ohjelmistokehitys on yleistynyt ja nykyisin laajalti käytössä. Esimerkkejä olioperustaisista ohjelmointikielistä ovat C# ja Java. Kuitenkaan pelkkä oliokieli ei vielä merkitse olioperustaisuuden hyödyntämistä, oliokielelläkin voi ohjelmoida ilman varsinaisen olioarkkitehtuurin käyttämistä. Tässä tutkielmassa kuitenkin lähdetään siitä olettamuksesta, että olioperustaisessa ohjelmistokehityksessä käytetään sitä tukevaa ohjelmistoarkkitehtuuria.

Olioperustainen ohjelmistokehitys tarkoittaa sitä, että järjestelmän osat kuvataan olioina. Olioilla on suhteita toisiinsa ja niillä on ominaisuuksia ja toimintoja. Olioperustainen menetelmä tuo järjestelmän rakentamiseen systemaattisuutta, se helpottaa järjestelmien ylläpitoa ja sen avulla voidaan lisätä komponenttien uudelleenkäytettävyyttä.

Olioperustaisuus vastaa ihmisen tapaa hahmottaa järjestelmän toimintaa, ja siksi sillä on merkitystä myös ohjelmiston testauksen näkökulmasta. [Koskimies, 2000]

Jokainen olio voidaan ajatella omana kokonaisuutenaan, jonka tulee toimia tietyllä tavalla. Olioiden suhteet toisiinsa voidaan ymmärtää järjestelmän toimintojen suhteina toisiinsa. Olioiden testaaminen on siten hieman vähemmän abstraktia kuin prosedulaarisen koodin. Toisaalta taas olion testaaminen on enemmän kuin vain syötteiden verifiointia, sillä oliolla on myös käsitteenä tila ja mahdollisia eritasoisia suhteita toisiin olioihin. [Binder, 1996]

Ohjelmiston virhealttiiden kohtien löytäminen kuuluu testaajaan toimenkuvaan. Olioperustaiselle ohjelmistolle on olemassa malleja, joilla voidaan päätellä haavoittuvimmat luokat. On esitetty epäilys siitä, että kompleksisuuteen perustuvat haavoittuvuuden arvioimismenetelmät sopisivat käytettäviksi vain iteratiivisen prosessin ensimmäisellä kierroksella. Olague ja kumppanit [2008] tutkivat useita monimutkaisuuden arviointiin kehitettyjä malleja iteratiivisessa olioperustaisessa prosessissa. He tulivat siihen johtopäätökseen, että mallit toimivat hyvin iteratiivisessa olioperustaisessa ohjelmistokehityksessä. Tutkimus osoitti myös sen, että pelkkä koodirivien lukumäärän laskenta menetelmänä arvioida virheiden määrää ei ole niin hyvä kuin kehittyneemmät menetelmät. [Olague et al., 2008]

Toisaalta Zhou ja kumppanit [2010] osoittivat koodirivien laskennan olevan hyvä menetelmä arvioida olioperustaisessa ohjelmistokehityksessä haavoittuvia luokkia. [Zhou et al., 2010] Ehkä tästä voidaan päätellä, että koon ja kompleksisuuden käyttäminen haavoittuvuuden arvioinnissa olioperusteisessa ohjelmistokehityksessä ei ole kovin yksiselitteinen asia. Ei voida siis osoittaa, että tietyllä metodilla saataisiin testauksen työmäärää varten erityisen tarkkaa arviota, vaan projektista riippuen useampi erilainen laskentamalli tuottaa melko oikeita tuloksia.

Olioperustaisuus kannustaa uudelleenkäyttöön, ja ohjelmoinnissa tätä on jo jonkin aikaa käytetty menestyksekkäästi. Kun oliot luodaan riittävän geneerisiksi, niitä voidaan käyttää muissakin sovelluksissa kuin vain siinä, mitä varten ne alun perin luotiin. Kuitenkin testaaminen on usein hoidettu alusta asti uudessa projektissa. Olioperustaisen arkkitehtuurin käyttäminen mahdollistaa vastaavaa uudelleenkäyttöä myös testaamisen suhteen, mutta siinä tulee huomioida joitakin erityistapauksia. Määrittelyt muuttuvat, eivätkä kaikki komponentit välttämättä toimi samalla tavalla uudessa ohjelmistossa kuin vanhassa. Samoin metodien parametrointia voidaan joutua muuttamaan, jolloin testitapauksiakin tulee muuttaa. Dallal ja Sorenson [2005] ovat tutkimuksessaan päätyneet siihen tulokseen, että uudelleenkäytettävillä testitapauksilla olioperustaisessa runkoarkkitehtuurissa päästään vähintään samaan virhekattavuuteen kuin ”round-trip path” -pohjaisella testitapausten luontitekniikalla. Round-trip path -tekniikalla tarkoitetaan sitä, että ensin arkkitehtuurin pohjalta luodaan suunnittelu ja toisaalta toteutuksessa

esiin tulleet muutokset päivitetään takaisin arkkitehtuurikuvaukseen. [Dallal and Sorenson, 2005]

Olioperustaisessa ohjelmistokehityksessä on myös ongelmakohtansa. Ohjelmoinnilla voidaan abstrahoida olioiden tietosisältöä ja estää muuttujien käyttö ulkopuolelta. Tällä on tietenkin hyötynsä ohjelmoinnin kannalta, mutta se asettaa testaukselle haasteita. Jos kaikkea ei saada yksikkötestattua kunnolla, on suuri riski, että ohjelmistoon jää virheitä, jotka jossain tilanteessa tulevat esiin. Kansomkeat ja Rivepiboon [2008] ovat kehittäneet metodin, jonka avulla tietovirtoja seuraamalla saadaan helpommin tietoa olion sisäisistä tiloista. Tämän metodin avulla olioperustaisen ohjelmistokehityksen testauskattavuus paranee. Toistaiseksi metodin käyttäminen vaatii vielä manuaalistakin työtä, mutta tutkijoilla on tarkoitus jatkaa tutkimuksensa parissa automatisoimalla metodia lisää. [Kansomkeat and Rivepiboon, 2008]

Edellä mainituista seikoista johtuen voidaan todeta olioperustaisen ohjelmistokehityksen antavan hyviä apuvälineitä testaamiseen. Testitapausten uudelleenkäyttämisellä säästetään niiden suunnitteluun kuluvaa aikaa, ja siten työmäärä pienenee. Vähemmän abstrakti mallinnustapa järjestelmästä auttaa vähemmän ohjelmointisuuntautunutta testaajaakin ymmärtämään ohjelmiston logiikan, ja se vaikuttaa työmäärää laskevasti. Toisaalta taas osa olioperustaisista menetelmistä voi aiheuttaa päänvaivaa testaamiseen, mutta työkaluja kehitetään koko ajan eteenpäin.

6. Automatisointi

On esitetty, että testauksen työmäärä olisi keskimäärin noin 30–50% toteutuksen työmäärästä. Silti ohjelmiston loppukäyttäjille valitettavan usein syntyy mielikuva, ettei ohjelmistoa olisi testattu lainkaan tai ei ainakaan riittävästi. On siis selvää, että testauksesta huolimatta ohjelmistoon jää usein virheitä, ja vasta loppukäyttäjät löytävät ne. [Kruchten, 2003]

Ohjelmistojen muuttuessa monimutkaisemmiksi ja aikataulujen kiristyessä laadunvarmistukseen on pitänyt kehittää uusia työtapoja tehokkuuden säilyttämiseksi. Erityisesti kriittisellä telekommunikaatioalalla testauksen kustannukset ovat saattaneet olla jopa 40–80 % ohjelmistoprojektin kokonaiskustannuksista. Toisaalta markkinat aikaansaavat aikataulupainetta, jolloin tuotteen pitäisi ehtiä markkinoille entistä nopeammin, laadun kuitenkaan heikentymättä. [Määttä et al., 2009]

Testausta voitaisiin teoriassa jatkaa ohjelmistoprojekteissa lähes ikuisesti. Tarkoitus olisi kuitenkin löytää se kompromissiratkaisu, jolla kriittiset virheet löydettäisiin mahdollisimman tehokkaasti suhteessa käytettyihin resursseihin. Eräs tapa tehostaa resurssien käyttämistä on testauksen automatisointi. Testauksen automatisoinnilla on saatu vähennettyä testaukseen kuluvia resursseja jopa 80 %. [Virkanen, 2002]

Testausautomaation käyttöönottaminen yleistyy koko ajan. Projektin testausta ei kuitenkaan voida automatisoida hetkessä, ja aina tulisi tapauskohtaisesti arvioida, kannattaako automatisointi. Toisaalta ohjelmistojen koko ajan kasvaessa automaatiolla voidaan parantaa testikattavuutta ja ohjata suunnittelumenetelmiä formaalimpaan suuntaan. [Harju ja Koskela, 2003]

Taipale ja kumppanit [2005] ovat huomioineet testauksen kasvavan tarpeen. He selvittivät tutkimuksessaan, mitä alueita testauksen tutkimuksessa kannattaisi tulevaisuudessa painottaa. Tutkimuksessa asiantuntijaryhmä koostui työssään testauksen kanssa toimivista ammattilaisista. Heidän mukaansa testauksen automatisointi ja siinä käytettävät työkalut olisivat tärkein tutkimuskohde. Tästä voidaan päätellä, että automatisoinnissa ja testaustyökaluissa olisi vielä paljon parannettavaa kaupallisten projektien testauksen näkökulmasta.

Testausautomaation käyttöönottoa harkitessa kannattaa huomioida useita seikkoja. Testausautomaatiot ovat usein projektikohtaisia: jos jokin projekti automatisoidaan, tehtyä automaatiota ei välttämättä voida hyödyntää muissa projekteissa. Testiautomaation rakentaminen vaatii kohtuullisen paljon resursseja. Automaatiolle vaaditaan ympäristö, jossa testejä pystytään automatisoidusti ajamaan. Tämän lisäksi jonkun tarvitsee suunnitella ja toteuttaa automatisoidut testitapaukset. Jos ohjelmistolle on tiedossa kohtuullinen elinkaari, eli sille tehdään paljon ylläpitotyötä vastaisuudessakin, on testiautomaatio tällaiseen projektiin järkevämpi valinta kuin pieneen ja kertaluontoiseen projektiin. [Harju ja Koskela, 2003]

Työkalut testauksen automatisointiin eivät useinkaan ole kovin helppokäyttöisiä ja testauksen automatisoijalta yleensä vaaditaan sekä tietämystä ohjelmoinnista että testaamisesta. Koska testiautomaatiotyökalut ovat hyvin sovellusratkaisukohtaisia, on todennäköistä, että automatisoijan tulee opetella uuden työkalun käyttötapaa ja soveltamista. Lisäksi tulee muistaa, että pelkkä suuri testimäärä ei tuo ohjelmistolle laatua, vaan testien pitää olla hyvin suunniteltuja, jotta niiden avulla voidaan todella saada paikannettua ohjelmiston virheet. [Harju ja Koskela, 2003]

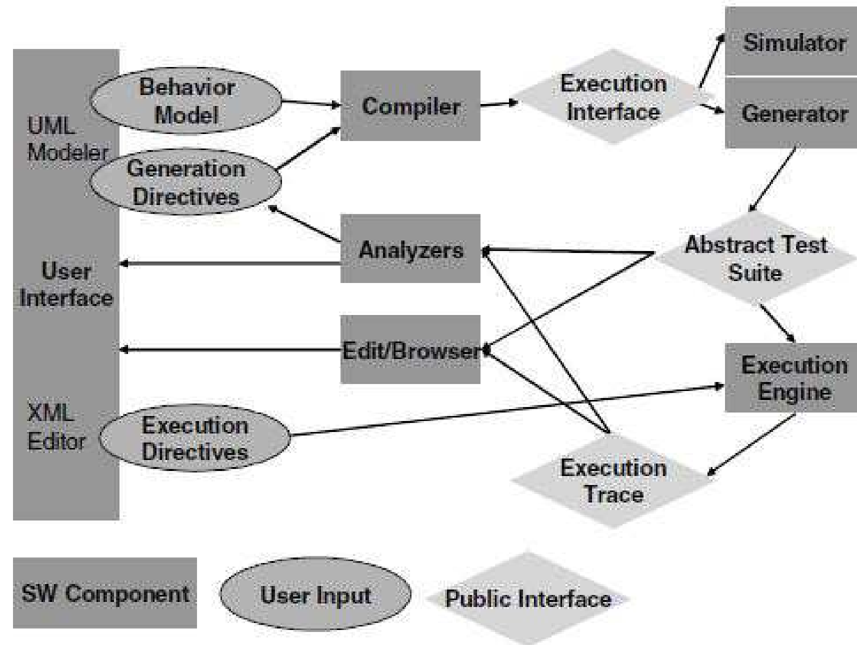
Testausta voidaan automatisoida monella eri tavalla monessa eri vaiheessa projektin testaamista. Siihen on mahdollista käyttää spesifejä työkaluja tai vain perinteisiä skriptejä. Automaation avulla voidaan esimerkiksi suorittaa yksikkötestausta kooditasolla, testata käyttöliittymää tai tehdä kuormitustestausta. Tarkoitus kuitenkin on aina vähentää testaajan suorittamaa manuaalista työtä tai aikaansaada tilanteita, joihin pelkkä ihminen ei testaajana kykene. Tällöin testaajalle jää enemmän aikaa perehtyä niihin osa-alueisiin, joita ei voida automatisoida. Toisaalta yrityksillä on epärealistisia mielikuvia testauksen automatisoinnista. Helposti oletetaan, että kone hoitaa testaajan työt nopeasti ja edullisesti. Samoin tulee muistaa, että testauksen automatisointi itsessään on jo ohjelmistoprojekti ja sen laatu pitää varmistaa samalla tavalla kuin varsinaisen testattavankin projektin. Jos automatisoitu testitapaus on virheellinen, se tuottaa virheellisen tuloksen. [Honkela, 2005]

6.1. Testitapausten luominen automaattisesti

On kehitetty työkaluja, joiden avulla suoraan ohjelmistodokumenteista voidaan jäsentää testitapauksia. Testitapausten luomiseen automaattisesti on monia erilaisia menetelmiä, jos määrittelyt luodaan systemaattisesti käyttäen tiettyä merkintätapaa.

Eräs työkalu mallipohjaiseen testitapausten automaattiseen muodostamiseen on AGEDIS (engl. Automated Generation and Execution of test suites for DIstributed component-based Software). Se kehitettiin Euroopan Unionin tukemassa yhteistyöprojektissa ja on saatavilla ilmaiseksi tutkimuskäyttöön ja maksullisesti kaupalliseen käyttöön. AGEDIS tukee UML-pohjaista suunnittelutapaa ja sisältää apuvälineitä mallinnukseen, testitapausten suunnitteluun ja testitapausta suorittamiseen. [Hartman and Nagin, 2005]

Kuvassa 10 on esitetty AGEDIS-kehitysympäristön arkkitehtuuri. Käyttäjän tulee antaa syötteinä järjestelmälle käyttötapakuvaus ja testausarkkitehtuurin vaatimukset UML-muodossa sekä testigeneroinnin vaatimukset XML-muodossa. Käyttötapauskuvaus voidaan antaa esimerkiksi luokkakaavioina. AGEDIS-kehitysympäristön mukana tulee myös muita testausta tukevia komponentteja. [Hartman and Nagin, 2005]



Kuva 10. AGEDIS-kehitysympäristön arkkitehtuuri: suorakulmiot ovat kehitysympäristön komponentteja, ovaalit ovat käyttäjän syötteitä, vinoneliöt ovat julkisia rajapintoja. [Hartman and Nagin, 2005]

AGEDIS-kehitysympäristön testitapausten luontikomponenttina on käytössä testienluontityökalu TGV. Se on ollut käytössä monissa tutkimuksissa, joissa on tutkittu erilaisten sovellusten määrittelyn muuntamista testitapauksiksi usealla eri merkintätavalla. TGV tarvitsee syötteenä määrittelyn ja tason, jolle testitapaukset luodaan (yksikkötestaus, systeemitestaus, jne). TGV:n vahvuus on sen tuki useille eri merkintätavoille. Koska TGV:n kehitysrajapinta on avoin, voi kuka tahansa halutessaan laajentaa sitä omiin tarkoituksiinsa sopivaksi. [Belifante et al., 2005]

Mallipohjaiset testitapausgeneraattorit tuottavat suuren määrän testitapauksia, mistä voi seurata ongelmia, mikäli testaus on kallista tai resurssit ovat rajalliset. Fraser ja kumppanit [2009] tutkivat, miten testitapausten luomista voitaisiin optimoida. He kehittivät menetelmän, joka testitapausten luomisen yhteydessä tarkkailee jo muodostettuja testitapauksia ja jos uusi testitapaus sisältää vanhan testitapauksen, vanha testitapaus poistetaan. Tällä tavalla testitapausten määrä saadaan pysymään kohtuullisena eikä muodostu turhia päällekkäisiä testitapauksia. Tutkimuksessa kokeiltiin myös testitapausten päämäärän vaikutusta testitapausten määrään ja sen huomattiin olevan merkityksetön tekijä testitapausten määrään. Kaikki tutkimuksessa käytetyt testauspäämäärät pienensivät testitapausten joukkoa, mutta merkittävää eroa eri päämäärien välillä eri saatu aikaan. [Fraser et al., 2009]

Olioperustaisen ohjelmistokehityksen sekvenssikaavioista on myös mahdollista luoda priorisoiden systeemitestaukseen soveltuvia testitapauksia, jos sekvenssikaavioissa on käytetty UML 2.0 -notaatiota. Kun testitapauksia priorisoidaan kaavioista saadun tiedon perusteella, testitapausten määrä ei kasva liian suureksi. Kundun ja kumppanei-

den [2009] esittämä malli paitsi priorisoi, myös generoi automaattisesti suositeltavia testattavia arvoja. Menetelmä priorisoi ohjelmiston tärkeät ja kriittiset alueet ja painottaa testitapauksia niille. Vähemmät tärkeät alueet voidaan jättää vähemmälle huomiolle. Kuten edellä mainittiin, automaattisesti luotujen testitapausten määrä voi nousta merkittävästi, joten priorisointi helpottaa tätä ongelmaa. [Kundu et al., 2009]

Testitapausten luomisessa voidaan myös yhdistää erilaisia testaustapoja. Eräs tapa voi tulevaisuudessa olla Liun ja Chen [2008] esittämä metodi, joka yhdistää toiminnallisten ja strukturaalisten testitapausten automaattisen luomisen. Relaatioperustainen metodi tarkoittaa relaatioiden määrittelyä erityyppisten muuttujien välille etukäteen. Näiden relaatioiden pohjalta muodostetaan automaattisesti testitapauksia. Toistaiseksi tälle metodille ei ole olemassa kunnollista työkalutoteutusta, joten sitä ei voi vielä kunnolla hyödyntää. [Liu and Chen, 2008]

Testitapauksia voidaan luoda myös käyttöliittymätestaukseen. On olemassa hierarkiaan perustuva malli, joka avustaa testisuunnittelijaa käyttöliittymätestitapausten luomisessa. Toistaiseksi tässä mallissa on vielä paljon vastuuta testisuunnittelijalla, eikä mallin käyttäminen edesauta huolehtimista esimerkiksi testikattavuudesta. Inhimillisen virheen mahdollisuus on siis käytännössä sama kuin manuaalisessa testitapausten luomisessa. Kuitenkin työmäärää saattaa hieman vähentää se, että automatisointi hoitaa siitä osan. [Memon et al., 2001]

Lutsky [2000] on esittänyt metodin, jolla voidaan muodostaa testitapaukset suoraan luonnollista kieltä olevasta dokumentista. Tämä tosin vaatii, että määrittelyissä käytetään yhtenäistä merkintätapaa ja tiukasti standardoitua muotoa. Tällöin katselmoinneissa tulee huomioida esitystapa entistä tarkemmin. Mahdollista on jopa purkaa sanalliset määrittelyt suoraan automaattisesti suoritettaviksi testitapauksiksi ilman, että testaajan tarvitsee puuttua asiaa. Menetelmä hyödyntää luonnollisen kielen tunnistamista [Lutsky, 2000]

Koska testauksen automatisointia käytetään yrityksissä vähän, niin Polo ja kumppanit [2007] tutkivat ohjelmistokehitysprosessien ja työkalujen integrointia automatisoitua testausta varten. Heidän empiirinen kokeensa osoitti, että automatisoitua työkalua apuna käyttänyt ryhmä pääsi parempiin tuloksiin kuin ryhmä, joka kirjoitti testitapaukset manuaalisesti. [Polo et al., 2007]

Lucio ja Samer [2005] esittävät väitteen, että yleensä testitapausten joukosta valitaan testattaviksi mielenkiintoisimmat, muttei välttämättä kriittisimpiä. Tätä perustellaan sillä, että mahdollisia testitapauksia on niin suuri määrä, ettei kaikkia ole mahdollista ehtiä testata. Tämän ongelman testitapausten automaattinen luominen poistaisi, sillä se on puolueeton tapa muodostaa ohjelmiston määrittelyistä testitapaukset. Oman kokemukseni perusteella olen väitteen kanssa eri mieltä, sillä useimmiten testaajilta löytyy kunnianhimoa, ja siten testitapaukset yleensä valitaan niiden kriittisyyden perusteella.

Automatisoidut testitapaukset tarvitsevat syötteitä, joilla testitapauksia ajetaan. Testauksen työkuormaa voidaan vähentää automatisoidulla testidatan generoinnilla. Metodeja, joilla työkalut päättelevät arvoja mahdollisille syötteille, on monia. Toistaiseksi nämä metodit ovat painottuneet lähinnä olioperustaiseen arkkitehtuuriin tukemiseen. [Chung and Bieman, 2009]

Useat työkalut osaavat luoda näitä syötteitä samalla kun generoivat testitapauksia. Työkalu joko muodostaa oman testitapauksen jokaiselle mahdolliselle syötteelle tai luo testitapausta kohden taulukon, jossa on listattu mahdolliset syötteet. Testitapausten hallinnoinnin kannalta on helpompaa, jos mahdollisista syötteistä muodostetaan taulukko, sillä tällöin testaaja voi lisätä ja poistaa syötteitä harkintansa mukaan.

Testidatan generoinnilla tarkoitetaan sitä, että testitapaustyökalu osaa päätellä, millä arvoilla testitapauksia tulisi yrittää ajaa ja mitä tuloksia näillä arvoilla pitäisi saada aikaan. Esimerkiksi oliolla voi olla metodi, joka laskee yhteen parametrina saamansa kaksi arvoa. Testidatana metodille voidaan antaa arvot 1 ja 2, sekä voidaan päätellä, että tulokseksi pitäisi tulla 3. Jos tulos on jotakin muuta kuin 3, metodissa on todennäköisesti jotakin vikaa. Kaikkia mahdollisia arvoja ei ole käytännössä mahdollista testata, joten testidataan pitäisi pystyä valitsemaan ne arvot, joilla ohjelmisto todennäköisimmin toimisi väärin ja ne, jotka ovat käytön kannalta oleellisimpia. Käytännössä tämä usein tarkoittaa raja-arvotestausta, eli testataan arvoja sen skaalan rajoilta, jossa metodin tulisi toimia. Käytännössä testitapaus ja sen data ovat harvoin niin yksinkertaisia kuin edellä mainitussa esimerkissä. [Arcuri and Yao, 2008]

Testidataa on huomattavasti helpompi generoida numeerisille arvoille. Kuitenkin käytännössä testidataa pitäisi olla myös merkkijonoille. Merkkijonot testidatana vaativat työkalulta enemmän, eikä niille ole vielä yhtä laajaa ja kattavaa tukea kuin numeraaliselle testidatalle. Merkkijonoja pitäisi voida verrata ja järjestää. Työkalujen tulisi myös voida ymmärtää säännöllisten lausekkeiden (engl. regular expression) olemassaolo. [Alshraideh and Bottaci, 2006]

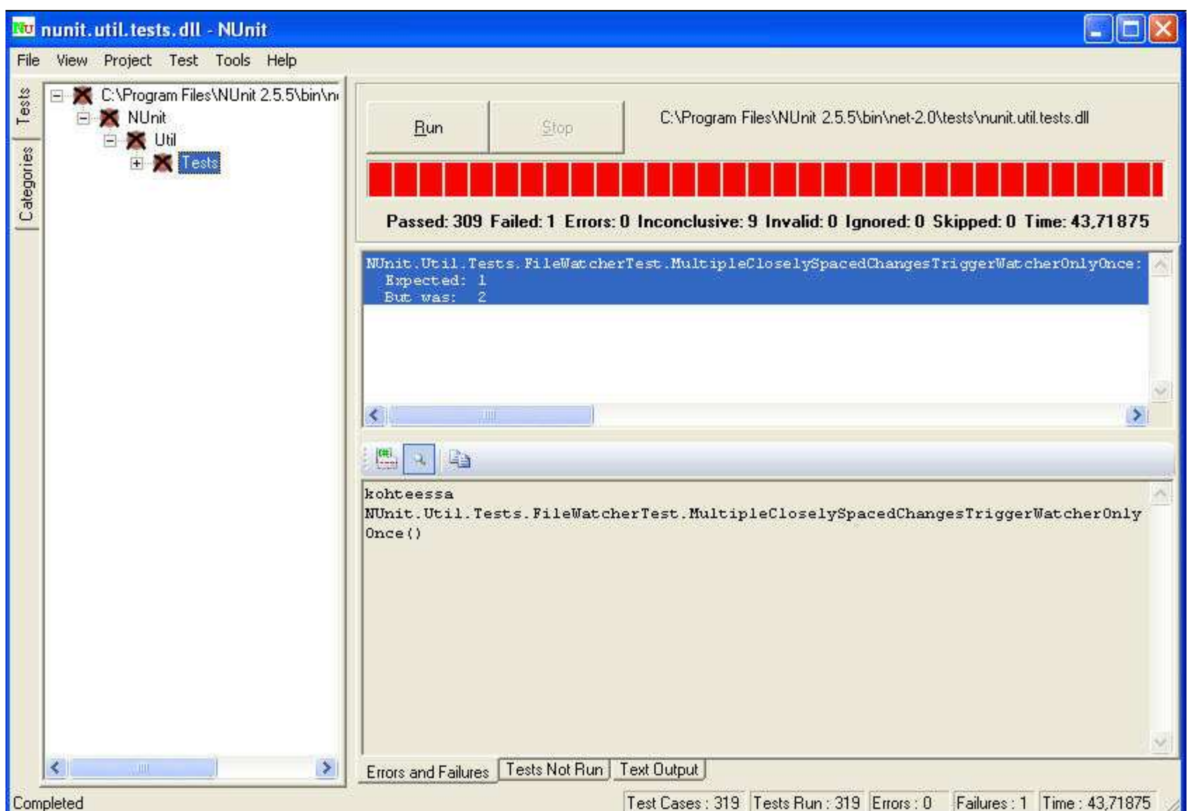
Jos työkalu toimii luotettavasti, niin testitapausten automaattisella luomisella voidaan välttyä siltä virheeltä, että jokin alue ohjelmistosta jää huomioimatta. Automatisoidun testitapausten luomisen pitäisi vaikuttaa myös siten, että dokumentoinnin laatuun kiinnitetään enemmän huomiota. Testaajan työmäärä vähenee, kun ainakin perusrunko testitapauksiin saadaan valmiiksi. Toisaalta jos testitapausten automaattinen generointi muodostaa erittäin suuren joukon testitapauksia, pitää ihmisen kuitenkin suorittaa karsinta realistiselle tasolle.

6.2. Automatisoitu yksikkötestaus

Automatisoitu yksikkötestaus sisältää useita vaiheita. Ensimmäinen vaihe on testattavien ohjelmiston osien luominen. Käytännössä olioperustaisessa kehityksessä tämä tarkoittaa olioiden luomista. Seuraavana vuorossa on testikohteiden valinta, sillä aina ei ole taloudellisesti eikä ajallisesti mahdollista testata koko ohjelmistoa kattavasti. Kuten

jo edellä esitettiin, tulee valita järkevä osajoukko kaikista mahdollisista syötteistä. Testikohteiden valinnan jälkeen on vuorossa testien ajaminen. On hyvin suositeltavaa, että tämä voidaan hoitaa automaattisesti öisin tai esimerkiksi jatkuvan kehityksen (engl. continuous integration) ympäristössä. Seuraavana on vuorossa testitapausten hyväksymisen määrittely, eli millä arvoilla testitapaus katsotaan läpäistyksi. Olennaiset testien ajamiseen liittyvät asiat tulee kirjata, erityisesti ei-läpäisseiden testitapausten tulokset. [Meyer et al., 2007]

Hyvin yleisesti käytössä oleva yksikkötestaustyökalu on NUnit. NUnitia voi käyttää .NET -tekniikalla toteutettavissa projekteissa, ja testitapaukset voi ohjelmoida millä tahansa .NET -perheen kielistä. NUnitin käyttäminen on hyvin samankaltaista kuin varsinainen ohjelmointi Visual Studio -ympäristössä. NUnit tarjoaa myös mahdollisuuden kääntää ja ajaa testit ajastetusti esimerkiksi öisin. [Makkonen, 2008]



Kuva 11. Kuva NUnitin käyttöliittymästä ohjelman omalla kirjastolla ajettujen testien jälkeen.

Kuten kuvasta 11 näkyy, NUnit antaa selkeästi palautetta siitä, miten testien ajaminen onnistui. Tällaisen työkalun käyttäminen yksikkötestauksessa automaattisin yöajoin edesauttaa sitä, että koodimuutosten mahdolliset virheet löytyvät aikaisessa vaiheessa.

Toinen esimerkki käytössä olevista automaattisista työkaluista on AutoTest. AutoTest sekä muodostaa testitapauksia että osaa päätellä, mikä olisi testitapauksille oikea lopputulos. Jos testitapausta ei läpäistä onnistuneesti, AutoTest lisää testitapauksen automaattisesti regressiotesteihin.

Meyer ja kumppanit [2007] tutkivat, miten AutoTest-työkalu löytää virheitä käytössä olevista ohjelmistokirjastoista. Heidän tutkimuksessaan oli mukana osia esimerkiksi EiffelBase- ja Gobo-kirjastoista. Kaikista tutkimuksessa testatuista ohjelmakirjastoista löytyi virheitä. Ei-läpäistyjen testitapausten suhde läpäistyihin testitapauksiin vaihteli välillä 1-50 %. Löytyneitä virheitä oli lukumäärällisesti yllättävän paljon. On kuitenkin oleellista, että virheistä yksikään ei vaikuta olevan ns. väärä, eli jokainen virhe on todellinen. Tosin virheiden taso vaihtelee, eivätkä kaikki löydetyt virheet ole ohjelman toiminnallisuuden kannalta kriittisiä. Mielenkiintoisena yksityiskohtana, merkittävä osa virheistä johtui väärin käytetystä void-tyypityksistä. Sinänsä ei asiana kriittinen, mutta hyvä esimerkki siitä, miten automaattisen yksikkötestaustyökalun käyttäminen voi auttaa estämään ajonaikaisia ongelmia. [Meyer et al., 2007]

Tietokantaproseduureille voidaan suorittaa yksikkötestausta aiemmin esitellyn NUtin avulla, mutta jos ei haluta ohjelmointiosaamista vaativaa testityökalua, voi työkalu olla SQLUnit. Se on Java-pohjainen työkalu, joka käyttää JDBC-yhteyttä tietokantaan. Testitapaukset määritellään XML-muodossa, eli varsinaista ohjelmointiosaamista ei vaadita. Kuvassa 12 on esimerkki siitä, miten SQLUnitilla muodostetaan testitapauksia. [Makkonen, 2008]

```
<?xml version="1.0"?>
<!DOCTYPE sqlunit SYSTEM "file:sqlunit/lib/sqlunit.dtd" [
  <!ENTITY connection SYSTEM "file:sqlunitConnectionConfig.xml">
  <!ENTITY data SYSTEM "file:sqlunitTestData.xml">
]>

<sqlunit>

  &connection;

  <setup>
   &data;
  </setup>

  <test name="Looking up New Member created from Register page">
   <sql>
    <stmt>select Firstname,Surname from Client where UserID=?</stmt>
    <param id="1" type="VARCHAR" inout="in">${test.newuser.id}</param>
   </sql>
   <result>
    <resultset id="1">
       <row id="1">
         <col id="1" type="VARCHAR">Dummy</col>
         <col id="2" type="VARCHAR">User</col>
       </row>
    </resultset>
   </result>
  </test>

  <teardown>
   <sql>
    <stmt>delete from Client where UserID=?</stmt>
    <param id="1" type="INTEGER" inout="in">${test.newuser.id}</param>
   </sql>
  </teardown>

</sqlunit>
```

Kuva 12. Esimerkkikoodia SQLUnitin käytöstä tietokannan yksikkötestauksessa. [Makkonen, 2008]

Oman kokemukseni perusteella etenkin ylläpitoprojekteissa on tärkeää, että kriittiset alueet ohjelmistosta on katettu automaattisella yksikkötestauksella. Vaikka varsinainen koodimuutos ei liittyisi ohjelmiston kriittiseen osaan, virhe voi konkretisoitua siellä. Työmäärän kohtuullistamisen vuoksi ylläpitovaiheessa olevia ohjelmia usein testataan vain muutosten osalta. Tämän vuoksi automatisoitu yksikkötestaus on hyvä keino varmistaa, etteivät muutokset ole rikkoneet mitään kriittistä toiminnallisuutta. Toisaalta, kun automatisoidulla yksikkötestauksella löydetään yksinkertaisimmat virheet pois, jää varsinaiselle testaajalle aikaa keskittyä monimutkaisempiin asioihin. Automatisoitu yksikkötestaus on ehdottoman tehokas keino pienentää testausprojektien työmäärää.

6.3. Automatisoitu toiminnallinen testaus

Toiminnallinen testaus on usein pääasiallinen testausmuoto ohjelmistoprojekteissa ja muiden testausmuotojen merkitystä saatetaan vähätellä. Toiminnallinen testaus perustuu vaatimuksiin, joiden pohjalta on luotu testitapaukset siitä, miten ohjelman tulisi toimia. [Woodward and Hennell, 2005]

Testaajien näkökulmasta toiminnallinen testaus on raskas ja hidas testausmuoto. Siinä tulisi varmistaa kaikkien ohjelman toimintojen toimiminen siten kuin on haluttu. Samalla tulee varmistaa, ettei ohjelma sisällä ei-haluttuja ominaisuuksia. [Baresi and Bezze, 2006]

Toiminnallinen testaus on hitautensa vuoksi loistava kohde automatisoinnille. Toisaalta taas juuri toiminnallinen testaus tapahtuu käyttöliittymää käyttäen, ja siitä saattaa seurata testityökaluille odottamattomia ongelmia, kun vuorovaikutuksessa on mukana satoja eri komponentteja. [Harju ja Koskela, 2003]

Pelkällä toiminnallisella testauksella kuitenkin saattaa jäädä huomaamatta kriittisiäkin virheitä. Woodward ja Hennell [2005] ovat esittäneet, että strukturaalista testikattavuutta tulisi seurata automatisoidusti toiminnallisen testauksen aikana. Jos kattavuudessa huomataan aukkoja, tulee testitapausten määrää lisätä kattavuuden parantamiseksi. Testauksen työmäärän kannalta saavutetaan ajansäästöä yhdistämällä kaksi eri testaustyyliä. [Woodward and Hennell, 2005]

Yleisin tapa automatisoida toiminnallista testausta on käyttää nauhoitus/toistotyökaluja. Näiden työkalujen perusidea on siinä, että käyttäjä suorittaa testattavassa ohjelmistossa toimenpiteitä, jotka työkaluja nauhoittaa konekieliseen muotoon. Näitä nauhoitettuja testejä voidaan toistaa rajattomasti ks. taulukko 8. Useimmat työkalut myös tarjoavat mahdollisuuden ajaa näitä testejä ajastetusti, ilman että testaajan täytyy olla mukana prosessissa. [Everett and McLeod, 2007]

Vaihe 1	Testaaja suorittaa testattavassa ohjelmistossa jonkin toiminnon, siten kun se kuuluisi suorittaa.
Vaihe 2	Testityökalu nauhoittaa testaajan toiminnot konekieliseen muotoon, esimerkiksi jollekin ohjelmointikielelle tai skriptiksi.
Vaihe 3	Testityökalu toistaa nauhoittamansa toiminnot, kuten testaaja aiemmin, mutta tämä vaihe ei vaadi testaajalta panosta.

Taulukko 8. Vaiheet, joilla luodaan automaattisia testitapauksia toiminnallista testausta varten nauhoitus/toisto-työkalulla [Everett and McLeod, 2007]

WatiN Test Recorder on yksi testityökalu, jolla on mahdollista nauhoittaa toiminnallisia testitapauksia selainpohjaisista sovelluksista. Siinä on mahdollista valita, millä selaimella haluaa testejä ajaa. Jos testattava ohjelmisto on web-sovellus, joka tukee useita selaimia, on työmäärän kannalta merkittävää, jos testaaja voi suorittaa testit vain yhdellä selaimella ja työkalu toistaa testit muissa tuettavissa selaimissa. WatiN muodostaa testaajan toimenpiteistä XML-muotoisen tiedoston ja sillä voi toistaa testiä tarvittavan monta kertaa. Kuvassa 13 on esimerkki WatiNin muodostamasta testitapauksesta. Siinä navigoidaan sivulle, kirjoitetaan tekstikenttään tekstiä ja painetaan painiketta. Vaikka esimerkki on hyvin yksinkertainen, siitä saa käsityksen, miten monipuolisesti tällä työkalulla kanssa pystyy web-pohjaisia sovelluksia testaamaan.

```
<?xml version="1.0" encoding="utf-8"?>
<WatinTests>
  <PageList>
    <WebPage>
      <FriendlyName>blank</FriendlyName>
      <URL>about:blank</URL>
      <HashCode>28633211</HashCode>
      <LoadDate>129184870393593750</LoadDate>
      <ParentPageHash>0</ParentPageHash>
    </WebPage>
    <WebPage>
      <FriendlyName>~kk63284testihtml</FriendlyName>
      <URL>http://www.uta.fi/~kk63284/testi.html</URL>
      <HashCode>102253570</HashCode>
      <LoadDate>129184876852343750</LoadDate>
      <ParentPageHash>0</ParentPageHash>
    </WebPage>
  </PageList>
  <Test Name="test">
    <Action ActionType="Navigate" Username="" Password="" PageHash="1103484735">http://www.uta.fi/~kk63284/testi.html</Action>
    <Action ActionType="BrowserClick" NoWait="0" PageHash="1103484735">
      <ElementType>TextField</ElementType>
      <FinderSet>
        <Finder FindMethod="Name" FindName="" FindValue="S1" Regex="0" />
      </FinderSet>
    </Action>
    <Action ActionType="TypeText" PageHash="102253570" TextToType="testi" ValueOnly="1" Overwrite="1">
      <ElementType>TextField</ElementType>
      <FinderSet>
        <Finder FindMethod="Name" FindName="" FindValue="S1" Regex="0" />
      </FinderSet>
    </Action>
    <Action ActionType="BrowserClick" NoWait="0" PageHash="102253570">
      <ElementType>Para</ElementType>
      <FinderSet>
        <Finder FindMethod="Text" FindName="" FindValue="" Regex="0" />
        <Finder FindMethod="Style" FindName="" FindValue="" Regex="0" />
        <Finder FindMethod="Value" FindName="" FindValue="" Regex="0" />
        <Finder FindMethod="Id" FindName="" FindValue="" Regex="0" />
        <Finder FindMethod="Src" FindName="" FindValue="" Regex="0" />
        <Finder FindMethod="Url" FindName="" FindValue="" Regex="0" />
      </FinderSet>
    </Action>
    <Action ActionType="BrowserClick" NoWait="0" PageHash="102253570">
      <ElementType>Button</ElementType>
      <FinderSet>
        <Finder FindMethod="Id" FindName="" FindValue="Button1" Regex="0" />
      </FinderSet>
    </Action>
  </Test>
</WatinTests>
```

Kuva 13. WatiN-työkalun muodostamaa XML-koodia selainpohjaisesta toiminnallisesta testauksesta

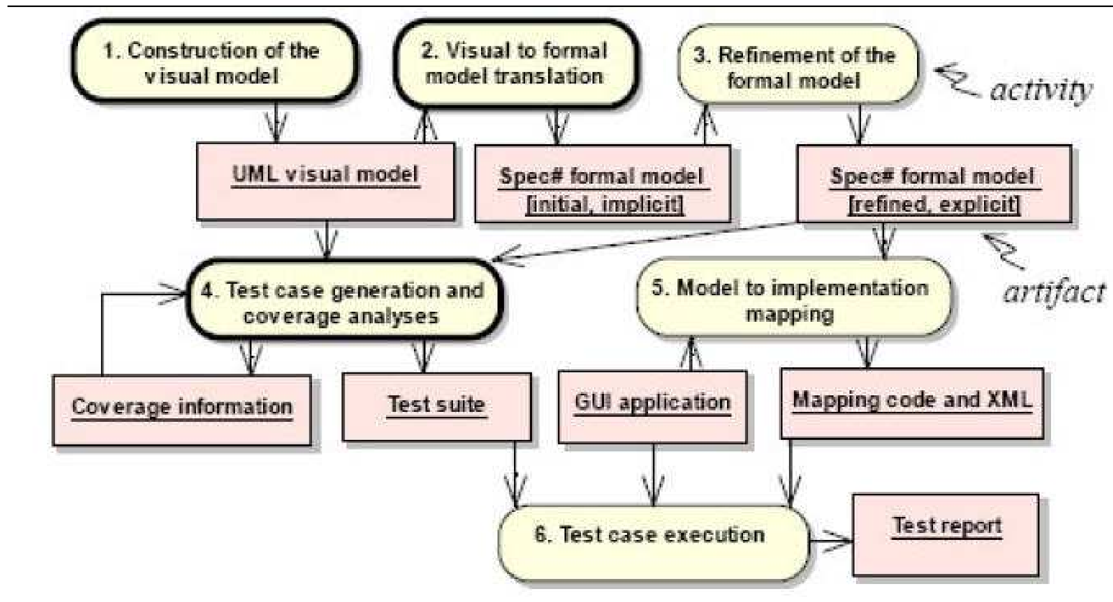
Toisaalta Li ja Wu [2005] esittävät, että nauhoitus/toisto-menetelmä ei olisi automatisoitua testausta. He myös nostavat esille joitakin ongelmia työkalujen käyttämisessä. Usein testaaja voi joutua manuaalisesti muokkaamaan työkalujen luomia skriptejä, koska ne eivät ole välttämättä tallentuneet oikein. Työkalujen toiminta mahdollisissa virhetilanteissa ei myöskään ole aina vakaata. [Li and Wu, 2005]

On tietenkin totta, että testit pitää ensin suorittaa manuaalisesti, jotta ne voidaan suorittaa uudelleen automaattisesti. Olen kuitenkin sitä mieltä, että normaalisti testejä joutuu toistamaan projektin aikana niin usein, että nauhoittamallakin saadaan työmäärällisesti hyötyä. Jos skriptit eivät tallennu kerrasta oikein, työmäärää kasvaa, kun testaaja joutuu tavallaan etsimään virhettä testityökalusta ja sen jälkeen kiertämään ongelman manuaalisesti. Kuitenkin kokonaisuutena testien nauhoittaminen tuo lisäarvoa testaukselle. Niitä voidaan käyttää apuna etenkin jatkuvissa ylläpitoprojekteissa, joissa kaikkea ei ehditä aina testaamaan uudelleen.

Mosley ja Posey [2002] esittävät arvion, että jos jonkin tietyn testisetin suorittaminen manuaalisesti vie 6 tuntia, niin sen muuntaminen automatisoiduksi vie ainakin 18 tuntia. Tämä aika koostuu niistä kuudesta tunnista, jotka kuluvat testien nauhoittamiseen suorittamalla ne. Sitten nauhoittamiseen kulunut aika tuplataan, niin saadaan se aika, joka menee todennäköisesti testiskriptien muokkaamiseen. Mosleyn ja Poseyn kokemuksen perusteella keskimäärin noin 60 % ohjelmiston toiminnallisesta testauksesta on mahdollista automatisoida. Automatisointi tulisi heidän mukaansa aloittaa aina ohjelman kriittisimmistä ja yleisimmistä toiminnoista ja lisätä automatisointia sitten vähän kerrallaan, kunnes se kattaa ohjelmiston niin laajalti kuin mahdollista.

Frezza ja kumppanit [1996] kehittivät prosessin toiminnalliseen testaukseen, jossa käytetään hyödyksi vaatimusmäärittelyä ja suunnitteludokumentteja. He yhdistivät vaatimusmäärittelyn suunnitteludokumenttiin semanttisilla kaavioilla, ja näin saivat hyödynnettyä toiminnalliseen testaukseen esimerkiksi mahdolliset syöte- ja paluuarvot.

On myös ehdotettu prosessia, jolla voitaisiin mallipohjaisesti aikaansaada käyttöliittymän testaamiseen tarvittavia testitapauksia. Prosessi myös huolehtii testauksen kattavuudesta. Prosessi on pääpiirteissään esitetty kuvassa 14 ja prosessin vaiheita on selvennetty taulukossa 9. [Paiva et al., 2007]



Kuva 14. Yksinkertaistettu kuva prosessista, jossa mallipohjaisen käyttöliittymän perusteella muodostetaan käyttöliittymän testitapaukset. [Paiva et al., 2007]

Vaihe 1	Visuaalisen mallin muodostaminen UML-kaavioiden avulla
Vaihe 2	Visuaalisen mallin muuttaminen formaaliksi sääntöjen avulla
Vaihe 3	Formaalin mallin hienosäätö
Vaihe 4	Testitapausten generointi ja testikattavuuden arviointi
Vaihe 5	Testaaja voi yhdistää mallin konkreettiseen testattavaan ohjelmistoon
Vaihe 6	Testit suoritetaan automaattisesti ja virheet raportoidaan

Taulukko 9. Prosessin vaiheiden selitys [Paiva et al., 2007]

Edellä kuvattu prosessi automatisoi käyttöliittymän kautta suoritettavaa testausta merkittävästi. Jotta prosessia voitaisiin hyödyntää, tulee projektiin liittyvien dokumenttien olla tehty huolella, ja jos niihin tulee merkittäviä muutoksia, pitää prosessi suorittaa alusta alkaen uudelleen.

Nykyisin monen ohjelmiston käyttöliittymä on web-pohjainen ja data tallennetaan tietokantaan. Näiden ohjelmistojen toiminnallisen testauksen automatisointi vaatii koko ketjun suorittamista, eli datan syöttämistä web-käyttöliittymän kautta tietokantaan. Verkkosovellusten käyttöliittymäpään testaamiseen on olemassa joitain työkaluja, kuten jo aiemmin mainitut nauhoitus/toisto-työkalut. Näiden lisäksi html-koodia voidaan validoida ja sivujen suorituskykyä mitata. Nämä työkalut kuitenkin tukevat lähinnä staattista testaamista. [Ran et al., 2009]

Dynaamisen tietokantaa käyttävän verkkosovelluksen testaamisen vaikeus tulee siitä, että tietokanta on alati muuttuvassa tilassa. Jotta voit esimerkiksi ostaa tuotteen verkkokaupasta, sitä pitää olla varastossa. Jotta voit perua tilauksen, tilaus pitää olla ensin tehty. Tällaiseen testaamiseen ei ole ollut saatavilla riittävän monipuolisia automatisoivia testityökaluja. Ratkaisuna tähän Ran ja kumppanit [2009] esittelevät Au-

toDBT-työkalun (Automatic Database Tester). AutoDBT:llä on kolme päätoiminnallisuutta. Ensimmäinen on tilakone, joka kertoo sovelluksen oletetun toiminnan. Informaation tähän se kerää tietokannan transaktioista. Seuraavaksi generoidaan automaattisesti testisekvenssejä tilakoneelta saatavan tiedon perusteella ja luodaan dynaaminen datamääritys. Kolmantena suoritetaan testiprosessi käyttäen aiemmin luotuja sekvenssejä ja datamäärityksiä hyväksi. Kolmannesta vaiheesta muodostetaan raportti. Työkalua testattiin pienikokoisissa tietokannoissa oikeilla sovelluksilla ja tuloksissa havaittiin esimerkiksi se, että suurin osa transaktioista on vain lukuja. Tutkimusryhmän seuraava tarkoitus on selvittää tämän työkalun suorituskykyä, ja se oikeastaan ratkaisee, onko AutoDBT:stä käytännössä apua kaupallisten sovellusten testaamiseen.

Myös Di Lucca ja Fasolino [2006] ovat havainneet selkeät puutteet verkkosovellusten testaustyökaluissa. Sen lisäksi, että hekin kaipaavat dynaamista testausta tukevia työkaluja, heidän mielestään automatisoituun testitapausten luomiseen ei ole vielä riittävän laadukasta tukea. Myös prosesseja ja näkökulmia tulisi kehittää erityisesti verkkosovellusten näkökulmasta, koska niiden määrä on kasvamassa. Tulevaisuudessa verkkosovellusten teknologiakenttä todennäköisesti laajenee entisestään, joten työkalutukea pitäisi olla saatavilla uusille teknologioille nopeammin.

Käyttöliittymien testauksen automatisointi pohjautuu aivan liikaa staattisiin malleihin. Useinkaan mallit eivät huomioi sitä, että käyttöliittymän rakenne usein muuttuu dynaamisesti. Esimerkiksi, kun jokin komponentti käyttöliittymästä saavuttaa tietyn tilan, se voi mahdollistaa jonkin uuden näkymän avaamisen tai vastaavasti jokin muutos voi muuttaa toisen käyttöliittymäosan toimintaa. Yuan ja Memon [2010] kehittivät algoritmin, joka nykyisistä työkaluista poiketen osaa löytää käyttöliittymästä ajonaikaisia ominaisuuksia ja luoda niistä testitapauksia. He testasivat kehittämänsä algoritmin toimintaa joihinkin vapaasti jaettaviin graafisiin ohjelmistoihin ja olivat tyytyväisiä algoritmin toimintaan. Dynaamisen käyttöliittymän testauksen automatisointimahdollisuus voi tulevaisuudessa vaikuttaa suuresti testauksen työmäärään.

6.4. Automatisoitu suorituskyky- ja kuormitustestaus

Jos suorituskyky on ohjelmiston kannalta merkittävä tekijä, se tulee testata. Jos ohjelmalla todennäköisesti on enemmän kuin yksi yhtäaikainen käyttäjä, tulee suorittaa kuormitustestaus. Suorituskyvyllä tarkoitetaan ohjelmiston vasteaikoja yms. suoriutumista erilaisissa tilanteissa. Kuormitustestauksella tarkoitetaan useita samanaikaisia käyttäjiä tekemässä asioita joita ohjelmistolla on tarkoitus tehdä, simuloiden oikeaa tilannetta. On tietenkin mahdotonta, että yksi testaaja yrittää toiminnallaan vastata satoja käyttäjiä. Suorituskykyyn liittyvissä testauksissa automaatio on yleensä ainoa tapa saada simuloitua oikeankaltaisia tilanteita. [Mosley, 2000]

Kun testataan kuormitusta, tulee testaajalla olla käsitys siitä, kuinka monta yhtäaikaista käyttäjää ohjelmistolla voi samanaikaisesti olla. Pitää myös tietää, miten käyttäjät yleensä ohjelmistoa käyttävät. Jos esimerkiksi jotkut tietyt raskaat haut ovat yleisiä,

pitää testata ohjelmiston suorituskykyä useiden samanaikaisten raskaiden hakujen aikana. Testauksessa tulisi huomioida ainakin seuraavat aspektit: tehtävän suorittamiseen kuluva aika, optimaalisen kokoonpanon löytäminen, tietokantakyselyiden optimointi ja suorituskyvyn heikot kohdat. [Mosley, 2000]

Suorituskyky voidaan määritellä sovelluksen vasteajaksi erilaisissa olosuhteissa. Verkkosovelluksen kyseessä ollessa voidaan testata esimerkiksi sitä, miten monta pyyntöä sovelluspalvelin ehtii käsitellä sekunnissa. Sovelluksen luonteesta riippuen sille tulee määritellä suorituskykyvaatimukset. Testatessa monitorointityökalun avulla seurataan päästäänkö näihin vaatimuksiin. [Iyer et al., 2005]

Esimerkiksi Microsoft Visual Studion testausversio tukee kuormitustestien muodostamista ja suorittamista automaattisesti. Siinä on mahdollista simuloida oikeaa käyttäjää hyvin monella tavalla. Voidaan asettaa mietintäaika, joka kuvastaa sitä aikaa, joka käyttäjällä kuluu, ennen kuin hän valitsee jonkin toiminnon. On mahdollista määritellä, miten monella yhtäaikaisella käyttäjällä testi suoritetaan tai valita asteittain nouseva käyttäjämäärä. Käyttäjien painotus sovelluksen eri osa-alueille voidaan jakaa niin, että kun esimerkiksi 10 % käyttäjistä on kirjautumassa sisään ohjelmaan ja 90 % hakee tietoa palvelimelta. Samoin pystytään määrittelemään, millaisella palvelinkombinaatiolla testejä suoritetaan. Verkkotestauksessa voidaan määritellä käytettävät selaimet sekä määritellä käyttäjälle jokin tietyntasoinen verkkoyhteys.

Visual Studion kuormitustestityökalun käyttäminen kuitenkin vaatii, että testitapauksen löytyvät Visual Studiosta. Jos toiminnallista testausta on suoritettu Visual Studion työkaluilla, ei testitapausten luomisesta synny kovin merkittävää lisätyömäärää kuormitustestaukseen. Kuvassa 15 esitetään osa Visual Studion kuormitustestiautomaation antamasta tuloksesta. Siinä on hyvin kattavasti esillä kaikki verkkosovellukseen liittyvät kuormitustekijät.

Overall Results

Max User Load	5
Requests/Sec	10,2
Requests Failed	1
Requests Cached Percentage	34,8
Avg. Response Time (sec)	0,96
Avg. Content Length (bytes)	2 998
Tests/Sec	0,58
Tests Failed	1
Avg. Test Time (sec)	6,04
Avg. Transaction Time (sec)	0
Avg. Page Time (sec)	2,40

Test Results

Name	Scenario	Total Tests	Failed Tests (% of total)	Avg. Test Time (sec)
WebTest1	Scenario1	10	1 (10,0)	6,04

Page Results

URL (Link to More Details)	Scenario	Test	Avg. Page Time (sec)	Count
http://www.cs.uta.fi/	Scenario1	WebTest1	4,43	10
http://www.cs.uta.fi/opiskelu/	Scenario1	WebTest1	0,37	10

Kuva 15. Esimerkki Microsoft Visual Studion kuormitustestituloksista

Jos suorituskykyä ja kuormitusta joudutaan ohjelmistosta testaamaan, se nostaa testauksen työmäärää, koska ne täytyy suorittaa erillisinä testauksina ohjelman virheettömyydestä varmistumisen lisäksi. Näiden testaaminen ilman automatisointia on käytännössä mahdotonta, mutta riippuen sovelluksen rakenteesta ja suorituskykyvaatimuksista, työmäärä vaihtelee. Mitä tiukemmat vaatimukset ovat, sitä tarkemmin suorituskykytestaus tulee suoritta. Mitä suuremmalla kuormalla ohjelmiston tulee kyetä toimimaan, sitä laajemmat kuormitustestit sille pitää tehdä. Jos sovelluksessa on useampi kerros, kuten esimerkiksi tietokanta, sovelluspalvelin ja loppukäyttäjän selain, sitä suurempi työmäärä testauksessa on, koska jokainen kerros tulee testata suorituskyvyn ja kuormituksen kannalta.

6.5. Yhteenveto testauksen automatisoinnista

Automaatiolla voidaan vaikuttaa testauksen työmäärään paljonkin. Automatisointi ei kuitenkaan poista tehtävää työtä, eikä se välttämättä vähennä testausprojektien työmäärää niin paljon kuin haluttaisiin. Joissain tapauksissa se voi jopa kasvattaa työmäärää. Kuitenkin pitää muistaa, että kone on nopeampi ja yksiselitteisempi toimimaan formaaleissa asioissa kuin ihminen. Vaikka vaikutus ei olisi merkittävä suoraan työmäärään, se voi näkyä merkittävästi ohjelmiston laadun parantumisena.

Mosley [2000] nostaa esiin sellaisenkin tärkeän seikan, että jatkuvalla automatisoidulla testauksella voidaan vähentää ohjelmoijien halua uhrata laatu tuottavuudesta. Kun ohjelmoija tietää, että kaikki kirjoitettu koodi tullaan testaamaan, hän todennäköisesti pyrkii virheettömään tuotokseen. Tällä ei tarkoiteta, että ohjelmoijat tekisivät virheitä tahallisesti, mutta kiire ja suorituspaineet saattavat vähentää laadun prioriteettia.

Automatisointia harkitessa kannattaa miettiä, onko kannattavaa investoida automaattisiin työvälineisiin, eli ehditäänkö projektin elinaikana kuolettaa investointi säästyneenä työmääränä tai voidaanko ohjelmiston laadunnousua mitata rahallisesti. Testauksen automatisoinnista saatua hyötyä voidaan mitata ainakin seuraavilla kriteereillä: suunniteltujen, suoritettujen ja valmiiden testitapausten lukumäärä, virheiden toimintokohtainen lukumäärä, käytettyjen testituntien määrä löydettyä virhettä kohti ja käytettyjen korjaustuntien määrä virhettä kohti (korjaaminen ja uudelleentestaaminen). Sekä lisäksi voidaan verrata automatisoitua testausta manuaaliseen ja päätellä, onko löydettyjen virheiden määrä tai laatu noussut merkittävästi. [Harju ja Koskela, 2003]

Harju ja Koskela [2003] ehdottavat virheen esiintymisen todennäköisyyttä mitattavaksi seuraavien muuttujien avulla: toimintojen muutostiheys, funktion koko, monimutkaisuus ja suunnitteludokumentin laatu. Jokaiselle näistä muuttujista annetaan arvo yhdestä kolmeen (pienestä suureen). Tämän jälkeen he esittävät kuvassa 16 esiteltyä laskentakaavaa virheiden todennäköisyyden ennustamiselle jokaista käsiteltyä ohjelmiston toimintoa kohden.

$$Riski(f) = p(f)\frac{C(a)+C(t)}{2}$$

Kuva 16. Kaava virheen todennäköisyyden laskemiseen, kun:

p(f) kuvaa testattavan toiminnon laatua,
C(a) kuvaa seurausten kustannuksia asiakkaan kannalta ja
C(t) kuvaa seurausten kustannuksia toimittajan kannalta.
[Harju ja Koskela, 2003]

Kun testaustyökalua ollaan hankkimassa, Mosley [2000] ehdottaa seuraavaa aikataulua. Ensin pitäisi käyttää viikko siihen, että löydetään vaatimukset testityökalulle. Seuraavan kahden viikon aikana tulisi tutkia mahdollisten testityökalujen toiminnallisuuksia ja tehdä yhteenveto niiden ominaisuuksista. Tästä seuraava viikko tulisi käyttää erilaisten testityökalujen kokeilemiseen. Kun valitaan jokin testityökalu sopivana vaihtoehtona, sitä tulisi koekäyttää noin kuukauden ajan. Automatisointi-innostuksessa tulisi siis muistaa, että testityökalun pitää olla yrityksen projekteihin ja prosesseihin sopiva, jotta siitä on hyötyä. Testityökalun käyttöönotto myös vie runsaasti resursseja, joten ei ole taloudellisesti kannattavaa vaihtaa valittua työkalua usein.

Testitapausten automaattinen luominen vähentää testaajan työmäärää, jos testitapausten suuri määrä ei aiheuta ongelmia. Automaattinen generointi kuitenkin vaatii formaaleja dokumentteja ja dokumenttien ajantasaista ylläpitoa, ja tämä lisää työmäärää kokonaisuutena. Jos yrityksessä jo ennestään noudatetaan formaalia ja tiukkaa dokumentointimenettelyä, niin lisätyömäärää ei tietenkään synny. [Polo et al., 2007]

Automatisoitu yksikkötestaus parantaa testaajalle tulevan koodin laatua. Tällöin testaajan ei tarvitse kuluttaa aikaa esim. raja-arvotestaukseen eikä yksinkertaisimpien ohjelmistovirheiden raportointiin. Automatisoitujen yksikkötestien luominen vie kehittäjältä aikaa, mutta pidemmässä projektissa tämä laadunvarmistusmenetelmää voidaan käyttää yhtä uudelleen. Testitapauksia tosin tulee ylläpitää siinä missä koodiakin. Apuna tähän voi olla AutoTestin kaltaiset ohjelmistot, jotka vaativat vähemmän manuaalista ylläpitoa.

Toiminnallisen testauksen automatisointiin on olemassa apuvälineitä, mutta monet niistä vaativat vielä testaajalta manuaalista työtä tai taitoa ohjelmoida. Sopivassa projektissa työvälineet kuitenkin vähentävät työmäärää. Suorituskyky- ja kuormitustestejä on käytännössä mahdotonta suorittaa ilman työkalutukea. Niiden suorittamiseen on kuitenkin olemassa erinomaisia työkaluja, jotka kohtuullisen pienellä työmäärällä antavat monipuolisia tulosraportteja.

7. Yhteenveto

Testauksen tarkoituksena on varmistaa, että ohjelmisto toimii virheittä ja sisältää sovitut toiminnot. Testauksen kuuluisi kattaa koko ohjelmistoprojektin elinkaari laadunvalvontamielessä. Ohjelmistoprosessista riippuen testausta voidaan suorittaa pienissä erissä tai koko testaamalla koko ohjelmisto kerralla. Testausta voidaan suorittaa ohjelmistoprojektissa monella eri tasolla, määrittelydokumenttien katselmoinnista ohjelmiston tilaajan suorittamaan hyväksymistestaukseen.

Tutkielman tavoitteena oli arvioida perinteisten ohjelmistotuotannon työmääräarviomallien sopivuutta testausprojekteille sekä nostaa esiin testausprojektien erityispiirteitä työmääräarviointiin liittyen. Tutkielmassa on esitelty joitakin työmääräarviomalleja ja arvioitu niiden soveltumista testausprojektien työmääräarviointiin. Arvioinnissa päädyttiin siihen, että toimintopisteperustaiset menetelmät soveltuvat parhaiten olemassa olevista malleista testausprojektienkin työmääräarviointiin.

Tutkielmassa esiteltiin myös erilaisia ohjelmistotuotannon menetelmiä ja prosesseja, joilla testausprojektien työmäärään voidaan vaikuttaa. Eniten testausprojektien työmäärään voidaan vaikuttaa sillä, että laatu huomioidaan projektin alusta lähtien. Tällä tarkoitetaan dokumenttien katselmointeja, kooditarkastuksia ja ohjelmoijien suorittamaa yksikkötestausta.

Tutkielmassa arviointiin automatisoinnin merkitystä testausprojekteille ja tultiin siihen tulokseen, että automatisoinnilla voidaan pitkäkestoisissa ja ylläpidettävissä ohjelmistoissa vaikuttaa työmäärään sitä alentavasti. Erityisesti yksikkötestauksen automatisointiin on jo olemassa laadukkaita työkaluja. Toiminnallisen testauksen automatisointiin tarkoitetuissa työkaluissa on vielä paljon parantamisen varaa, sillä toistaiseksi niiden käyttäminen lyhytkestoisissa projekteissa ei ole työmäärän näkökulmasta kannattavaa.

Käyttäjäkunnan kasvu ja yhä kriittisemmät sovellusalueet aiheuttavat ohjelmistojen laatukriteerien kasvamisen. Tästä syystä testauksen merkitys ohjelmistoprojekteissa kasvaa tulevaisuudessa entisestään. Loppukäyttäjäkunnan laajeneminen luo lisää paineita ohjelmistojen käytettävyydelle, virheensiedolle ja suorituskyvylle. Ohjelmistojen kasvaminen sekä arkaluonteisten tietojen käsittely verkon yli lisää haastetta rajapintojen ja tietoturvallisuuden testaamiselle.

Testauksen työkaluja kehitetään koko ajan automaattisempaan suuntaan ja testauksen työmäärän näkökulmasta eräät tutkitut ominaisuudet toisivat merkittävästi hyötyä. Tulevaisuudessa voi olla tarjolla puoliälykkäitä automatisointityökaluja, jotka osaavat esimerkiksi käyttöliittymästä etsiä kaikki mahdolliset käyttäjän toimintapolut ja jotka automaattisesti luovat näistä poluista suoritettavat automaattiset testit. Dynaamisten ohjelmistojen testaamisen avuksi on kehitteillä työkalu, joka osaa luoda käsityksen tie-

tokannan tilasta suhteessa mahdollisiin testitapauksiin. Tällaisilla automatisointityökaluilla pystyttäisiin tehokkaasti vähentämään testauksen työmäärää tämänhetkisestä.

Viiteluettelo

[Andersson et al., 2003] Johan Andersson, Geoff Bache and Peter Sutton, XP with acceptance-test driven development: a rewrite project for a resource optimization system. In: M. Marchesi and G. Succi (Eds.), *XP 2003*, LNCS 2675, Springer, 2003, 180-188.

[Ali-Yrkkö and Jain, 2005] Jyrki Ali-Yrkkö and Monika Jain, Offshoring software development – case of Indian firms in Finland, Elinkeinoelämän tutkimuslaitos, No.971, 7.3.2005

[Alshraideh and Bottaci, 2006] Mohammad Alshraideh and Leonardo Bottaci, Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability*, **16** (2006), 175-203

[Arcuri and Yao, 2008] Andrea Arcuri and Xin Yao, Search based software testing of object-oriented containers. *Information Sciences*, **178** (2008), 3075-3095.

[Baresi and Bezze, 2006] Luciano Baresi and Mauro Pezze, An introduction to software testing, *Electronic Notes in Theoretical Computer Science,* **148** (2006), 89-111.

[Belifante et al., 2005] Axel Belinfante, Lars Frantzen and Christian Schallhart, Tools for test case generation. In: M. Broy et al. (Eds.), *Model-Based Testing os Reactive Systems*, LNCS 3472, Springer-Verlag, 2005, 391-438.

[Binder, 1996] Robert V. Binder Testing object-oriented software: a survey. *Software Testing, Verification and Reliability*, **6** (1996), 125-252.

[Bygstad et al., 2008] Bendik Bygstad, Gheorghita Ghinea and Eivind Brevik, Software development methods and usability: perspectives from a survey in the software industry in Norway, *Interacting with Computers,* **20** (2008), 375-385.

[Capretz and Ahmed, 2010] Luiz Fernando Capretz and Faheem Ahmed, Why do we need personality diversity in software engineering. *ACM SIGSOFT Software Engineering Notes*, **35**, 2 (2010), 1-11.

[Chrissis et al., 2003] Mary Beth Chrissi, Mike Konrad and Sandy Shrum, *CMMI Guidelines for Process in Integration and Product Improvement*, Addison-Wesley Longman Publishing Co, 2003.

[Chung and Bieman, 2009] Insang Chung and James M. Bieman, Generating input data structures for automated program testing. *Software Testing, Verification and Reliability*, **19** (2009), 3-36

[Cohen et al., 2004] David Cohen, Mikael Lindvall and Patricia Costa, An introduction to agile methods. In: Marvin V. Zelkowitz, *Advances in Computers, **62***, Elsevier, 2004.

[Collopy, 2007] Fred Collopy, Difficulty and complexity as factors in software effort estimation. *International Journal of Forecasting*, **23** (2007), 469–471.

[Dallal and Sorenson, 2005] Jehad Al Dallal and Paul Sorenson, Reusing class-based test cases for testing object-oriented framework interface classes. *Journal of Software Maintenance and Evolution: Research and Practice*, **17** (2005), 169-196.

[Di Lucca and Fasolino, 2006] Giuseppe A. Di Lucca and Anna Rita Fasolino, Testing web-based applications: the state of the art and future trends. *Information and Software Technology*, **48** (2006), 1172-1186.

[Dubinsky and Hazzan, 2004] Yael Dubinsky and Orit Hazzan, Roles in agile software development teams. In: *Extreme Programming and Agile Processes in Software Engineering*, Springer, Berlin/Heidelberg, 2004, 157-165.

[Everett and McLeod, 2007] Gerald D. Everett and Raymond McLeod, *Software Testing: Testing Across the Entire Software Development Life Cycle*. Wiley-IEEE, 2007.

[Folmer and Bosch, 2005] Eelke Folmer and Jan Bosch, Cost effective development of usable systems: gaps between HCI and software architecture design. In: Anders G. Nilsson, Remigijus Gustas, Wita Wojtkowski, W. Gregory Wojtkowski, Stanislav Wrycza and Joze Zupancic, *Advances in Information Systems Development*, Springer, 2005, 337-348

[Forselius, 1999], Pekka Forselius, Ohjelmistojen koon mittaaminen erityyppisissä kehityshankkeissa. *Systeemityö*, **1** (1999), 14-18.

[Fraser et al., 2009] Gordon Fraser, Angelo Gargantini and Franz Wotawa, On the order of test goals in specification-based testing. *The Journal of Logic and Algerbraic Programming*, **78** (2009), 472-490.

[Frezza et al., 1996] Stephen T. Frezza, Steven P. Levitan and Panos K. Chrysanthis, Linking requirements and design data for automates functional evaluation. *Computers in Industry*, **30** (1996), 13-25.

[Georgiadou, 2004] Elli Georgiadou, Software process and product improvement a historical perspective. *International Journal of Cybernetics*, **1**, 1(2003), 172-197.

[Harju ja Koskela, 2003] Hannu Harju ja Mika Koskela, Kustannustehokas ohjelmiston luotettavuuden suunnittelu ja arviointi. Osa 2. *VTT Tiedotteita – Research Notes* **2193**, 2003.

[Hartman and Nagin, 2005] Alan Hartman and Kenneth Nagin, The AGEDIS tools for model based testing. In: Jardim Nunes et al. (eds.), *UML 2004 Satellite Activities*, LNCS 3297, Springer, 2005, 277-280.

[Helminen, 2008] Heli Helminen, Työmääräarviointi ja aikataulusuunnittelu IT-projekteissa. Tampereen yliopisto, pro gradu -tutkielma, 2008.

[Honkela, 2005] Suvi Honkela, Testausprosessi ja sen hallinta –automatisoinnin näkökulma, Jyväskylän yliopisto, pro gradu -tutkielma, 2005.

[Hughes and Cotterell, 2006] Bob Hughes and Mike Cotterell, *Software Project Management.* McGraw-Hill, 2006.

[Huttunen, 2006] Janne Huttunen, Ketterän ohjelmistokehitysmenetelmän määrittely, vertailu ja käyttäjäkysely. Teknillinen korkeakoulu, Diplomityö, 2006.

[Iyer et al., 2005] Lakshmi S. Iyer, Babita Gupta and Nakul Johri, Performance, scalability and reliability issues in web applications. *Industrial Management & Data Systems*, **105** (2005), 561-576.

[Jones, 2007] Capers Jones, *Estimating Software Costs: Bringing Realism to Estimating*. McGraw-Hill, 2007.

[Kakkonen, 2009] Kari Kakkonen, Testaus hajautettuna suunnitelmavetoisessa ja ketterässä mallissa. *Systeemityö*, **16**, 1 (2009), 12–14.

[Kansomkeat and Rivepiboon, 2008] Supaporn Kansomkeat and Wanchai Rivepiboon, An analysis technique to increase testability of object-oriented components. *Software testing, Verification and Reliability*, **18** (2008), 193-219.

[Kollanus, 2006] Sami Kollanus, Ohjelmistojen tarkastuskäytänteiden puutteet ja ongelmat teoriassa ja käytännössä. Jyväskylän yliopisto, lisensiaattityö, 2006.

[Kortum, 2008] Philip Kortum, Introduction to the human factors of nontraditional interfaces. In: Philip Kortum, *HCI Beyond the GUI*, Elsevier, 2008.

[Koskimies, 2000] Kai Koskimies, *Oliokirja*, Satku, 2000.

[Kruchten, 2003] Philippe Kruchten, *The Rational Unified Process an Introduction*. Addison-Wesley, 2003.

[Kumari et al., 2009] Mitu Kumari, Archana Sharma and Vipin Kamboj, Replacement of s/w inspection with s/w testing. *International Journal of Information Technology and Knowledge Management*, **2** (2009), 257-261.

[Kundu et al., 2009] Debasish Kundu, Monalisa Sarma, Debasis Samanta and Rajib Mall, System testing for object-oriented systems with test case priorization. *Software Testing, Verification and Reliability*, **19** (2009), 297-333.

[Kyllönen, 2008] Tuula Kyllönen, Ohjelmistojen testauksen kehittäminen ja parantaminen. Joensuun yliopisto, pro gradu -tutkielma, 2008.

[Li and Wu, 2005] Kanglin Li and Mengqi Wu, *Effective GUI Test Automation: Developing an Automated GUI Testing Tool*, SYBEX, 2005.

[Liu and Chen, 2008] Shaoying Liu and Yuting Chen, A relation-based method combining functional and structural testing for test case generation. *The Journal of Systems and Software*, **81** (2008), 234-248.

[Lucio and Samer, 2005] Levi Lucio and Marko Samer, Technology of test-case generation. In: M. Broy et al. (Eds.), *Model-Based Testing os Reactive Systems*, LNCS 3472, Springer-Verlag, 2005, 323-354.

[Lutsky, 2000] Patricia Lutsky, Information exraction from documents for automating software testing. *Artificial Intelligence in Engineering*, **14** (2000), 63-69.

[Makkonen, 2008] Jussi Makkonen, Testauslähtöinen ohjelmistokehitys. Joensuun yliopisto, pro gradu -tutkielma, 2008.

[Martin, 2003] Robert Cecil Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2003.

[McDonald, 2005] James McDonald, The impact of project planning team experience on software project cost estimates. *Empirical Software Engineering*, **10** (2005), 219–234.

[Memon et al., 2001] Atif M. Memon, Martha E. Pollack and Mary Lou Soffa, Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, **27** (2001), 144-155.

[Meyer et al., 2007] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner and Lisa Ling Liu, Automatic testing of object-oriented software. In: Jan van Leeuwen et al. (Eds.), *SOFSEM 2007*, LNCS 4362, Springer, 2007, 114-129.

[Mitro, 2009] Jari Mitro, Loppukäyttäjä testaajana – kokemuksia keskeneräisistä tuotteista. *Systeemityö*, **16,** 1 (2009), 22–23.

[Mohan et al., 2008] K. Krishna Mohan, A. Srividya and Ravi Kumar Gedela, Quality of service prediction using fuzzy logic and RUP implementation for process oriented development. *International Journal of Reliability, Quality and Safety Engineering*, **15** (2008), 143-158.

[Moløkken-Østvold and Jørgensen, 2004] Kjetil Moløkken-Østvold and Magne Jørgensen, Group processes in software effort estimation. *Empirical Software Engineering,* **9** (2004), 315–334.

[Mosley, 2000] Daniel J. Mosley, *Client-Server Software Testing on the Desktop and the Web*. Prentice Hall, 2000.

[Mosley and Posey, 2002] Daniel J. Mosley and Bruce A. Posey, *Just Enough Software test Automation*, Prentice Hall, 2002.

[Myers, 2004] Glenford J. Myers, *The Art of Software Testing*. John Wiley & Sons, 2004.

[Määttä et al., 2009] Juha Määttä, Janne Härkönen, Tauno Jokinen, Matti Möttönen, Pekka Belt, Matti Muhos and Harri Haapasalo, Managing testing activities in telecommunications: a case study. *Journal of Engineering and Technology Management*, **26** (2009), 73-96.

[Olague et al., 2008] Hector M. Olague, Letha H. Etzkorn, Sherri L. Messimer and Harry S. Delugach, An empirical validation of object-oriented class complexity metrics and their ability to predict error-prone classes in highly iterative, or agile, software: a case study. *Journal of Software Maintenance and Evolution: Research and Practice*, **20** (2008), 171-197.

[Ormandjieva et al., 2008] O. Ormandjieva, V.S. Alagar and M. Zheng, Early quality monitoring in the developmet of real-time reactive systems. *The Journal of Systems and Software,* **81** (2008), 1738-1753.

[Paiva et al., 2007] Ana C. R. Paiva, Joao C. P. Faria and Raul F. A. M. Vidal, Towards the integration of visual and formal models for GUI testing. *Electronic Notes in Theoretical Computer Science*, **190** (2007) 99–111.

[Patton, 2005] Ron Patton, *Software Testing*. Sams, 2005.

[Polo et al., 2007] Macario Polo, Sergio Tendero and Mario Piattini, Integrating techniques and tools for testing automation, *Software Testing, Verification and Reliability*, **17** 2007 3-39**.**

[Pressman, 1997] Roger S. Pressman, *Software Engineering a Practitioner's Approach*. McGraw-Hill, 1997.

[Pyykkö, 2010] Timo Pyykkö, Ohjelmistotestaus siirryttäessä perinteisistä ohjelmisto-kehitysmenetemistä Scrumiin. Jyväskylän yliopisto, pro gradu -tutkielma, 2010

[Ran et al., 2009] Lihua Ran, Curtis Dyreson, Anneliese Andrews, Renee Bryce and Christopher Mallery, Building test cases and oracles to automate the testing of web database applications. *Information and Software Technology*, **51** (2009), 460-477.

[Soini et al., 2006] Jari Soini, Vesa Tenhunen and Markku Tukiainen, Current practices of measuring quality in Finnish software engineering industry. In: Ita Richardson; Per Runeson & Richard Messnarz (eds.), *Software Process Improvement*, LNCS 4257, Springer, 2006, 100-110.

[Staron and Meding, 2008] Miroslaw Staron and Wilhelm Meding, Predicting weekly defect inflow in large software projects based on project planning and test status. *Information and Software Technology*, **50** (2008) 782-796.

[Taipale et al., 2005] Finding and ranking research directions for software testing, Ossi Taipale, Kari Smolander and Heikki Kälviäinen. In: I. Richardson et al. (eds.), *Software Process Improvement*, LNCS 3792, Springer, 2005, 39-48.

[Trammell et al., 1996] Carmen J. Trammell, Mark G. Pleszkoch, Richard C. Linger and Alan R. Hevner, The incremental development process in Cleanroom software engineering. *Decision Support Systems*, **17** (1996), 55-71.

[Virkanen, 2002] Hannu Virkanen, Ohjelmistojen testaus ja virheenjäljitys. Kuopion yliopisto, pro gradu -tutkielma, 2002.

[Wieczorek, 2002] Isabella Wieczorek, Improved software cost estimation – a robust and interpretable modelling method and a comprehensive empirical investigation. *Empirical Software Engineering*, **7** (2002), 177–180.

[Woodward and Hennell, 2005] Martin R. Woodward and Michael A. Hennell, Strategic benefits of software test management: a case study. *Journal of Engineering and Technology Management*, **22** (2005), 113-140.

[Yoo and Harman, 2010] S. Yoo and M. Harman, Regression testing minimization, selection and prioritization: a survey. *Software testing, Verification and Reliability,* to appear (2010).

[Yuan and Memon, 2010] Xun Yuan and Atif M. Memon, Iterative execution-feedback model-directed GUI testing. *Information and Software Technology*, **52** (2010), 559-575.

[Zhou et al., 2010] Yuming Zhou, Baowen Xu and Hareton Leung, On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *The Journal of Systems and Software*, 83, (2010), 660-674.