TESTAUKSEN AUTOMATISOINTIPROSESSIN KEHITTÄMINEN TERVEYDENHUOLLON TIETOJÄRJESTELMÄÄN

Kaisa-Mari Vehkaperä
2010
Oulun seudun ammattikorkeakoulu

TESTAUKSEN AUTOMATISOINTIPROSESSIN KEHITTÄMINEN TERVEYDENHUOLLON TIETOJÄRJESTELMÄÄN

Kaisa-Mari Vehkaperä
Opinnäytetyö
15.10.2010
Hyvinvointiteknologian koulutusohjelma
Oulun seudun ammattikorkeakoulu

OULUN SEUDUN AMMATTIKORKEAKOULU TIIVISTELMÄ

Koulutusohjelma	Opinnäytetyö	Sivuja + Liitteitä
Hyvinvointiteknologia	Opinnäytetyö	43 + 0
Suuntautumisvaihtoehto	Aika	
Sairaalateknologia	2010	

Työn tilaaja	Työn tekijä
Yritys X	Kaisa-Mari Vehkaperä

Työn nimi
Testauksen automatisointiprosessin kehittäminen terveydenhuollon tietojärjestelmään

Asiasanat
ohjelmistotestaus, testauksen automatisointi, automatisoinnin apuvälineet

Tämän opinnäytetyön aiheena oli kehittää pienen ohjelmistokehitysyrityksen testausprosessia. Tämä toteutettiin selvittämällä nykyiset testauskäytännöt sekä tutkimalla testauksen automatisointimahdollisuuksia. Lisäksi etsittiin sopivia työkaluja sen toteuttamiseksi.

Työ toteutettiin haastattelemalla opinnäytetyön tilaajayrityksen työntekijöitä. Teoreettiset tiedot ohjelmiston testausprosessista ja automatisoinnista sekä sen mahdollistavista työkaluista selvitettiin alan kirjallisuutta sekä internetsivustoja hyväksikäyttäen.

Työn tuloksina suositeltiin yritykselle testausprosessin automatisointia tukevia ilmaisia työkaluja. Lisäksi selvitystyössä ilmi tulleet automatisoimiseen liittyvät haasteet ja kustannukset ovat yritykselle hyödyllistä tietoa arvioitaessa tarvetta muutoksille.

SISÄLTÖ

TIIVISTELMÄ. 3
SISÄLTÖ 4
1 JOHDANTO 6
2 TESTAUSPROSESSI 7
2.1 Testaustasot 7
2.1.1 Yksikkötestaus 8
2.1.2 Integrointitestaus 9
2.1.3 Toiminnallinen testaus 9
2.2 Testivetoinen ohjelmistokehitys 10
2.3 Projektin hallintatyökalut 11
3 TESTAUSPROSESSIN NYKYTILANTEEN KUVAUS 13
3.1 Kehitysvaihe 13
3.2 Toiminnallisen testauksen vaihe ja tuotantoasennus 14
4 TESTAUSPROSESSIN AUTOMATISOINTI 16
4.1 Testitapausten hyvyys 16
4.2 Automatisoinnin tavoitteet ja hyödyt 17
4.3 Automatisoinnin haasteet 19
4.4 Automatisoinnin ylläpidettävyys 20
4.4.1 Ongelmat 20
4.4.2 Ratkaisuja 21
5 AUTOMATISOINNIN KOHTEET 22
5.1 Automatisoitavat toiminnot 22
5.2 Suunnittelun automatisoiminen 23
5.3 Suorituksen automatisointi 24
5.3.1 Testisyötteen nauhoittaminen 25
5.3.2 Nauhoittamisen ongelmat 25
5.4 Vertailun automatisointi 26
6 TESTAUKSEN APUVÄLINEET 28
6.1 Apuvälineet ohjelmistokehityksen elinkaaren aikana 28
6.2 Apuvälineen valinta testauksen automatisoinnissa 30

6.3 Apuvälineen käyttöönotto 31
7 TESTAUKSEN AUTOMATISOINNIN APUVÄLINEET 32
7.1 Ilmaiset apuvälineet 32
7.1.1 MaxQ 32
7.1.2 iValidator 33
7.1.3 Concordion 33
7.1.4 Cucumber 33
7.2 Maksulliset apuvälineet 34
7.2.1 SilkTest 34
7.2.2 HP QuickTest Professional Software 35
8 JOHTOPÄÄTÖKSET 37
9 POHDINTA 39
LÄHTEET 41

1 JOHDANTO

Ohjelmistoprojektissa lopullisen tuotteen laadun varmentaminen on tärkeää, sillä ohjelmiston täytyy täyttää sille asetetut vaatimukset, joita ovat ohjelmistosta riippuen esimerkiksi asetettujen tehtävien suorittaminen, käytettävyys, virhesietoisuus ja tehokkuus. Lisäksi erityisesti terveydenhuollon ympäristössä toimivien sovellusten täytyy olla turvallisia, sillä ne eivät saa vaarantaa potilastietoja, jotka ovat luottamuksellisia asiakirjoja.

Varsinainen testausprosessi, jolla varmistetaan ohjelmiston laatu, on tärkeä kehittämisen ja parantamisen kohde ja sen tulee pystyä vastaamaan muuttuvien ohjelmistojen vaatimuksiin. Tässä opinnäytetyössä keskityttiin selvittämään pienen hyvinvointiteknologia-alan ohjelmistokehitysyrityksen nykyinen testausprosessi. Lisäksi selvitettiin testauksen automatisointimahdollisuuksia sekä mahdollisia työkaluja sen toteuttamiseen.

Työ toteutettiin tekemällä yhteistyötä työn tilaajan kanssa keskustelemalla työntekijöiden kanssa. Teoreettiset tiedot ohjelmiston testausprosessista ja automatisoinnista sekä sen mahdollistavista työkaluista selvitettiin alan kirjallisuutta ja internetsivustoja hyväksikäyttäen. Varsinaisten työkalujen käyttöönotto jäi tämän opinnäytetyön ulkopuolelle.

2 TESTAUSPROSESSI

Ohjelmistojen laadukkuus on tärkeä kriteeri asiakkaille. Ohjelman tulee olla käytettävyydeltään hyvä sekä täyttää sille asetetut tehtävät. Lisäksi ohjelman tulee olla hyvin tehokas, eikä se saa tuhlata turhia resursseja. Ohjelman on myös oltava virhesietoinen sekä ylläpidettävä, jotta tuote on muokattavissa muuttuviin vaatimuksiin. Näiden takaamiseksi täytyy ohjelmistoilla olla takana hyvä ja toimiva testausprosessi. (Haikala – Märijärvi 2006, 283.)

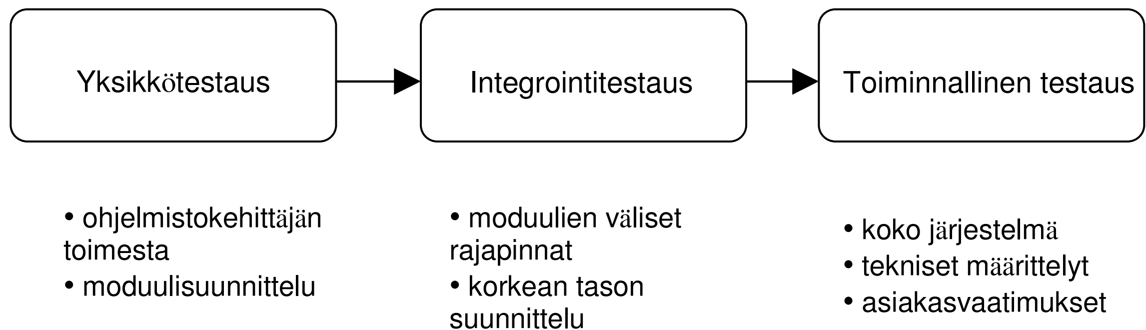
Testausprosessi ei ole irrallinen osa ohjelmistoprojektia vaan prosessi, jonka täytyy olla mukana ohjelmistokehityksen joka vaiheessa, jotta päästäisiin testauksen tärkeimpään tavoitteeseen, löytää ohjelmistovirheet mahdollisimman varhaisessa vaiheessa. Kun virheet löydetään ohjelmistotuotannon alkuvaiheessa, on niiden korjaaminen edullisempaa ja yksikertaisempaa. Virheettömään ohjelmaan on mahdotonta päästä, mutta virheiden korjaaminen mahdollisimman varhaisessa vaiheessa lisää ohjelmiston luotettavuutta. (Haikala – Märijärvi 2006, 275.)

2.1 Testaustasot

Testaustasot määrittelevät, mille järjestelmän tasolle testaus ulottuu. Testaustasoja ovat yksikkö- eli moduulitestaus, integrointitestaus ja toiminnallinen testaus. Lisäksi testaustasosta riippumatta suoritettavaa uudelleentestausta kutsutaan regressiotestaukseksi. (Haikala – Märijärvi 2006, 288–290.)

Kuvassa 1 on esitetty kolme eri testaustasoa: yksikkö-, integrointi- ja toiminnallinen testaus. Tasoja ei suoriteta pelkästään kertaluontoisesti, sillä ohjelmointimuutosten vaatima uudelleentestaus tekee testaustoiminnasta jatkuvaa. Testaus etenee yksittäisten moduulien testaamisesta aina koko järjestelmän tutkimiseen. Testausvaiheet on suunniteltu ennen testejä, jolloin on myös määritelty

esimerkiksi moduulien rakenteet. Näin testauksen tulokset ovat myös verrattavissa suunnitelmiin. (Stenberg 2007, 23–25.)



KUVA 1. Yritys X:n käyttämät testaustasot

2.1.1 Yksikkötestaus

Yksikkötestaus on ohjelmistokehittäjän suorittamaa testausta. Yksikkönä voi olla esimerkiksi ohjelmiston metodi tai luokka, jota testataan itsessään, välittämättä muusta kokonaisuudesta. Kun voidaan varmistua, että jokainen yksikkö toimii halutulla tavalla, on suurempi todennäköisyys, että myös kokonaisuus toimii. (Rainsberger 2005, 4–5.)

Yksikkötestauksessa tuloksia verrataan tavallisimmin tekniseen määrittelydokumenttiin, missä moduulien tehtävät ovat määriteltyinä, eli etsitään koodin eroavaisuuksia tavoitemäärittelystä. Hyvällä yksikkötestauksella saavutetaan myös hyvä lausekattavuus. Sataprosenttisen lausekattavuuden saavuttamiseksi jokainen ohjelman lause suoritetaan vähintään kerran. Luonnollisesti tämä ei kuitenkaan takaa sitä, että kaikki mahdolliset tilanteet ja vaihtoehdot olisi testattu. (Haikala – Märijärvi 2006, 83, 289, 295.)

JUnit-testit ovat Javalla kirjoitettuja yksikkötestejä. JUnit kehitettiin mahdollistamaan testitapausten automatisointi, ryhmittely ja testien ajaminen helpolla tavalla, jotta kehittäjä voi keskittyä itse ohjelmointityöhön. Lopuksi testiajon tuloksena kehittäjä saa raportin hylätyistä testeistä. JUnit-testit muodostuvat itsenäisiksi yksikkötesteiksi, jolloin kuka tahansa pystyy ajamaan testit ja tarkastelemaan

tuloksia tietämättä, mitä koodissa täsmälleen tapahtuu. (Rainsberger 2005, 8–9.)

2.1.2 Integrointitestaus

Integrointitestausta suoritetaan usein yksikkötestauksen rinnalla tai sen jälkeen. Moduulit kootaan ryhmiksi, joiden välistä toimivuutta testataan, eli tutkitaan virheitä moduulien välisissä rajapinnoissa. Yleensä virheet tässä vaiheessa kertovat suunnitteluvirheistä ja virheet yksikkötestauksessa virheistä koodissa. (Haikala – Märijärvi 2006, 290.)

Moduuleja voidaan testata usealla eri menetelmällä. Integrointitestaus voidaan suorittaa jäsentävästi integroimalla ja testaamalla ensin korkean tason moduulit ylhäältä alas (top-down-menetelmä) tai kokoavasti testaamalla alemman tason moduulit alhaalta ylös (bottom-up-menetelmä). Nämä menetelmät voidaan myös yhdistää. (Haikala – Märijärvi 2006, 290.)

2.1.3 Toiminnallinen testaus

Toiminnallisessa testauksessa tarkastellaan koko järjestelmää ja varmistetaan, että se toimii määrittelydokumenttien osoittamalla tavalla. Näkökulma testaukseen on ulkoinen ja tarkastelun kohteena on se, mitä ohjelma tekee. Testauksen suorittajana tulee olla henkilö, joka on ollut mahdollisimman vähän mukana kehitystyössä. Tällöin testaaja uskaltaa kokeilla myös ohjelmiston rajoja ja kestävyyttä. (Haikala – Märijärvi 2006, 290.)

Toiminnallista testausta ovat käyttöliittymätestaus, järjestelmien välisten liittymien testaus ja käyttötapaukset. Lisäksi testataan järjestelmän ei-toiminnalliset ominaisuudet, kuten kuormitustestit, luotettavuustestit, asennustestit ja käytettävyystestit. Kuormitustestauksessa testataan ohjelmiston äärirajoja syöttämällä suuri tietomäärä lyhyessä ajassa sekä stabiilisuutta sopivalla tietomäärällä pit-

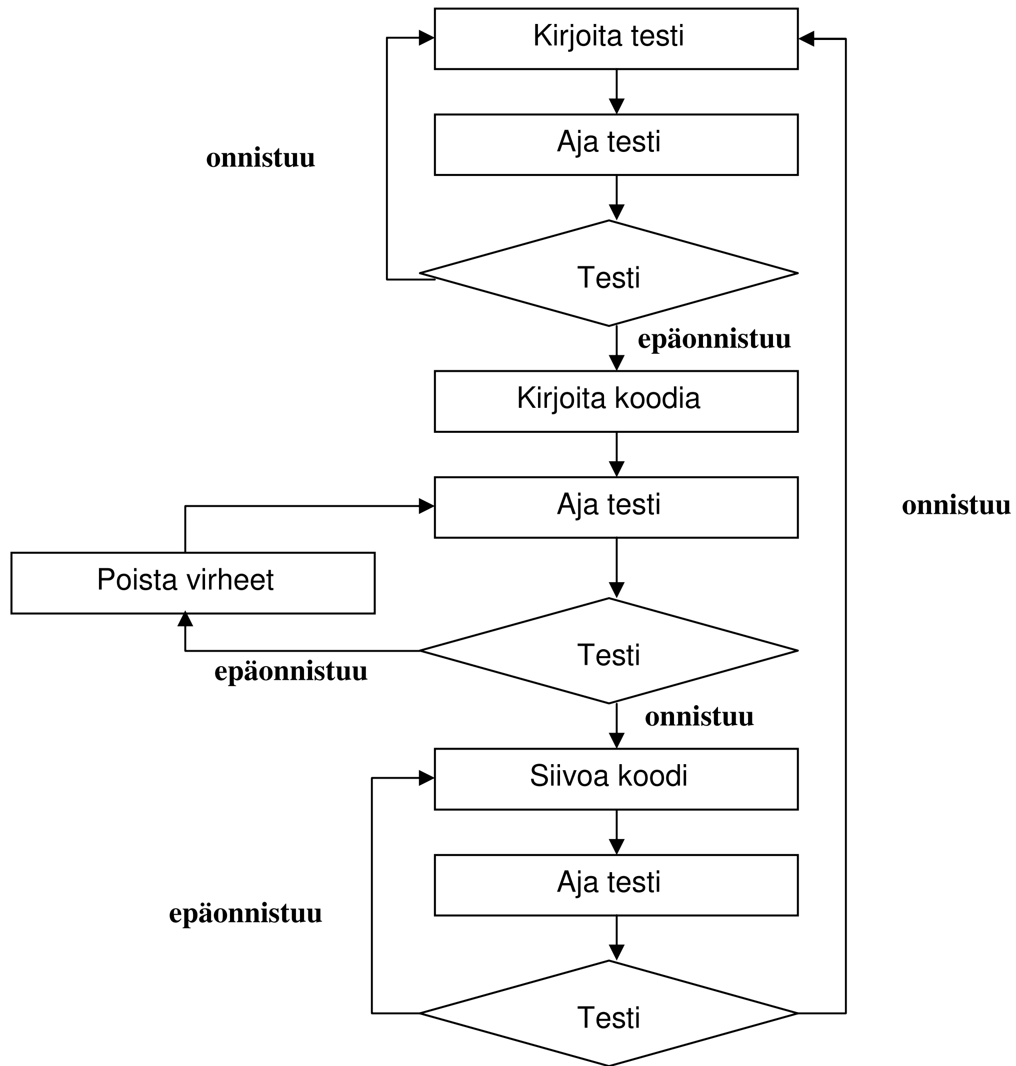
källä ajalla. Luotettavuustestauksessa tutkitaan ympäristön häiriötekijöitä ja tietoturvaa. (Haikala – Märijärvi 2006, 290.)

Toiminnallisen testauksen vaiheessa havaitut virheet ja niiden korjaus voivat aiheuttaa muutoksia useisiin moduuleihin eli ohjelmiston osiin. Tehdyt muutokset täytyy testata yksikkötesteistä lähtien sekä tutkia niiden vaikutus muihin moduuleihin ja lopuksi jälleen suorittaa toiminnallinen testaus. Uudelleentestausprosessia nimitetäänkin regressiotestaukseksi. (Haikala – Märijärvi 2006, 290.)

2.2 Testivetoinen ohjelmistokehitys

Testivetoisessa ohjelmistokehityksessä (TDD, Test-Driven Development) kirjoitetaan testit ennen varsinaista ohjelmistokoodia ja itse koodi laaditaan läpäisemään testi. Näin testit kattavat koko järjestelmän, joka muodostuu yhtenäisistä objekteista. Varsinainen testaustekniikka TDD ei siltikään ole, vaan se on suunnittelukäytäntö, jonka avulla kehittäjä joutuu suunnittelemaan ohjelmiston toteutuksen ennen varsinaisen koodin laatimista. (Rainsberger 2005, 9.)

TDD:n voidaan ajatella muodostuvan kolmesta vaiheesta kuten kuvassa 2. Ensimmäiseksi laaditaan testi, jota ei aina läpäistä, sillä itse ohjelmakoodia ei ole vielä olemassa. Seuraavaksi kirjoitetaan ohjelma, jota muutetaan niin kauan, että testi läpäistään. Viimeisessä vaiheessa koodista siivotaan kaikki ylimääräinen pois eli se rakennetaan uudelleen. Näin koodista saadaan helpommin ymmärrettävää ja ylläpidettävää, eikä se sisällä mitään turhaa. Näitä vaiheita voidaan toistaa, kunnes ohjelma on täysin valmis. TDD-testit eivät silti todista ohjelman virheettömyyttä, mutta sen, että se toimii testien mukaan. Testejä voidaan silti hyödyntää yksikkötestauksen apuna. (Ambler 2009.)



KUVA 2. TDD-sykli (Siniaalto 2006, 6)

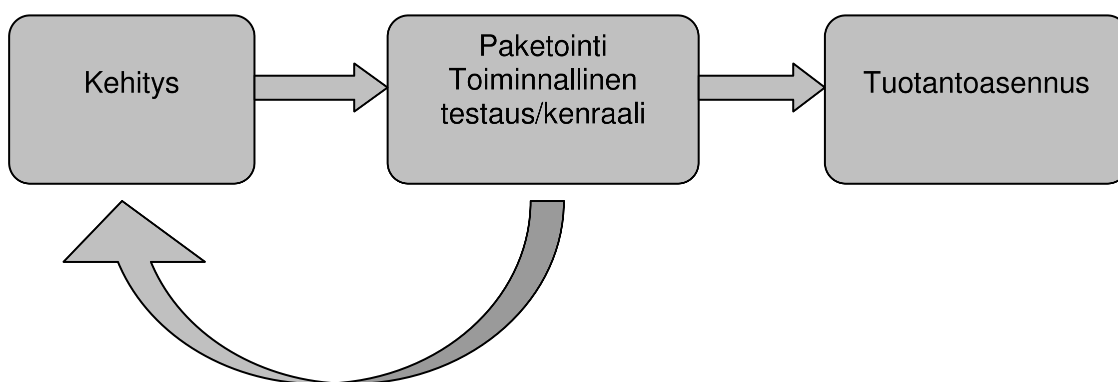
2.3 Projektin hallintatyökalut

Ohjelmistotuotantoprosessissa projektinhallintaa auttavat hallintatyökalut. Työkaluna voi toimia sovellus, jonne kootaan projektissa työskentelevien työtunnit, tiedot projektiorganisaatiosta, tehtävät toteuttavine ominaisuuksineen ja pienine kehitystehtävineen, aikataulutus ja seuranta. Tämä mahdollistaa myös sen, että ohjelmistokehittäjien resurssit saadaan kohdennettua oikein parhaalla mahdollisella tavalla. (Haikala – Märijärvi 2006, 226–228.)

Projektin hyvä hallinnointi mahdollistaa myös muutosten seurannan. Lisäksi virhelistat ja versionhallintaan liitettävät tiedot tehdyistä muutoksista esimerkiksi komponentteihin mahdollistavat jäljitettävyyden. Kun kaikki muutokset, tehtävät ja virhelistat yksityiskohtineen kirjataan ylös ja päivitetään säännöllisesti, tukee se kehittäjien vakaata työympäristöä. Näin he eivät myöskään häiritse toistensa työskentelyä suorittamalla esimerkiksi samaa moduulin korjausta yhtä aikaa. (Haikala – Märijärvi 2006, 260.)

3 TESTAUSPROSESSIN NYKYTILANTEEN KUVAUS

Ohjelmiston testausprosessi ylläpitovaiheessa muodostuu kohdeyrityksessä kuvan 3 tavalla: kehitys, paketointi toiminnallista testausta varten, toiminnallinen testaus hyväksymistestinä, jota kutsutaan yrityksessä kenraaliksi, sekä viimeisestä vaiheesta, jossa ohjelmisto asennetaan tuotantoon. Toiminnallisessa testauksessa ilmi tulleet ongelmat palauttavat prosessin osittain takaisin kehitysvaiheeseen, jotta virheet voidaan korjata.



KUVA 3. Yritys X:n käyttämä testausprosessi

3.1 Kehitysvaihe

Varsinaisissa ohjelmistojen toteutuksissa käytetään testauslähtöistä ohjelmistokehitystä. Kehittäjä laatii siis yksikkötestit sisältävän ohjelmistokoodin sekä testaa jokaisen moduulin toiminnan yksinään, riippumatta muusta ympäristöstä. Kun jokainen moduuli toimii halutulla tavalla, on todennäköisempää, että ne toimivat myös yhdessä. Tämän varmistamiseksi kehittäjä toteuttaa integraatiotestejä, joilla varmistetaan moduulien toiminta ryhmissä sekä ryhmien välillä. Vastaavat kehitysvaiheet suoritetaan myös, kun ohjelmistoon lisätään muutoksia ylläpitovaiheessa.

Kehittäjä toimittaa yksikkötestit sisältävän ohjelmistokoodin versionhallintaan, josta jatkuvan integraation periaatteella toimiva ohjelmistotyökalu pääsee testaamaan ohjelmistoa erillisellä palvelimella. Testaus on automatisoitu sekä käynnistyy aina, kun lähdekoodi on muuttunut. Testauksessa koodi käännetään ja suoritetaan yksikkötestit sekä integraatiotestit. Epäonnistuneesta testistä lähetetään kehittäjälle viesti. Koska versionhallinta tallentaa kaikki versiot, pystytään palaamaan aikaisempaan versioon, mikäli tehdyt muutokset ovat ei-toivottuja. Versionhallinta mahdollistaa myös useamman ohjelmistokehittäjän kehittävän samaa ohjelmistomoduulia rinnakkain.

Koska ohjelmistotuotantoprosessissa on useita kehittäjiä, on ohjelmistokehityksen hallinnan kannalta tärkeää koota virhelistaa projektinhallintasovellukseen. Virhelistaan merkitään virheiden lisäksi niiden prioriteettiaste, mikä ohjaa korjaamisen kiireellisyyttä. Virhelistojen avulla kehittäjät pystyvät suuntaamaan resurssinsa mahdollisimman hyödyllisesti ohjelmiston kehityksen kannalta.

Kehittäjä suorittaa alustavaa kehitystestausta omalla koneellaan sekä logiikkatestausta selaimen avulla. Ohjelmistotestauksen kokonaistilannetta tutkitaan ohjelmistotyökaluilla, jotka antavat tietoa ohjelmiston testien kattavuudesta, virheraporteista sekä suoritetuista testeistä yhteenvedon muodossa.

Kehitysvaihe sisältää siis kehittäjän tekemän yksikkö- ja integraatiotestauksen sekä nämä testit automatisoituina ja jatkuvina. Näin tehdyt muutokset tulevat testatuiksi myös kehitysvaiheessa. Automatisoitu jatkuvan integraation menetelmä mahdollistaa ohjelmistovaatimuksien tarkastamisen jatkuvasti jolloin mahdolliset virheet havaitaan nopeasti, eivätkä ongelmat ilmene vasta integrointivaiheen ja projektin loppupäässä.

3.2 Toiminnallisen testauksen vaihe ja tuotantoasennus

Kun ohjelmistosta saadaan tehtyä kokonainen versio, se paketoidaan. Samassa yhteydessä ajetaan vielä automaattiset yksikkö- ja integraatiotestit. Mahdolliset virheet korjataan ja kun testit on läpäisty hyväksytysti, siirrytään toiminnalliseen

testaukseen. Testaus suoritetaan tuotantopalvelimen demopuolen ympäristössä, joka vastaa mahdollisimman hyvin todellista käyttöympäristöä. Tämä mahdollistaa myös tehokkuuden testauksen, jolloin voidaan esimerkiksi tutkia ohjelmistoon tehtävien tiedonhakujen kestoa.

Toiminnallinen testaus on manuaalista testausta, jossa testataan ohjelman toimintaa ja tutkitaan tallennusten onnistumista. Toiminnallisen testauksen vaiheita on useita ja löydetyt virheet palauttavat prosessin osittain takaisin kehitysvaiheeseen, jossa kehittäjä korjaa virheet, suorittaa yksikkö- ja integraatiotestit ja päivittää versionhallinnan.

Versionhallintaan tulleet muutokset johtavat automaattiseen yksikkö- ja integraatiotestaukseen. Seuraavassa toiminnallisen testauksen vaiheessa suoritetaan virheen paljastanut testitapaus uudelleen sekä ne järjestelmän osat, joihin virheen korjaus on voinut mahdollisesti vaikuttaa, ja varmistetaan, etteivät tehdyt muutokset ole aiheuttaneet uusia virheitä eli suoritetaan regressiotestaus.

Kun toiminnallisen testauksen vaiheessa testeistä saadaan halutut lopputulokset ja ne ovat suoritettuina hyväksytysti, ohjelmisto siirretään tuotantopuolelle tuotantopalvelimelle. Myöhemmin tehtävät päivitykset suoritetaan ensin myös tuotantopalvelimen demopuolelle, jotta voidaan varmistua muutoksien toimivuudesta.

4 TESTAUSPROSESSIN AUTOMATISOINTI

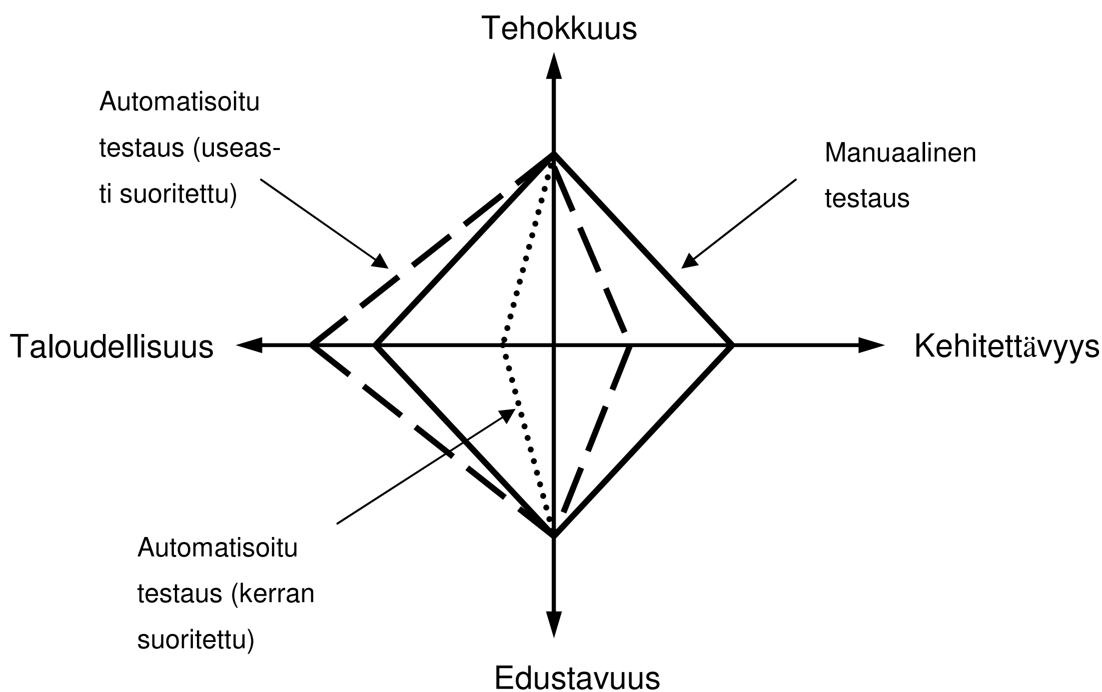
Kun manuaalinen testaus koneellistetaan, puhutaan testauksen automatisoinnista. Tämä edellyttää jo olemassa olevaa toimivaa testausjärjestelmää. Automatisoinnille asetetaan yleensä korkeat tavoitteet, joihin pyritään pääsemään erilaisten työvälineiden avulla. Automatisointi voi olla silti monimutkainen ja kallis prosessi, jossa ihminen on edelleen koko testausjärjestelmän tärkein osa.

Automatisointi voi säästää manuaaliseen testaukseen verrattuna kustannuksissa jopa 80 %. Yleensä automatisointi myös parantaa ohjelmistojen laatua, vaikka suoraa taloudellista hyötyä ei olisikaan saavutettu. (Fewster – Graham 1999, 3.)

4.1 Testitapausten hyvyys

Jotta vältytään automatisoinnin ongelmilta mahdollisimman hyvin, on tärkeää valita oikeat testitapaukset automatisoitaviksi. Hyvä eli laadukas testitapaus voidaan kuvata Keviatin kaavion neljän attribuutin avulla (kuva 4). Nämä neljä attribuuttia ovat tehokkuus, kehitettävyys, edustavuus ja taloudellisuus. (Fewster – Graham 1999, 4.)

Ehkä tärkeimpänä tehokkuus määrittää sen, kuinka paljon virheitä löydetään tai kuinka todennäköistä niiden löytäminen on. Edustava testitapaus vähentää tarvittavia testitapauksia, sillä sen tulee testata useampaa eri asiaa. Tehokkuus ja edustavuus eivät vaikuta kustannuksiin, sen sijaan taloudellisuus ja kehitettävyys vaikuttavat. Kehitettävä testitapaus on helppo ylläpitää eikä vaadi kohtuuttomasti resursseja aina kun ohjelmistoon tehdään muutoksia. Taloudellisuus tarkoittaa sitä, kuinka edullista testitapauksen suorittaminen, analysoiminen ja virheiden poistaminen on. (Fewster – Graham 1999, 4.)



KUVA 4. Keviatin kaavio (Fewster – Graham 1999, 5)

Keviatin kaaviossa yhtenäinen viiva kuvaa manuaalista testausta ja katkoviiva automatisoitua testausta. Testitapaus on sitä parempi, mitä pitempi attribuutin viiva on ja mitä suuremman alan yhtyvät viivat sulkevat sisäänsä. Näin ollen ensimmäisellä automatisoidulla testauskerralla ei saavuteta kovin suurta taloudellista hyötyä eikä se ole kehityskelpoinen. Vasta useasti suoritettuna automaation taloudellinen hyöty ja kehitettävyys kasvavat. (Fewster – Graham 1999, 5.)

4.2 Automatisoinnin tavoitteet ja hyödyt

Automatisoinnille täytyy asettaa selkeät tavoitteet. Tavoitteita voivat olla testikattavuuden lisääminen, testikierrokseen käytettävän ajan vähentäminen ja manuaalisesti suoritettavan testauksen vähentäminen, jolloin säästetään myös kustannuksissa sekä saadaan luotettavat testitulokset. Lähtökohdat ovat väärät, jos tavoitteena on kaiken testauksen automatisointi, sillä se on mahdotonta.

Vapautuneet resurssit, kuten testaajien ajansäästö, kannattaa mahdollisesti siirtää vaativampiin testauskohteisiin. (Hendrickson 1998, 3–7.)

Jotta automatisoinnista saataisiin paras hyöty, täytyy selvittää, mitä kannattaa automatisoida. Yksinkertaisin tapa on tutkia sitä, mihin testeihin käytetään eniten aikaa. Testitapauksissa, jotka suoritetaan usein, on yleensä jo mahdollisimman toimiva ja luotettava rakenne ja niiden automatisointi ajan säästön kannalta on järkevää. Vähän manuaalisesti läpikäydyt testitapaukset eivät välttämättä ole järkeviä automatisoinnin kohteita, sillä niihin voi löytyä vielä tehokkaampia menetelmiä. Testien täytyy myös olla tärkeitä, jotta automatisointiin kuluva aika kannattaa käyttää. (Pettichord 2001.)

Jos testitapaus suoritetaan vain kerran tai harvoin, on manuaalinen testaus järkevämpää resurssien kannalta. Regressiotestaus ja kuormitustestaus ovat hyviä automatisoinnin kohteita, sillä ne yleensä ajetaan useasti lähes muuttumattomina. Myös automatisoiduilla testeillä on elinkaari, jonka aikana niiden tulee maksaa itsensä takaisin. Täytyy siis tietää, millainen testi on hyödyllinen pitkään ja mitkä tapahtumat johtavat sen hylkäämiseen. Testin, joka kannattaa automatisoida, tulee olla myös hyvä löytämään virheitä, etenkin jos se ei kestä kovin monta ohjelmistokoodin muutosta. (Marick, 1–5.)

Automatisoinnin hyötyjä voivat olla tuloksien saaminen nopeammin ja edullisemmin eli aikatauluun ja budjettiin vaikuttaminen tai testausmahdollisuuksien laajentaminen. Hyvä automaatio vähentää resursseja lyhentämällä kehityksen ja testauksen läpimenoaikoja, pienentämällä kehityksen ja testauksen kustannuksia, parantamalla työmäärien hallintaa testikierroksen aikana, antamalla etenemiseen näkyvyyttä ja mahdollistamalla kyvyn muuttaa suunnitelmia markkinatarpeisiin vastaamiseksi. (Pyhäjärvi – Pöyhönen 2004, 16.)

Onnistunut automatisointi laajentaa testausmahdollisuuksia ja lisää luotettavuutta: testauksen julkaisuista saadaan nopeammin palaute, testejä voidaan mahdollisesti yhdistellä eri kombinaatioin tai suorittaa yhtäaikaisesti, uusintatestaus ja kyky toistaa löytyneitä virheitä sekä testien suuntaaminen riskialueille helpottuvat. Lisäksi testien suorittaminen valvomatta mahdollistaa keskittymisen moti-

voivampaan työhön rutiinien sijasta. Esimerkiksi kuormitustestaus manuaalisesti voi olla hidasta ja raskasta, mutta automatisointi mahdollistaa testien ottamisen käyttöön laajemmin todellisessa verkkoympäristössä. (Pyhäjärvi – Pöyhönen 2004, 16.)

4.3 Automatisoinnin haasteet

Automatisoiminen on kallis projekti, jonka riskinä ovat toteutumattomat hyödyt. Projektin tavoitteiden tulisi olla mahdollisimman realistiset. Automatisointi ei takaa virheettömyyttä eikä ratkaise kaikkia nykyisiä testausprosessin ongelmia. Automatisoiminen vie aikaa etenkin aluksi, jolloin joudutaan valitsemaan sopiva menetelmä automatisoimisen toteuttamiseksi, esimerkiksi uusi työkalu, ja opettelemaan sen käyttö. Kustannukset kasvavat myös silloin, kun merkittävät virheet löytyvät ohjelmistokehityksen loppupäästä ja aikaa niiden korjaamiseen on vähemmän. (Pyhäjärvi – Pöyhönen 2004, 17, 35.)

Ennen automatisointia täytyy tietyt asiat olla jo olemassa nykyisessä testausjärjestelmässä, jotta automatisoinnista saadaan mahdollisimman tehokas. Testausjärjestelmän tulee sisältää vähintään suunnitteludokumentteihin ja vaatimusmäärittelyihin perustuvat yksityiskohtaiset testitapaukset odotettuine tuloksineen. Lisäksi järjestelmään tulee kuulua erillinen testausympäristö testitietokantoineen, jonka tulee olla tallennettavissa uudelleen vakiomuodossa, kun sovellukseen tehdään muutoksia. Tämä mahdollistaa testitapausten uudelleen suorittamisen. (Zambelich, 4.)

Jotta automatisoimisprojekti voisi onnistua, täytyy koko organisaation tukea sitä, erityisesti johdon. Kommunikaation tulee säilyä jatkuvasti organisaation sisällä, jotta tiedetään, mihin rahat ollaan käyttämässä ja mitä hyötyjä ollaan saavuttamassa. Hyötyjen saavuttaminen on tosin hankalaa, mikäli ei pystytä selvittämään hyvin, mitä kannattaa automatisoida ja mitkä ovat realistisia tavoitteita. Lisäksi tulee muistaa, ettei kaikkea voi automatisoida. Tulisi siis varata riittävästi aikaa nykyisen testausprosessin mahdolliseen kehittämiseen sellaisenaan ja

tavoitella tuotteen helpompaa testattavuutta sekä varata resurssit järkevien automatisoimiskohteiden selvittämiseen. (Hendrickson 1998, 26–29.)

Automatisoiminen on ohjelmointia, mikä täytyy huomioida, sillä aina testaajilla ei ole takana vahvaa ohjelmointitaitoa. Vaikka automatisoinnin toteuttamiseksi valittu työkalu vaikuttaisi helpolta käyttää, tulee testaajan silti ymmärtää, miten ohjelmisto suunnitellaan, kehitetään ja ylläpidetään. Automaattisen testausohjelman löytämät virheet täytyy myös työntekijän osata tulkita oikein. Automatisointiprojektiin tulee siis valita riittävällä ammattitaidolla valitut henkilöt, joita muu organisaatio tukee, sillä myös automatisoinnin tuloksena syntyneet testaustyökalut joudutaan testaamaan. (Pettichord 2001.)

Automatisoinnissa tulee myös muistaa, ettei se löydä paljon uusia virheitä, vaan ne löydetään ensimmäisellä kierroksella testauksessa. Uudet virheet syntyvät yleensä silloin, kun ohjelmaa on muutettu tai se ajetaan toisessa ympäristössä. Tällöin saatetaan joutua uusimaan myös testit, mikä tekee ylläpidosta raskasta. Väärä turvallisuuden tunne on siis vaarallista, sillä testit voivat itsessään olla puutteellisia ja odotetut tulokset vääriä. (Pettichord 2001.)

4.4 Automatisoinnin ylläpidettävyys

4.4.1 Ongelmat

Vaikka automatisoiminen saataisiin onnistumaan, voi ylläpidettävyys osoittautua haasteeksi, kun uusia ohjelmistoversioita ollaan ottamassa käyttöön. Usein testit vaativat myös päivityksen ennen uudelleen käyttöä. Nykyisistä testeistä on voinut tulla turhia tai ne korvautuvat uusilla testeillä. Tällöin ylläpitokustannukset kasvavat, kun joudutaan tekemään muutoksia. (Fewster – Graham 1999, 191.)

Automatisoinnin yhteydessä testien lukumäärä ja niiden sisältö voivat kasvaa nopeasti. Tämä lisää jälleen työmäärää ohjelmiston muutosten yhteydessä sekä testien suoritusaikaa. Kun työmäärä kasvaa, on helppo unohtaa testauksen do-

kumentointi. Testausta tulisi käsitellä kuten ohjelmistokehitystä siitä huolimatta, että käytössä on automatisointityökalu suorittamassa testauksen. (Fewster – Graham 1999, 191–198.)

4.4.2 Ratkaisuja

Automatisoinnissa testien ylläpidettävyyttä voidaan helpottaa. On tärkeää valita tarkkaan, millaiset testit automatisoidaan. Testien lukumäärää on syytä tarkkailla ja rajata, samoin niiden kokoa. Nämä vaikuttavat testien suoritusaikaan ja resursseihin, joita käytetään testien päivitykseen ohjelmistomuutosten yhteydessä. Muutosten jälkeen osa testeistä saattaa olla turhia ja uusia joudutaan lisäämään. Täytyy löytää sopiva tasapaino testien määrään varmentamaan testauksen luotettavuus ja huolehtia silti testauksen ylläpidettävyyden onnistumisesta. Lisäksi kannattaa välttää liian monimutkaisten testien automatisoiminen, sillä niiden ylläpidettävyys on työlästä. (Fewster – Graham 1999, 191–193.)

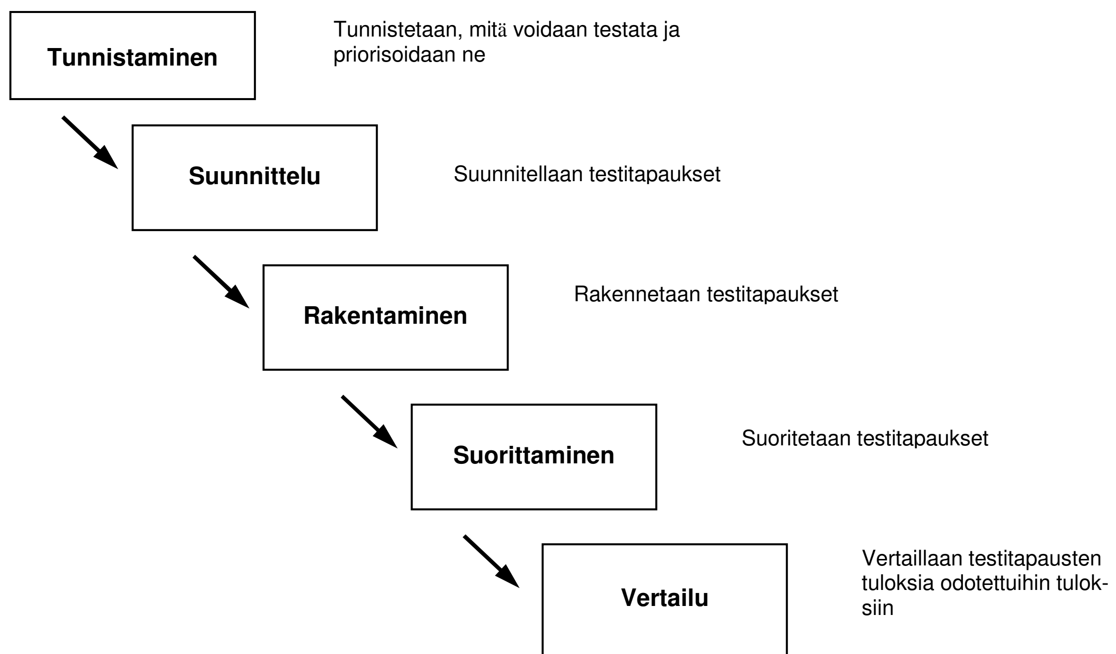
Testitapausten tallentaminen tekstimuotoon sallii paremman joustavuuden eri alustojen ja järjestelmien välillä. Vaikka kaiken tiedon konvertoiminen tekstimuotoon vie myös aikaa, ylläpito helpottuu, kun käsitellään vain yhtä tietomuotoa. Lisäksi on tärkeää tutkia testitapausten riippuvuutta toisistaan. Pitkät ketjut, joissa edellisestä testistä saatu tulos on seuraavan testin käyttämä lähtötieto, ovat vaarallisia. Mikäli virhe tapahtuu heti alussa, eivät seuraavat testit pysty suoriutumaan oikein. Vaikka ketjuttaminen oikein toimiessaan on tehokas työkalu, on se syytä aloittaa varovaisesti ja laajentaa maltillisesti. (Fewster – Graham 1999, 196–197.)

Hyvä dokumentointi ja järkevä tiedostojen nimeäminen ovat myös tärkeitä asioita ottaa huomioon. Etenkin usean henkilön tehdessä työtä samassa projektissa yhteneväiset nimeämiskäytännöt helpottavat ja nopeuttavat työtä sekä vähentävät virheiden määrää. Laadukas dokumentointi on myös tärkeää, jotta testeistä voidaan saada aina vähintään yleiskuva eikä kenenkään tarvitse työskennellä dokumentoimattomien testien kanssa, siitä huolimatta että testityökalu suorittaa testauksen. (Fewster – Graham 1999, 197–199.)

5 AUTOMATISOINNIN KOHTEET

5.1 Automatisoitavat toiminnot

Testitapausten kehityksen elinkaari muodostuu kuvan 5 perusteella viidestä eri toiminnosta. Eri organisaatioissa toiminnot suoritetaan eri tavalla, mutta ihannetapauksessa jokainen testitapaus käy läpi tunnistamisen, huolellisen suunnittelun, varsinaisen rakentamisen ennen suorittamista ja lopulta saatujen tulosten vertailun odotettuihin tuloksiin. (Fewster – Graham 1999, 14–15.)



KUVA 5. Testitapausten kehityksen elinkaari (Fewster – Graham 1999, 13)

Kaksi ensimmäistä toimintoa, eli tunnistaminen ja suunnittelu, ovat luonteeltaan älykkyyttä vaativia toimintoja ja määräävät testitapausten laadun. Kaksi viimeistä, testitapausten suorittaminen ja vertailu, ovat luonteeltaan verrattavissa toimistotyöhön, ne suoritetaan useita kertoja ja soveltuvat siten hyvin automatisoinnin kohteiksi. (Fewster – Graham 1999, 17–19.)

5.2 Suunnittelun automatisoiminen

Testitapausten tunnistamisen ja suunnittelun automatisoimista on syytä harkita tarkkaan, sillä ne suoritetaan yleensä vain kerran. On silti olemassa lukuisia testaustyökaluja testitapausten suunnittelun automatisoimiseksi. Ongelmaksi työkalua käytettäessä voi muodostua suuri testien lukumäärä, eikä työkalu osaa määritellä, mitkä testit ovat kaikista tärkeimpiä ajan säästämiseksi. Testitapausten generointivälineet ovat tosin tarkempia ja täsmällisempiä kuin ihminen käytettäessä samoja algoritmeja. (Fewster – Graham 1999, 17–19.)

Testitapausten generointivälineet voivat pohjautua koodiin, rajapintoihin tai määrittelyihin. Koodiin pohjautuva työkalu generoi testisyötteet tutkimalla ohjelmistokoodin rakennetta. Koodi muodostaa polun, joka koostuu lohkoista, joita määrittävät ehdot. Työkalu pystyy määrittämään jokaisen polkulohkon vaatimat loogiset ehdot automaattisesti ja näin saadaan samalla tietoa myös kattavuuden mittaamiseen. (Fewster – Graham 1999, 19–20.)

Koodiin pohjautuvan generointityökalun käytössä täytyy huomioida se, ettei se sisällä testioraakkelia eli lähdettä oikeista odotetuista testien tuloksista. Näin ollen ei voida vertailla varsinaisen testin tavoin, olivatko saadut tulokset oikeita. Lisäksi tässä menetelmässä ongelmana voivat olla puuttuvat koodit, joita ei pystytä havaitsemaan. (Fewster – Graham 1999, 19–20.)

Rajapintoihin pohjautuva generointityökalu voi olla hyödyllinen verkkosovellusten ja graafisten rajapintojen testaamisessa. Työkalu pystyy esimerkiksi testaamaan näytön painikkeet ja linkit aktivoimalla ne. Lisäksi pystytään testaamaan, että sovellus sisältää kaikki halutut asiat. Virheet pystytään ainakin osittain paikantamaan. (Fewster – Graham 1999, 20–21.)

Määrittelypohjaisten testitapausten generoinnissa saadaan myös odotetut tulokset, kunhan määrittelyt ovat sellaisessa muodossa, että työkalu voi niitä tulkita. Määritelmä voi sisältää esimerkiksi teknistä dataa, kuten tiloja ja siirtymisiä. Hyötynä saavutetaan se, että testi todellakin tutkii, mitä ohjelmiston pitäisi teh-

dä. Odotetut tulokset voidaan generoida, jos ne on sisällytetty määritelmiin ja tulokset ovat oikeita. (Fewster – Graham 1999, 21.)

Markkinoilla on useita suunnittelun apuvälineitä, esimerkiksi määrittelyihin perustuva Aonixin Validator/Req, joka generoi testit suoraan määrittelyistä (Product Overwiew.1998, 2). IMI Systems Inc. yhtiön RadSTAR-työkalu on malleihin perustuva yhdistelmä testien suunnittelusta ja automaattisesta koneesta, joka suorittaa testitapaukset. Testitapausten toistaminen voi olla myös apuna regressiotestaukseen liittyvässä automatisoinnissa. (Fewster – Graham 1999, 476, 486–487.)

Vaikka päädyttäisiin automatisoimaan suunnittelu, on syytä muistaa, että ihmistä tarvitaan edelleen priorisoimaan testit ja suunnittelemaan ne testit, joita työkalut eivät pysty laatimaan. Lisäksi kaikki edellä mainitut menetelmät voivat generoida liikaa testejä ja vaikuttavat näin ajankäyttöön ja ylläpidettävyyteen. Parhaimmassa tapauksessa hyödyt voivat silti olla vaivan arvoisia. Esimerkiksi rajapintoihin pohjautuvassa generointityökalussa voidaan saada kattavampi tarkastelu aikaan verrattaessa verkkosovelluksen manuaaliseen läpikäymiseen painike ja manuaali kerrallaan. (Fewster – Graham 1999, 21–22.)

5.3 Suorituksen automatisointi

Suunnittelun automatisoimisen lisäksi voidaan automatisoida testien rakentamista, suorittamista tai testien lopputulosten vertailua odotettuihin tuloksiin. Näiden automatisoimiseen vaikuttavat käytetyt työkalut ja menetelmät, työntekijöiden taidot, automatisoitavan ohjelmiston vaatimukset esimerkiksi ympäristön suhteen ja itse ohjelmisto. Koska vaikuttavia tekijöitä on useita, voidaan arvioida automatisoimiseen kuluvan viisi kertaa enemmän aikaa kuin manuaaliseen testaukseen (Fewster – Graham 1999, 34–35.)

Suorituksen automatisoimiseksi ei välttämättä tarvita työkalua. Etenkin ohjelmat, jotka eivät ole suoraan käyttäjän kanssa tekemisissä, ovat helpompia automatisoida. Yksittäinen komentotiedosto voidaan ohjelmoida käynnistämään

ohjelma ja suorittamaan testitulosten vertailun. Luonnollisesti pelkkä testien käynnistäminen ei tarjoa samoja mahdollisuuksia kuin työkalu. (Fewster – Graham 1999, 42.)

5.3.1 Testisyötteen nauhoittaminen

Testisyötteet voidaan nauhoittaa työkalun avulla. Työkalu kirjoittaa testaajan suorittamista toimenpiteistä skriptin eli komentosarjan, joka sisältää testisyötteet ja hiiren liikkeet. Skriptin avulla testi voidaan toistaa yhä uudelleen tai sitä voidaan muokata muokkaamalla itse skriptiä, joka on omana tiedostona. (Fewster – Graham 1999, 43.)

Nauhoitetun testin etuna on sen nopeus. Testaus on helppo aloittaa, mutta vaarana on huonosti suunnitellut tai kokonaan suunnittelematon testaus, jolloin ei välttämättä tuoteta hyödyllisimpiä testejä. Nopeuden lisäksi hyötynä saavutetaan automaattisesti tuotettu dokumentaatio, josta selviää suoritetut testit. Päivitysten yhteydessä nauhoituksesta on hyötyä, mikäli sama muutos tehdään usealle eri tiedostolle. (Fewster – Graham 1999, 45–46.)

5.3.2 Nauhoittamisen ongelmat

Nauhoittaminen on hyvä alku automatisoinnille, mutta ei välttämättä ole hyvä pohja lopulliselle ja pitkäaikaiselle automaatiolle. Nauhoittamalla saadut skriptit eivät ole välttämättä kovin luettavia jälkikäteen ja niihin täytyisi sisällyttää tietoa testin kohteesta ja sen tarkoituksesta. Lisäksi ohjelmistomuutosten jälkeen alkuperäinen skripti ei sovellu enää käyttöön ja joudutaan tekemään uusia nauhoituksia, mikä vaikuttaa kustannuksiin. (Fewster – Graham 1999, 46–47.)

Nauhoittamalla saatujen testien tarkastus on työlästä ilman tulosten vertailua, sillä testaaja saattaa joutua seuraamaan ruudulta toistetun testin ja varmistamaan, että testitapaus toimi odotetulla tavalla. Nauhoittamiseen tulisi siis lisätä myös vertailu, jotta saataisiin aikaan varsinainen testi eikä pelkkää syötteiden

automatisoimista. Pelkkä testausprosessin parantaminen muilla keinoilla voi tuoda paremman tehokkuuden kuin pelkkä nauhoittamisen käyttäminen. (Fewster – Graham 1999, 46–48.)

5.4 Vertailun automatisointi

Vertailun avulla tarkistetaan, tuottaako ohjelmisto halutut tulokset. Testit voivat vaatia yhden tai useamman vertailun odotettujen ja todellisten tulosten välillä, esimerkiksi silloin kun halutaan varmistua uuden informaation oikeellisuudesta niin näytöllä kuin tietokannassakin. Vertailun automatisoiminen on järkevää, sillä ihminen tekee helposti virheitä verratessaan silmämääräisesti pitkiä listoja tuloksista sekä kokee työn helposti tylsänä ja hitaana. (Fewster – Graham 1999, 101–103.)

Vertailun automatisoimiseen tarvitaan yleensä työkalu. Työkalun ominaisuuksista riippuu se, millaisia tiedostomuotoja voidaan vertailla. Työkalu suorittaa vertailun kahden tiedoston välillä etsien niistä eroavaisuuksia ja ilmoittaa löydöt käyttäjälle. Kehittyneempi työkalu pystyy antamaan lisätietoa löydetyistä eroavaisuuksista esimerkiksi korostamalla värillä huomioitavat kohdat. Lisäksi saadaan muuta tietoa itse vertailusta, kuten suorituksen kesto ja päivämäärä. (Fewster – Graham 1999, 105–106.)

Vertailua voidaan suorittaa testaamisen aikana eli suorittaa dynaaminen vertailu tai tehdä vertailu suorituksen jälkeen sekä käyttää molempia vertailutapoja yhdessä. Dynaamisessa vertailussa ohjeet vertailuun ovat testiskriptissä. Ohjeet kertovat, mitä ja milloin verrataan sekä mihin verrataan. Vertailu voidaan yleensä pysäyttää kesken suorituksen tarkempaa tarkastelua varten ja lisätä saatu tulos odotetuksi lopputulokseksi. Testiskriptejä voidaan myös muokata manuaalisesti, mikä luonnollisesti edellyttää ohjelmointitaitoja, mutta siten voidaan lisätä testitapauksiin älykkyyttä. Voidaan esimerkiksi ohjelmoida vertailun lopettaminen tietynlaisen virheen esiintyessä, mikäli jatkaminen olisi turhaa. (Fewster – Graham 1999, 107–108.)

Suorituksen jälkeisessä vertailussa on yleensä kyse uusien tiedostojen tarkastamisesta sekä tietokannan sisällön oikeellisuudesta päivityksen yhteydessä. Testien suorittamiseen käytettävät työkalut harvoin tukevat testauksen jälkeistä vertailua, mutta testitapaukset on mahdollista suunnitella niin, että tulokset tulostetaan näytölle dynaamista vertailua varten. Suorituksen jälkeinen vertailu on kumminkin työläämpää automatisoida, joten sen tarpeellisuus kannattaa arvioida tarkasti sekä jakaa tulosten tarkistusta ryhmiin. Esimerkiksi tarkastetaan tuloksia tarkemmin, mikäli ensimmäinen suurpiirteisempi tarkistus antaa aihetta epäillä virheitä. (Fewster – Graham 1999, 108–110.)

Vaikka vertailu on hyvä automatisoinnin kohde, täytyy silti muistaa, että myös siihen liittyy haasteita. Dynaamisessa vertailussa testiskriptien muokkaus tekee niistä yhä monimutkaisempia ja hankalammin ylläpidettäviä, mikä vaikuttaa resursseihin. Yksinkertaisessa vertailussa puolestaan virheilmoituksia voivat aiheuttaa jo esimerkiksi eri päivämäärä, joten vertailun suorittavalta välineeltä vaaditaan myös älykkyyttä. Vertailun hyödyllisyys riippuu myös siitä, miten hyvin odotetut tulokset ovat osattu määritellä ja sisältävätkö ne jo virheitä. (Fewster – Graham 1999, 108.)

6 TESTAUKSEN APUVÄLINEET

Testausvälineet voidaan ryhmitellä testauksen tukemisen välineisiin ja testiautomaatiovälineisiin. Testausta tukevia välineitä ovat esimerkiksi suunnittelua ja hallinnointia helpottavat välineet, testitapausten suunnittelu, testiympäristön alustaminen sekä testisuorituksen seuranta ja hallinta, eli kaikki sellaiset välineet jotka tukevat sekä käsin suoritettavia ja automatisoituja testejä ja jotka jäävät varsinaisen automatisoinnin ulkopuolelle. (Pyhäjärvi – Pöyhönen 2004, 7–10.)

Varsinaisilla testiautomaatiovälineillä suoritetaan testausta tietokoneavusteisesti. Apuvälineitä on esimerkiksi testitapausten generointiin, toiminnalliseen testaukseen, tulosten vertailemiseen ja suorituskykytestaukseen. (Pyhäjärvi – Pöyhönen 2004, 7–10.)

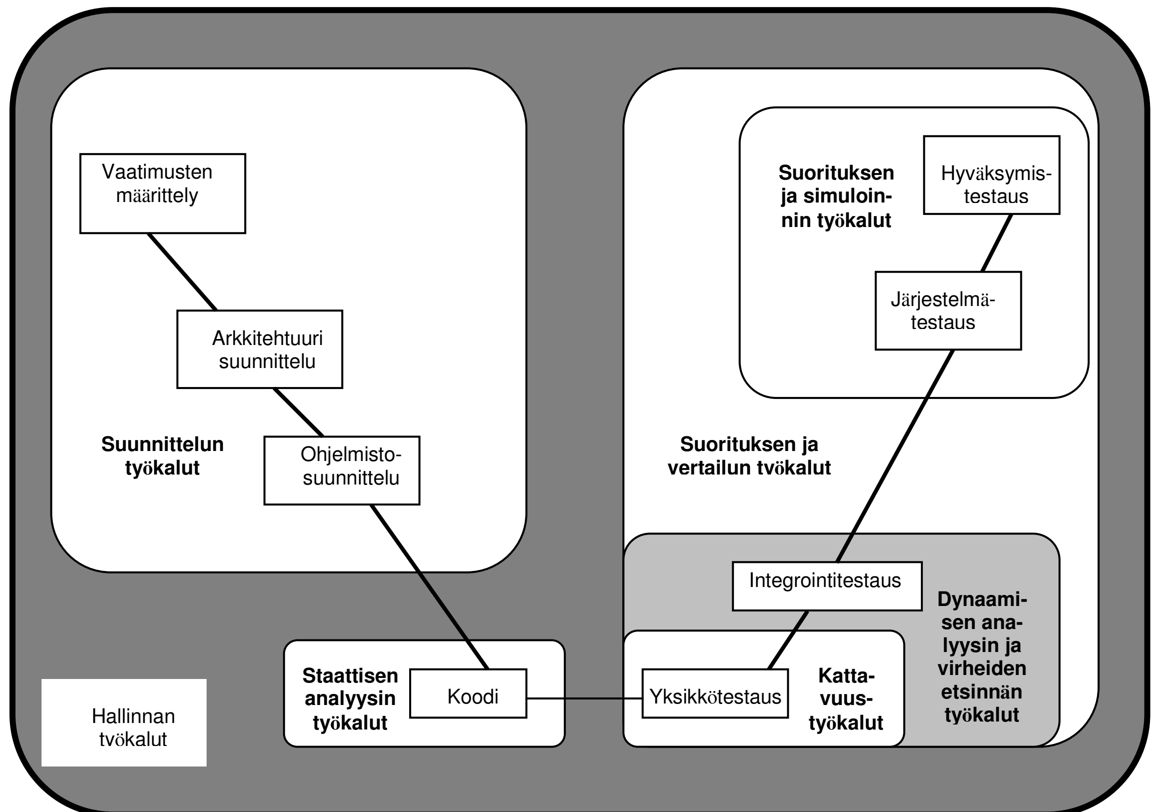
6.1 Apuvälineet ohjelmistokehityksen elinkaaren aikana

Jokaiselle ohjelmistokehityksen elinkaaren tasolle on valittavissa oma apuväline. Kuvassa 6 on esitelty apuvälineiden sijoittuminen eri ohjelmistokehityksen vaiheelle. (Fewster – Graham 1999, 7.)

Testauksen hallinnan apuvälineet ovat mukana jokaisella elinkaaren tasolla. Suunnittelun apuvälineet ovat käytössä vaatimusten määrittelyssä ja arkkitehtuuri- sekä ohjelmistosuunnittelussa. Ne auttavat määrittelemään ja suunnittelemaan tarvittavat testit sekä suorittamaan testitapausten generoinnin. (Fewster – Graham 1999, 8.)

Staattisen analyysin työkalut analysoivat koodia sitä ajamatta. Apuvälineillä pystytään määrittämään muun muassa koodin kompleksisuutta ja analysoimaan ohjelman rakennetta. CMTJava on apuväline mittaamaan koodin kompleksisuutta, kun ohjelmointikielenä on Java. CMTJava pystyy määrittämään suuresta

koodin määrästä todennäköisimmät ongelmakohdat sekä mittaamaan kuinka paljon koodia on olemassa ja kirjoittamaan raportit HTML-muotoon. (Testwell CMTJava. 2009.)



KUVA 6. Apuvälineet ohjelmistokehityksen elinkaaren eri tasoilla (Fewster – Graham 1999, 7.)

Koodin kattavuuden apuvälineet toimivat yleensä yksikkötestauksen kanssa samalla tasolla. Dynaamisen analyysin ja virheiden etsinnän apuvälineet arvioivat ohjelmistoa sen suorituksen aikana. (Fewster – Graham 1999, 8.)

Ohjelmiston testauksen suoritukseen on tarjolla useita apuvälineitä. Java-ohjelmointikielelle on esimerkiksi TestComplete 7 -apuväline, joka on automatisoitu työkalu testien laadintaan, hallintaan ja suoritukseen. Apuvälineellä voidaan automatisoida myös regressiotestausta ja se soveltuu projekteihin, jossa on käytössä testivetoinen ohjelmistokehitys. (TestComplete 7. 2010.)

Automatisoinnin kannalta on lisäksi tärkeää löytää apuvälineet suorittamaan tulosten vertailua sekä järjestelmä- ja hyväksymistestauksen vaatimat kuormitus- ja suorituskykytestit. Yksi tällainen apuväline, joka kattaa useamman eri testauksen alueen, testienkattavuudesta kuormitustestaukseen ja automatisoituun regressiotestaukseen tulosten vertailuineen, on AppPerfect (Automated testing. 2010). Usein kaupalliset työkalut myydäänkin laajoina paketteina, jolloin ne saattavat sisältää myös ohjelmistoprojektille tarpeettomia ja jo olemassa olevia ominaisuuksia.

6.2 Apuvälineen valinta testauksen automatisoinnissa

Apuvälineen valinta on projekti, jolle täytyy varata aikaa ja resursseja. Tärkein ja ensimmäinen tehtävä on kartoittaa ne vaatimukset, jotka apuvälineen tulisi täyttää. Kun asiat, jotka halutaan automatisoida, esimerkiksi regressiotestaus, manuaalisesti suoritettava testaus ja testauksen tehostaminen, on määritelty, tulisi vielä tarkastella, voitaisiinko testausta parantaa muilla menetelmillä kuin apuvälineellä. Voidaan esimerkiksi miettiä, ovatko kaikki manuaalisesti tehtävät testit ylipäänsä tarpeellisia, jolloin säästettäisiin aikaa ja vaivaa, tai voitaisiinko työntekijöitä lisätä testaukseen. Ja mikäli regressiotestaus vie liikaa aikaa, tulisi miettiä, tapahtuuko ohjelmistokehityksen loppupäässä liikaa muutoksia, jotka vaikuttavat regressiotestaukseen. (Fewster – Graham 1999, 252–254.)

Kun testausprosessi on saatu mahdollisimman tehokkaaksi ja automatisoinnille on löytynyt hyvä kohde, tulisi valita tavoitteet, jotka apuvälineen tulisi täyttää. Mitattavissa olevat tavoitteet auttavat myöhemmin arvioimaan, oliko apuvälineen käyttöönotto järkevää. Voidaan esimerkiksi määrittää, että kolmen kuukauden apuvälineen käytön jälkeen testitapauksista on automatisoitu 50–60 % ja työmäärää on pystytty vähentämään noin puoleen. (Fewster – Graham 1999, 256–257.)

Vaatimusten määrittelyn ja tavoitteiden lisäksi tulee laatia budjetti apuvälineen hankinnalle. Sopivat apuväline-ehdokkaat saadaan valittua vaatimusten avulla, joita voivat olla esimerkiksi apuvälineen toimiminen tietyillä alustoilla ja käyt-

töönoton koulutuksen kustannukset. Mikäli kaupallisilta markkinoilta ei löydy vaatimukset täyttävää apuvälinettä, kannattaa myös harkita sellaisen rakentamista itse. (Fewster – Graham 1999, 258–263.)

Kun rakennetaan apuväline itse, saadaan todennäköisesti paremmin vaatimukset täyttävä työkalu, mutta suuremmilla kustannuksilla. Ostetun apuvälineen tuki, koulutus ja dokumentointi ovat yleensä parempia. Rakentamalla itse saadaan tuskin sen joustavampaa apuvälinettä kuin kaupallinenkaan. Automatisoitavan kohteen tulee olla tarkasti määritelty, muuten on resurssien tuhlaamista alkaa niin rakentaa kuin ostaakaan apuvälinettä. (Fewster – Graham 1999, 267–268.)

6.3 Apuvälineen käyttöönotto

Kun apuväline on saatu valittua, on ensimmäinen tehtävä myydä se yrityksen sisälle eli saada kaikki työntekijät hyväksymään tulevat muutokset. Vähintään yhden henkilön tulisi tutustuttaa ja markkinoida apuväline muille työntekijöille ja helpottaa siirtymistä uusiin työskentelytapoihin. Tämä on tärkeää, jotta apuväline ei jää heti pois käytöstä. (Fewster – Graham 1999, 284–285.)

Apuvälineen käyttöönotto on kaiken kaikkiaan vastakkainen menetelmä sen valinnalle. Nyt laajennetaan apuvälineen käyttöönoton hyväksymistä yrityksen sisällä, saatavia hyötyjä sekä itse käyttöä. Käyttöönoton kustannukset ovat silti alussa suuret, sillä tarvitaan koulutusta, tukea sekä jonkinlainen pilottiprojekti, jotta voidaan nähdä apuvälineen vaikutukset testaukseen. Luonnollisesti ei tule unohtaa arvioida käyttöönoton jälkeen apuvälineestä saatavia hyötyjä. (Fewster – Graham 1999, 284–291.)

7 TESTAUKSEN AUTOMATISOINNIN APUVÄLINEET

Testauksen automatisointia tukevia apuvälineitä on markkinoilla runsaasti. Kaupallisten apuvälineiden lisäksi saatavilla on myös ilmaisia apuvälineitä. Ne eivät yleensä ole toiminnoiltaan yhtä kattavia kuin maksulliset, mutta niitä voidaan käyttää tiedettäessä tarkkaan, mitä halutaan automatisoida. Tähän valitut apuvälineet on valittu yritys X:n käyttöön soveltuviksi, jolloin on huomioitu esimerkiksi käytössä oleva Java-ohjelmointikieli.

7.1 Ilmaiset apuvälineet

7.1.1 MaxQ

MaxQ on työkalu verkkosovellusten testaamiseen. Nauhoittamiseen perustuva ilmainen työkalu on tehty vastineeksi maksullisille Astra QuickTest ja Empirix e-Test -apuvälineille. MaxQ tarjoaa http-testauksen nauhoittamisen, skriptauksen ja nauhoituksen toiston. (Open Source Software Testing Tools. 2009.)

MaxQ-työkalun käytölle vaatimuksina on Java 1.2 tai myöhempi versio. MaxQ toimii verkkoselaimen ja palvelimen välissä, jolloin selaimen pyynnöt ja palvelimen vastaukset kulkevat MaxQ:n kautta. MaxQ nauhoittaa Python-skriptit käyttäjän painaessa linkkiä. Kun skripti ajetaan, toimii MaxQ verkkosovelluksen asemasta. Tämä mahdollistaa regressiotestauksen verkkosovellukselle sekä tarkistuksen, että sovellus tuottaa oikeanlaista HTML-kieltä. (Open Source Software Testing Tools. 2009.)

MaxQ:n käyttö ei vaadi suuria ohjelmointitaitoja. Koska työkalu on kirjoitettu Javalla, voidaan sitä käyttää eri alustoilla ja sen muokkaaminen on helppoa. (Open Source Software Testing Tools. 2009.)

7.1.2 iValidator

iValidator on Javalla kirjoitettu kehys regressiotestaukseen sekä erityisesti monimutkaisiin testitapauksiin ja integraatiotestauksen automatisointiin. Vaatimuksena on Java-kääntäjä JDK 1.3. Työkalu sopii ketterän ohjelmistokehityksen menetelmään, jossa integraatiotestausta on yleensä vaikea automatisoida. (Test Automation.)

Testaus pohjautuu yksikkötesteihin. Testitapaukset ilmoitetaan XML-muodossa. Tarkka jako varsinaisiin testiluokkiin parametreista ja testin kulun määritelmä mahdollistavat joustavuuden. Kaiken kaikkiaan iValidator tukee kaikkia testauksen tasoja: kehitys-, integraatio- ja hyväksymistestejä. (Test Automation.)

7.1.3 Concordion

Concordion on työkalu hyväksymistestien automatisointiin. Se integroituu suoraan JUnitin kanssa. Työkalu muuttaa englanninkielisen kuvauksen vaatimuksista automaattiseksi testiksi. Dokumentaatio on helppoa helposti luettavien testien ansiosta sekä työkalu auttaa dokumentoimaan systeemin logiikan ja käyttäytymisen. (Peterson 2009a.)

Concordionin spesifikaatiot ovat aina aktiivisia, jolloin ne ovat linkitettyinä testeihin ja ovat aina ajan tasalla. Kun on tehty muutoksia systeemin käyttäytymiseen ja testit osoittavat vääriin määrittelyihin, saadaan siitä ilmoitus. Käyttöönotto vaatii Javan version 1.5 tai uudemman. (Peterson 2009a.)

7.1.4 Cucumber

Cucumber on työkalu, joka suorittaa tavallisella tekstillä kirjoitettuja toiminnallisia testejä. Työkalu toimii useilla eri kielillä, esimerkiksi Javalla ja Rubylla sekä millä tahansa kielellä kirjoitetulla verkkosovelluksella. (Cucumber Behaviour Driven Development with elegance and joy. 2009.)

Testien ajatus on yksinkertainen: 1. määritellään jokin ohjelmiston käyttäytymisen osa pelkän tekstin avulla, 2. kirjoitetaan ohjelmointikielellä yksi määritelmän askel, 3. ajetaan testi ja nähdään sen epäonnistuvan, 4, kirjoitetaan koodia jotta askel läpäistään, 5. ajetaan testi ja läpäistään se ja 6. toistetaan askelia 2.–5., kunnes koko määritelty skenaario on suoritettu ja testattu. Skenaario nähdään koko ajan kirjoitetussa muodossa ruudulla ja tekstin värin muutos ilmaisee edistymisen. Kirjoitettu teksti palvelee dokumentaationa, automatisoituina testeinä ja kehityskehyksenä, kaikki samassa formaatissa. (Cucumber Behaviour Driven Development with elegance and joy. 2009.)

7.2 Maksulliset apuvälineet

7.2.1 SilkTest

Borlandin SilkTest on työkalu testauksen automatisointiin. Toiminnallinen testaus sekä regressiotestaus saadaan tehokkaammaksi ja helpommin ylläpidettäväksi, jolloin myös virheet löytyvät testausprosessin aikaisessa vaiheessa, jolloin säästetään kustannuksissa. Erityisesti Java-testeille on kehitetty Eclipse-liitännäinen Silk4J. Apuväline keskittyy testien luontiin ja nauhoittamiseen sekä niiden ajamiseen. (SilkTest 2010. 2010.)

SilkTest-työkalun avulla pyritään saavuttamaan suurin hyöty säästämällä aikaa, kun testit ovat ajettavissa milloin tahansa, testit ovat helposti ylläpidettäviä ja käytettävissä uudelleen sekä testien kattavuus on hyvä ja tulosten analysointi on täsmällistä ja nopeaa. Ympäri vuorokauden suoritettavan testauksen mahdollistaa työkalun itsenäinen toiminta virhetilanteissa. Sovellus, jossa virhe esiintyy, kirjataan ja palautetaan alkuperäiseen tilaan ja jatketaan testauksen suoritusta seuraavista testeistä. Näin testaus ei jumiudu jokaisessa virhetilanteessa eikä jää odottamaan testaajan reagointia. (Data Sheet Silk Test. 2010, 1.)

SilkTest tukee Java-ohjelmointikieltä (versioista Sun JVM 1.5 tai 1.6) sekä useita selaimia ilman skriptin muokkaustarvetta. Alustaksi sopivat yleisimmät Windows-käyttöjärjestelmät. (Data Sheet Silk Test. 2010, 2–3.)

7.2.2 HP QuickTest Professional Software

Regressiotestauksen ja toiminnallisten testien automatisointiin on kehitetty maksullinen HP QuickTest Professional Software -työkalu. Työkalu auttaa luomaan testitapauksia ilman, että käyttäjät tarvitsevat laajaa koulutusta, turvaamaan oikeanlaisen toiminnallisuuden kaikissa ympäristöissä ja projekteissa, hyvän dokumentoinnin, helpon regressiotestauksen ja paremman tuottavuuden sekä laadun. Työkalu soveltuu tuottamaan testit yleisimmille ohjelmistoille ja ympäristöille, joita ovat esimerkiksi Windows, NET ja J2EE. (HP QuickTest Professional software Data sheet. 2007, 1.)

Testejä voidaan tehdä, hallinnoida ja muokata ilman skriptien kirjoittamista graafisten näkymien avulla tai niitä voidaan nauhoittaa. Työkalun muodostamia skriptejä voidaan silti helposti muokata, lisätä ja poistaa. Testiskriptit voidaan julkaista, jolloin muut testaajat voivat ottaa ne käyttöön ja päällekkäisyydet eliminoituvat. Avoimeen XML-formaattiin perustuva työkalu mahdollistaa eri testaajaryhmien työskentelyn yhdessä sekä tietojen jakamisen, jolloin tehdyt muutokset myös päivittyvät. (HP QuickTest Professional software Data sheet. 2007, 2.)

Ylläpidettävyys helpottuu, kun testejä ajettaessa muutokset tehdään samalla. Työkalu ehdottaa muuttuneisiin kohteisiin muutoksia sen sijaan, että lähettäisi virheraportteja, jotka analysoitaisiin jälkikäteen. Näin säästetään aikaa. (HP QuickTest Professional software Data sheet. 2007, 3.)

Vaatimuksina työkalun käyttöönotolle ovat jokin Windows-käyttöjärjestelmä (XP, Vista tai Windows 2000 Professional) sekä Internet-selaimista Microsoft Internet Explorer 6.0 tai uudempi. Työkalun version uutuuden ja haluttujen ominaisuuksien mukaan esimerkiksi tarvittava muistin sekä vapaan levytilan määrä vaihte-

lee aina noin yhteen gigatavuun asti. (HP QuickTest Professional software System Requirements. 2010.)

8 JOHTOPÄÄTÖKSET

Valittaessa testausprosessin automatisoimiseen työkalua tai muita menetelmiä prosessin parantamiseksi täytyy ottaa huomioon kohteena oleva yritys. Koska tässä tapauksessa ohjelmistokehitysyritys sisältää vain kolme vakituista työntekijää, on haastavaa valita oikeanlainen menetelmä, joka ei kuluta liikaa jo ennestään vähäisiä resursseja.

Selvitys yrityksen nykyisestä testausprosessista paljastaa, että se on melko pitkälle suunniteltu. Käytössä ovat projektinhallintasovellus sekä ohjelmistotyökaluja, jotka helpottavat ohjelmiston kehityksen kokonaiskuvan hahmottamista ja välittävät tietoa muille työntekijöille kohteena olevasta ohjelmasta. Lisäksi ohjelmiston kehitysvaiheessa on käytössä automatisoitu jatkuvan integraation menetelmä, joka mahdollistaa virheisiin puuttumisen mahdollisimman pian ohjelmiston kehityksessä.

Varsinaiseksi kehittämisen kohteeksi muodostuu siis manuaalisesti suoritettava toiminnallinen testaus, jossa testataan ohjelman toimintaa ja tutkitaan tallennusten onnistumista eli varmistetaan ohjelman käytettävyys mahdollisimman oikeankaltaisessa tilanteessa. Tässä vaiheessa löydetyt virheet ovat ongelmallisia, sillä ne palauttavat prosessin ainakin osittain kehitysvaiheeseen. Tehdyt muutokset ja virheiden korjaukset johtavat uudelleentestaukseen eli regressiotestaukseen. Tätä vaihetta yrityksessä ei ole automatisoitu ja sen automatisoiminen olisi kaikista järkevintä resurssien kannalta, jolloin työvoimaa vapautuisi.

Koska yritys on pieni, ei sen ole järkevää lähteä rakentamaan itse automatisoinnin mahdollistavaa työkalua, ellei sillä ole mahdollisuutta irrottaa vähintään yhtä erittäin kokenutta ja ohjelmointitaitoista työntekijää tähän tehtävään. Työkalun rakentaminen vaatisi ehdottomasti oman projektin, johon kuluisi aikaa ja rahaa. Myös valmiin työkalun hankkiminen tulee vaatimaan opetteluvaiheen ja jonkinlaisen pilottiprojektin, jossa voidaan varmistua sen toimivuudesta ja tehokkuudesta testaamaan ohjelmistoa, mutta se pystyisi tarjoamaan myös muita

apuvälineitä ja laajentamismahdollisuuksia testausprosessiin nopeammin kuin itse tekemällä.

Maksulliset testaustyökalut tarjoavat paljon ominaisuuksia ja mahdollisuuksia koko testausprosessin eri vaiheisiin, mutta vaativat ainakin esimerkiksi SilkTestiä käytettäessä useiden tuhansien eurojen sijoituksen (SilkTest. 2010). Jotta tällaisesta laajasta työkalusta saataisiin paras mahdollinen hyöty, saattaisi olla järkevää sulauttaa se koko testausprosessiin eikä pelkästään regressiotestaukseen. Tämä taas on iso projekti, mikä ei välttämättä ole kannattavaa ajallisesti ja rahallisesti, mikäli nykyinen testausprosessi koetaan muutoin hyväksi ja riittäväksi lukuun ottamatta manuaalisen testauksen automatisoimisen toivetta.

Mikäli halutaan ottaa käyttöön jokin työkalu testausprosessiin, ilmaiset työkalut sopisivat parhaiten kohteena olevalle yritykselle. Suoraa rahallista alkuinvestointia ei tarvita, mutta luonnollisesti työvoimaa täytyy pystyä siirtämään projektiin, jossa otetaan käyttöön uusi työkalu. Pelkät nauhoittamalla automatisointiin pyrkivät työkalut eivät ole järkeviä valintoja, sillä niiden ylläpidettävyys voi olla hyvin haasteellista ja tulosten vertailu edelleen manuaalista (Fewster – Graham 1999, 46–48). Sen sijaan myös dokumentaatioon helpotusta ja käyttöönoton helpottamiseksi esimerkkejä ja vinkkejä tarjoavat Concordion (Peterson 2009b) ja Cucumber (Tutorials and Related Blog Posts. 2010). Myös nämä työkalut voidaan ottaa koko testausprosessin jokaiseen vaiheeseen mukaan tai hyödyntää niitä vain haluttujen testien laatimisessa.

Kohteena olevassa yrityksessä tulisi silti vielä tutkia kriittisesti, kuinka paljon on mahdollista ottaa käyttöön työvoimaa nykyisistä resursseista mahdolliseen testauksen automatisoimisprojektiin. Koska kaikkea ei kumminkaan pystytä automatisoimaan ja yritys on melko pieni, saattavat nykyiset käytössä olevat menetelmät olla edelleen hyvä ratkaisu, ellei pystytä tekemään rahallisia ja investointeja työkalun käyttöönottoon liittyen. Lisäksi on myös aina olemassa riski, ettei työkalu tuo haluttuja säästöjä vielä pitkänkään käyttöajan jälkeen.

9 POHDINTA

Tämä opinnäytetyö muodostui lopulta puhtaaksi selvitystyöksi, jonka tehtävät muuttuivat vielä kesken opinnäytetyön teon. Varsinaiseksi tehtäväksi muodostui lopulta tutkia ohjelmistotestauksen automatisoimisen mahdollisuuksia ja löytää työkaluja sen toteuttamiseksi.

Aiheena testausprosessin kehittäminen on tärkeä ja hyödyllinen antaessaan kuvan ohjelmistokehityksen parissa toimivan yrityksen käytännöistä. Lisäksi olisi voinut olla mielenkiintoista myös nähdä ja osallistua käytännössä esimerkiksi jonkun automatisoimiseen valitun työkalun käyttöönottoon.

Koska aihe vaati minulta hyvän perehtymisen koko ohjelmistoprosessiin, oli teoriatiedon löytäminen tärkeää. Tämä osoittautui yllättävän haastavaksi ja etenkin automatisoimisesta kertovaa kirjallisuutta löytyi melko vähän tai ne pohjasivat tietonsa vuonna 1999 Fewsterin ja Grahamin kirjoittamaan kirjaan (Fewster – Graham 1999). Koska kirjan kirjoittamisesta on jo aikaa, etsin tietoa myös internetistä ja sieltä löytyivät etenkin automatisoimiseen käytettävät työkalut ja niiden tiedot.

Koko opinnäytetyöprosessi olisi voitu hoitaa paremmin etenkin aikataulutukseltaan. Koska työ aloitettiin keväällä, se venyi pakostakin syksyyn kesälomien takia, eikä varsinaista tavoitepäivämäärää koskaan asetettu. Työn loppuun saattamista vaikeutti yhteistyön puute yrityksen kanssa, jolloin en saanut palautetta tehdystä selvitystyöstäni ja näin ollen oli haasteellista päätyä suosittelemaan mitään tiettyä työkalua automatisoimisprosessiin. Näin ollen päädyin suosittelemaan ilmaisia työkaluja, jolloin investoinnit jäävät pienemmiksi verrattuna kaupallisiin työkaluihin, sillä yritys on henkilömäärältään pieni.

Haasteista ja vaikeuksista huolimatta olen tyytyväinen, että tämäkin puoli ohjelmistokehityksessä on tullut tutuksi ja selvisin työstä varsin itsenäisesti. Uskon, että mikäli työelämässä kohtaan vastaavankaltaisia tehtäviä ja selvitystöitä,

pystyn selviytymään niistä yhä paremmin ja uskallan kohdata haasteita, joiden tietopohja vaatii täysin uusien asioiden opiskelua, kuten tässä opinnäytetyössä.

LÄHTEET

Ambler, Scott W. 2009. Introduction to Test Driven Design (TDD). Saatavissa: http://www.agiledata.org/essays/tdd.html. Hakupäivä 18.3.2010.

Automated testing. 2010. Saatavissa: http://www.appperfect.com/products/test-automation/automated-testing.html. Hakupäivä 20.7.2010.

Cucumber Behaviour Driven Development with elegance and joy. 2009. Saatavissa: http://cukes.info/. Hakupäivä 4.8.2010.

Data Sheet Silk Test. 2010. Saatavissa: http://www.borland.com/resources/en/pdf/products/silk/Borland-SilkTest.pdf. Hakupäivä 9.8.2010.

Fewster, Mark – Graham Dorothy 1999. Software Test Automation. Effective use of test execution tools. New York: ACM Press.

Haikala, Ilkka – Märijärvi, Jukka 2006. Ohjelmistotuotanto. Jyväskylä: Talentum.

Hendrickson, Elisabeth 1998. The Differences Between Test Automation Success and Failure. Quality Tree Consulting. Saatavissa: http://testobsessed.com/wordpress/wp-content/uploads/2007/01/dbtasaf.pdf. Hakupäivä 11.6.2010.

HP QuickTest Professional software Data sheet. 2007. Saatavissa: https://h10078.www1.hp.com/cda/hpdc/navigation.do?action=downloadPDF&caid=3885&cp=54_4000_100&zn=bto&filename=4AA1-2116ENW.pdf. Hakupäivä 10.8.2010.

HP QuickTest Professional software System Requirements. 2010. Saatavissa: https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^9674_4000_100__. Hakupäivä 11.8.2010.

Marick, Brian. When Should a Test Be Automated? Saatavissa: http://www.exampler.com/testing-com/writings/automate.pdf. Hakupäivä 11.6.2010.

Open Source Software Testing Tools. 2009. Saatavissa: http://maxq.tigris.org/. Hakupäivä 4.8.2010.

Peterson, David 2009a. Concordion. Saatavissa: http://www.concordion.org/. Hakupäivä 4.8.2010.

Peterson, David 2009b. Concordion. Saatavissa: http://www.concordion.org/Tutorial.html. Hakupäivä 9.9.2010.

Pettichord, Bret 2001. Success with Test Automation. Saatavissa: http://www.io.com/~wazmo/succpap.htm. Hakupäivä 11.6.2010.

Product Overwiew. 1998. Saatavissa: http://www.aonix.com/pdf/2140-AON.pdf. Hakupäivä 6.7.2010

Pyhäjärvi, Maaret – Pöyhönen Erkki 2004. Testausvälineet ja testauksen automatisointi. Saatavissa: http://users.jyu.fi/~kolli/testaus2006/materiaali/8_TestausvalineetJaTestauksen Automatisointi_v0_1.ppt. Hakupäivä 20.6.2010.

Rainsberger, J.B. 2005. JUnit Recipes Practical Methods for Programmer Testing. Greenwich: Manning Publications Co.

SilkTest. 2010. Saatavissa: http://www.componentsource.com/products/silktest/index-eur.html. Hakupäivä 9.9.2010.

SilkTest 2010. 2010. Saatavissa: http://www.borland.com/us/products/silk/silktest/index.html. Hakupäivä 9.8.2010.

Siniaalto, Maria 2006. Test driven development: empirical body of evidence. Saatavissa: http://www.agile-itea.org/public/deliverables/ITEA-AGILE-D2.7_v1.0.pdf. Hakupäivä 21.9.2010.

Stenberg, Arto 2007. Ohjelmiston testaus 2007 -johdanto. Saatavissa: http://www.pori.tut.fi/~stenberg/index_files/johdanto.pdf. Hakupäivä 20.9.2010.

Test Automation. Saatavissa: http://www.ivalidator.org/. Hakupäivä 4.8.2010.

TestComplete 7. 2010. Saatavissa: http://www.automatedqa.com/products/testcomplete/. Hakupäivä 20.7.2010.

Testwell CMTJava. 2009. Saatavissa: http://www.testwell.fi/cmtjdesc.html. Hakupäivä 20.7.2010

Tutorials and Related Blog Posts. 2010. Saatavissa: http://wiki.github.com/aslakhellesoy/cucumber/tutorials-and-related-blog-posts. Hakupäivä 9.9.2010.

Zambelich, Keith. Totally data-driven automated testing. Saatavissa: http://www.automation.org.uk/downloads/documentation/white_papers-totally_data_driven_automated_testing.doc. Hakupäivä 20.6.2010.