

Marko Lehtinen

Automatisoitujen regressiotestien elinkaari ja tulosten analysointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinööriytyö

18.5.2014

Tekijä(t) Otsikko	Marko Lehtinen Automatisoitujen regressiotestien elinkaari ja tulosten analysointi
Sivumäärä Aika	38 sivua + 0 liitettä 18.5.2014
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Simo Silander Konsultti Jesse Julkunen
<p>Insinööriyössä oli tavoitteena dokumentoida sovelluskehitysprojektin automatisoidun regressiotestausprosessin elinkaaren vaiheet yksittäisten testien tasolla. Testausprosessista pyrittiin myös kartoittamaan haasteita ja löytämään niihin ratkaisuja regressiotestauksen tehostamiseksi. Lähtökohtana työlle oli osittain toteutettu ja käytössä oleva automaatiotestausprosessi, jonka laatua haluttiin parantaa ja vähentää vaadittavan manuaalisen työn määrää.</p> <p>Työn alussa käsitellään sovellustestausta yleisellä tasolla myöhemmän työn taustaksi. Sen pohjalta esitellään projektin regressiotestauksen nykytila syventyen testausautomaatioon tarkemmin. Automaatiotestien elinkaari esitellään vaiheittain teoreettisesti ja myös teknisen toteutuksen osalta. Kartoitetut haasteet analysoidaan yksitellen ja niiden ratkaisemiseksi esitellään todettuja kehitysehdotuksia. Työssä käydään läpi myös kehitysehdotusten pohjalta tehtyjä konkreettisia toteutuksia.</p> <p>Insinööriyön tuloksena syntyi kattava dokumentaatio sovelluksen regressiotestauksesta sovelluskehitysprojektin tarpeisiin. Automaatiotestausprosessiin tehtiin insinööriyön myötä useita laatua ja vakautta parantavia muutoksia, joiden myötä testien ylläpitämisen tarve on vähentynyt ja regressiotestauksen laatu parantunut.</p>	
Avainsanat	testaus, regressiotestaus, automaatiotestaus, automaatio, selenium

Author(s) Title	Marko Lehtinen Life Cycle of Automated Regression Tests and Analysis of Test Results
Number of Pages Date	38 pages + 0 appendices 18 May 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Simo Silander, Senior Lecturer Jesse Julkunen, Consult
<p>The goal of the thesis was to provide a documentation of an automated regression testing process in a software development project. It was done by examining the life cycle of automated tests and detecting possible problems in the current automation implementation. The intention was to find solutions and put them into practice to improve the quality of the testing process.</p> <p>The concepts of testing in general are described in the beginning of the thesis. Based on these concepts the current state of the regression testing process in project is then presented. Theoretical and technical in-depth analysis for the life cycle of the automated regression tests is executed and problems in testing process are identified. The thesis demonstrates in detail the solutions found and developing proposals for each individual problem.</p> <p>As a result of the thesis, a detailed documentation of the regression testing process for the software in development was created. Numerous improvements were carried out in an automation testing process, improving the quality and stability of the testing framework. As an outcome, the need for test maintenance was decreased and simultaneously the quality of the regression testing was improved.</p>	
Keywords	Testing, Regression Testing, Automation Testing, Automation, Selenium

Sisällys

Lyhenteet ja käsitteet

1	Johdanto	1
2	Testaus osana iteratiivista sovelluskehitystä	2
2.1	Iteratiivinen sovelluskehitys	2
2.2	Testausstrategiat	3
2.3	Testauksen prosessimalli ja testaustasot	4
2.3.1	V-malli	4
2.3.2	Yksikkötestaus	5
2.3.3	Integrointitestaus	5
2.3.4	Järjestelmätestaus	5
2.3.5	Hyväksymistestaus	5
2.4	Regressiotestaus	6
2.4.1	Regressiotestauksen tarkoitus	6
2.4.2	Regressiotestauksen hyödyntäminen iteratiivisessa kehityksessä	6
2.4.3	Automaation hyödyntäminen regressiotestauksessa	6
3	Regressiotestauksen nykytila projektissa	9
3.1	Regressiotestausprosessi	9
3.2	Regressiotestitapausten automatisointi	10
4	Automaatiotestien elinkaari	11
4.1	Yleisesti automaatiotestien elinkaaresta	11
4.2	Testitapaukset	12
4.3	Automaatiotestien toteuttaminen	13
4.3.1	Automatisoinnin työkalut	13
4.3.2	Testitapauksesta automaatiotestiksi	15
4.4	Automaatiotestien ajaminen	17
4.5	Automaatiotestien ylläpidettävyys	18
4.6	Tulosten seuranta ja luotettavuus	19
4.7	Automaatiotestauksessa havaitut virheet	20
5	Projektissa kohdatut automaation haasteet	21

5.1	Testiaineisto ja sen muutokset	21
5.2	Automaatiotestien laadulliset ja suoritusajalliset tekijät	22
5.3	Testien häiriönsieto	23
5.4	Sovelluksen arkkitehtuuri ja haarautuminen	24
5.5	Ulkoiset tekijät	26
6	Automaation kehittäminen projektissa	26
6.1	Sopeutuminen testiaineistojen muutokseen	27
6.2	Modulaarisuuden ja laadun parantaminen	28
6.3	Yhtenäinen ohjeistus ja kommunikaatio	31
6.4	Testien automaattisen ajamisen kehittäminen	32
7	Automaatiotestien tulokset	33
7.1	Manuaalinen työ	34
7.2	Tulosten analysoinnin uudet menetelmät	34
7.3	Raportointi	35
8	Yhteenveto	35
	Lähteet	37

Lyhenteet ja käsitteet

AJAX	Ajax eli Asynchronous JavaScript and XML on kokoelma verkkotekniikoita, jotka mahdollistavat muun muassa verkkosivujen dynaamisen päivittämisen.
Apache Ant	Apache Ant on työkalu automaattisten käännösten tekemiseksi XML-perustaisten skriptien pohjalta.
Hibernate	Hibernate on Java-kirjasto olio- ja relaatiomallien yhteensovittamiseksi sovelluksessa.
Jenkins	Jenkins on jatkuvan integraation palvelin, joka perustuu Hudson-palvelimeen.
Selenium IDE	Selenium IDE on Firefox-selaimen lisäosa, joka mahdollistaa yksinkertaisten verkkosovellusten testaamisen suoraan selaimessa.
Selenium Server / RC	Selenium Server on laajennettava Java-sovellus Selenium-testien toteuttamiseksi ohjelmoimalla. Selenium Server on osa Selenium 2 -kokonaisuutta. Ensimmäisessä versiossa se tunnettiin nimellä Selenium RC eli Remote Control.
Selenium WebDriver	Selenium WebDriver on rajapinta Selenium-testien ajamiseksi selaimessa.
SSH	SSH eli Secure Shell on protokolla turvalliseen ja salattuun tietoliikenteeseen.
SVN	SVN eli Subversion on avoimen lähdekoodin versionhallintajärjestelmä.
XML	XML eli Extensible Markup Language on merkintäkieli tiedon jakamiseksi ja säilyttämiseksi.

1 Johdanto

Testauksen rooli ohjelmistokehityksessä on merkittävä. Tarve testaukselle korostuu erityisesti, kun kehitetään uutta, monimutkaista ja toiminnallisesti laajaa järjestelmää. Regressiotestauksen suorittaminen on tärkeä osa pitämään sovellus eheänä ja laatu-taso korkeana koko sovelluskehityksen elinkaaren ajan. Regressiotestauksella pyritään havaitsemaan sovelluksen muutoksista syntyneet virheet kaikissa sovelluksen osissa.

Tässä insinööriyössä kartoitetaan automatisoidun regressiotestauksen nykytila suu-ressa sovelluskehitysprojektissa. Projektin tilaajana on monta eri asiakasta. Asiakkai-den sovellukseen kohdistuvat vaatimukset ja määrittelyt eroavat toisistaan hieman, joten nämä erot tulee ottaa huomioon myös testaamisessa. Tekstissä kuvataan regres-siotestausprosessissa havaittuja haasteita ja tutkitaan, millä tavoin testausautomaatiota voisi kehittää. Työssä käydään läpi projektin näkökulmasta automaatiotestien koko elinkaari kehittämisestä ylläpitoon ja tulosten analysointiin. Lisäksi etsitään keinoja, kuinka automaatio voisi palvella regressiotestauksessa entistä paremmin tarkoitustaan. Työssä myös pohditaan keinoja manuaalisen työn vähentämiseksi karsimatta regres-siotestauksen laadusta.

Työ tehdään finanssialan sovellusprojektin toteuttavalle konsulttiyritykselle jo olemassa olevan ja jatkuvasti laajenevan automatisoidun regressiotestauksen kehittämiseksi. Tavoitteena on tutkimuksen ohella toteuttaa insinööriyössä nousseita ideoita projektin automatisoidun regressiotestauksen hyödyksi.

Projektissa kehitettävän sovelluksen jatkuva regressiotestaus suoritetaan suurimmalta osin ajamalla Selenium-automaatiotestejä. Testien ajamiseen käytetään jatkuvan integ-raation Jenkins-palvelinta, mutta automaatiotestejä on mahdollisuus ajaa myös paikal-lisesti kehittäjän työasemalla. Lisäksi osa testeistä ajetaan käytännön syistä manuaali-sesti.

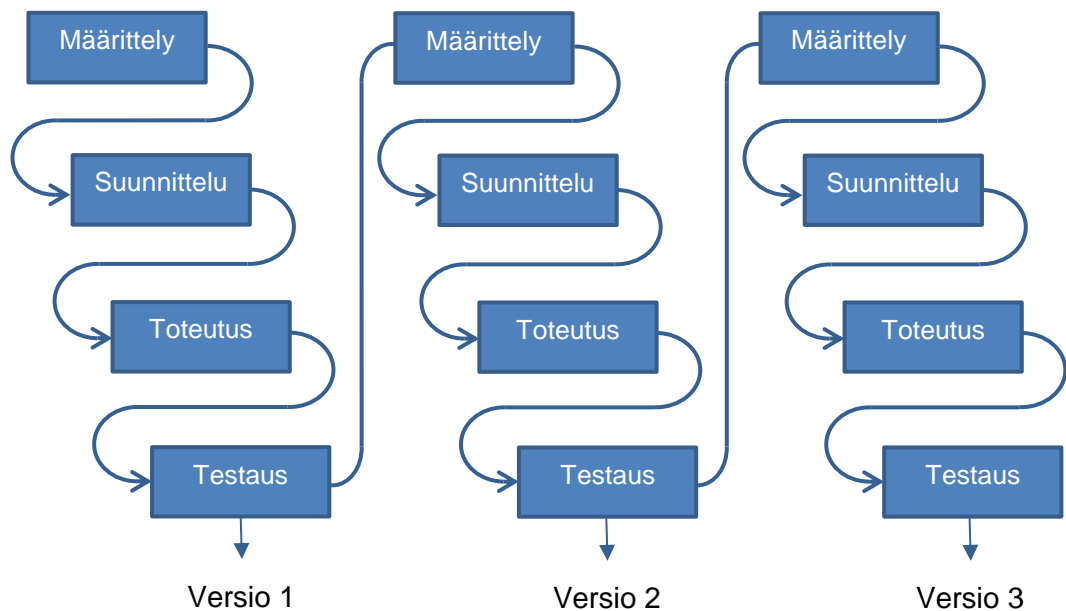
Olen toiminut projektissa automaatiotestaajan tehtävissä vuoden ajan. Hyödynnän in-sinööriyössä tästä saamaani kokemusta tietoperustana ja analysoinnin lähteenä.

2 Testaus osana iteratiivista sovelluskehitystä

Tässä luvussa käsitellään sovellustestausta yleisesti teoriasella insinööriyön taustaksi. Luvussa käydään läpi testauksen terminologiaa, prosessimalleja ja testaustasoja. Regressiotestaukseen syvennytään luvun loppupuolella alustamaan seuraavia lukuja, joissa esitellään regressiotestausta projektin näkökulmasta.

2.1 Iteratiivinen sovelluskehitys

Iteratiivinen sovelluskehitys etenee sykleittäin siten, että sovellukseen kohdistuvien vaatimusten toteuttamiset jaetaan pienempiin osiin eli iteraatioihin. Näitä vaatimuksia ovat kaikki toiminnallisuudet, jotka halutaan lopputuotteesta löytyvän. Yksittäisten iteraatioiden toiminnallisuudet määritellään, toteutetaan ja testataan erikseen ennen seuraavan iteraation kehityksen aloittamista. Tätä iteratiivisuutta suoritetaan peräkkäin, kunnes päästään haluttuun lopputulokseen, tässä tapauksessa valmiiseen sovellukseen, joka täyttää kaikki sovelluksen määrittelyt. Iteratiivisuus mahdollistaa muun muassa sovellusversioiden luovutuksen vaiheittain asiakkaalle esimerkiksi siten, että toimitettava versio käsittää muutaman iteraation sisältämät vaatimukset. [1, s. 42.]

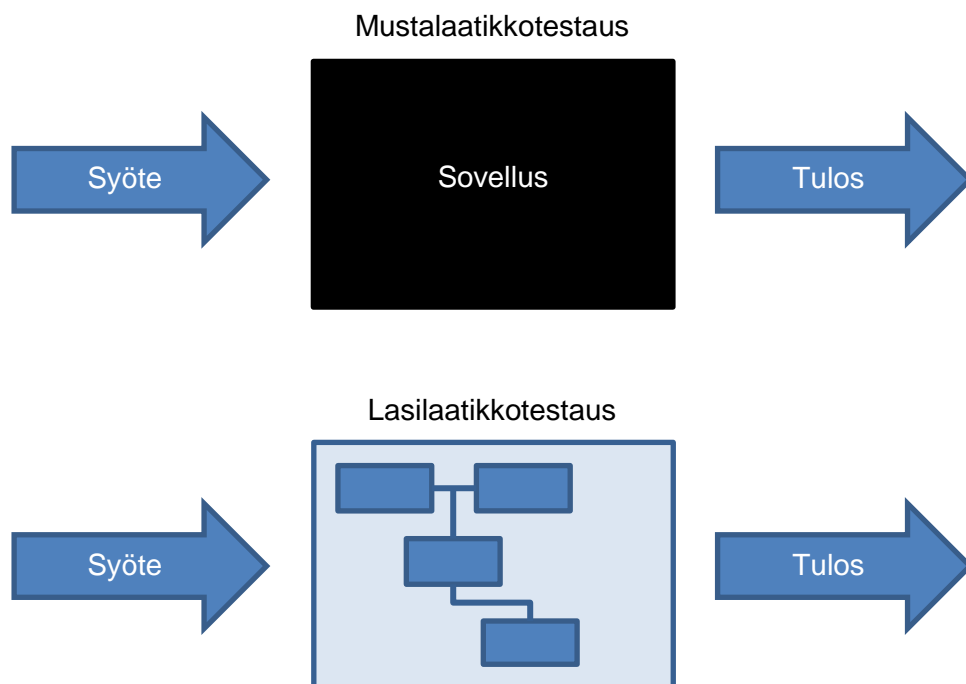


Kuva 1. Esimerkki iteratiivisesta sovelluskehityksestä

Iteratiivisuuden edut näkyvät myös siinä, että sovelluksen määrittämiä voidaan tarkentaa kesken kehitystyön. Tämä kuitenkin näkyy lisääntyneenä sovelluksen testaustarpeena, mistä johtuen muun muassa lisääntyvä regressiotestauksen määrä puoltaa testiautomaation käyttöä. [1, s. 42.]

2.2 Testausstrategiat

Testausta voidaan perinteisesti toteuttaa mustalaatikko- (black box testing) tai lasilaatikko- (white box testing) testausstrategialla (white box testing). On myös mahdollista yhdistää molempien mainittujen strategioiden ominaisuuksia suorittamalla niin kutsuttua harmaalaatikko- (grey box testing) testausstrategiaa. Mustalaatikko- (black box testing) testauksessa sovellusta testataan ainoastaan ulkoisesti havaittavan toiminnallisuuden perusteella, eli testaaja ei tiedä, mitä sovelluksen loogisella, eli kooditasolla tapahtuu. Tällöin regression testitapauksissa tarkastellaan ainoastaan käyttäjille näkyvien osien toimivuutta ja vastaavuutta määrittelyihin. Testauksen voi toteuttaa myös lasilaatikko- (white box testing) testausstrategiana, jolloin testaajalla on pääsy sovelluksen koodiin, algoritmeihin ja esimerkiksi tietokantaan. [2, s. 118; 3, s. 42.]



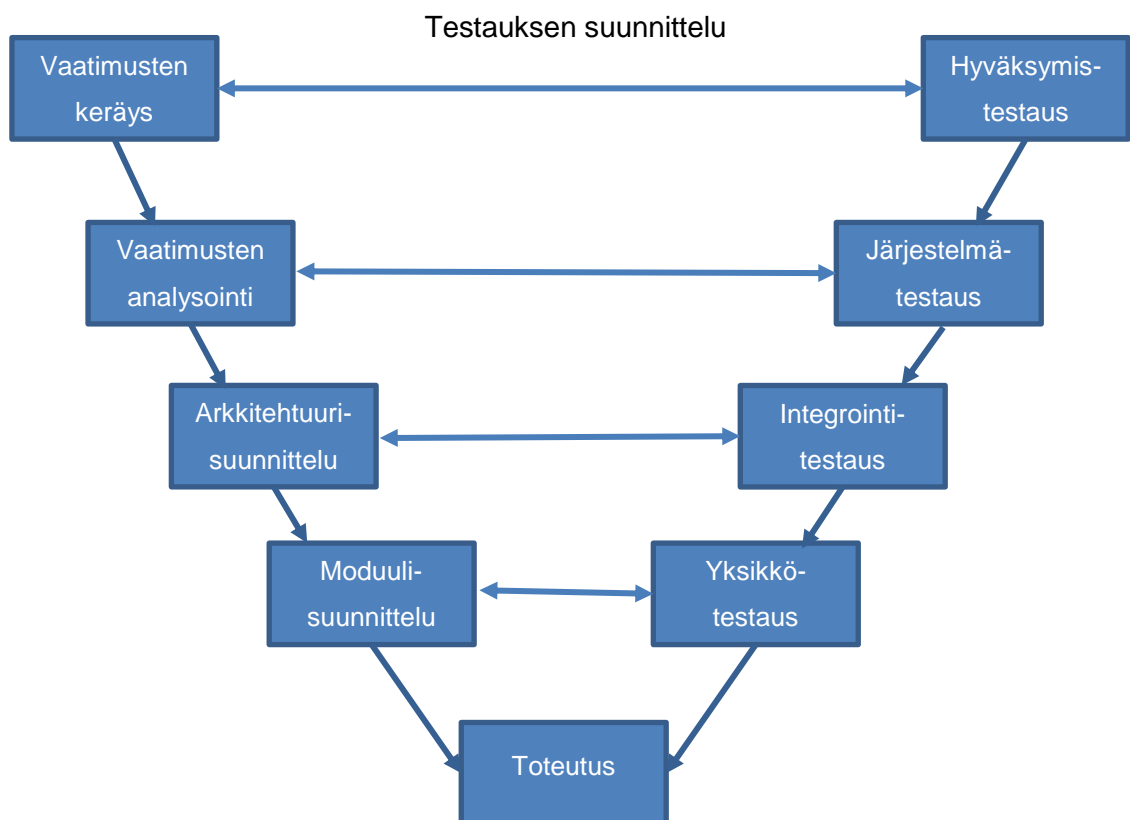
Kuva 2. Mustalaatikko- ja lasilaatikkotestaus

Näitä kahta strategiaa yhdistelevä harmaalaatikkotestaus testaa sovellusta molemmat näkökulmat huomioiden. Tällöin testausta tapahtuu niin käyttöliittymän kautta kuin myös suoraan järjestelmän loogisia osia vasten. Harmaalaatikkotestaus soveltuu hyvin useista irrallisista komponenteista koostuvien sovellusten, kuten esimerkiksi web-sovellusten, testaamiseen. [4.]

2.3 Testauksen prosessimalli ja testaustasot

2.3.1 V-malli

V-malli on yksi yleisesti käytetty prosessimalli havainnollistamaan sovelluskehitystä ja sen testausta. V-mallin perusajatus on, että sovelluksen testaussuunnittelu tapahtuu kehitysvaiheessa jokaista testaustasoa vastaavalla tasolla, kuten kuvassa 3 havainnollistetaan. Alimpana linkkinä mallissa on itse sovelluksen toteutus eli ohjelmointi. [1, s. 207.]



Perinteisen V-mallin mukaisesti testausstrategian käyttäminen muuttuu lasilaatikkotestauksesta enemmän mustalaatikkotestaukseen alimmalta testaustasolta ylöspäin edessä. Regressiotestausta voidaan kuitenkin suorittaa testaustasosta riippumatta, eli se voi sisältää esimerkiksi järjestelmä-, integrointi- ja yksikkötestaamista. Insinööriyössä keskitytään järjestelmätestitapauksien uudelleensuorittamiseen perustuvaan regressiotestaukseen, jossa hyödynnetään mustalaatikkotestauksen ohella myös lasilaatikkostrategiaa. [1, s. 208-209.]

2.3.2 Yksikkötestaus

Yksikkötestaus tunnetaan myös nimillä moduuli- ja komponenttitestaus. Yksikkötestauksessa testataan sovelluksen luokan tai moduulin pienimpiä osia, yksiköitä, jotka voivat olla esimerkiksi yksittäisiä metodeja. Yksikkötestit toteuttaa yleensä testattavan sovellusosan ohjelmoija itse. [1, s. 207.]

2.3.3 Integrointitestaus

Integrointitestaus muistuttaa järjestelmätestausta, mutta siihen verrattuna keskittyy voimakkaammin sovelluksen tekniseen toteutukseen. Integrointitestauksessa testataan yksiköiden toimintaa yhdessä, siten kuin osat toimivat sovelluskokonaisuudessa. [2, s. 354.]

2.3.4 Järjestelmätestaus

Järjestelmätestauksessa testataan koko järjestelmän toimintaa, eli kaikkien sovellusten toimintaa yhdessä. Testaus tapahtuu korkeammalla tasolla kuin yksikkö- ja integrointitestauksessa, eli testauksessa hyödynnetään enemmän vain ulkoisesti havaittavaa toiminnallisuutta ottamatta kantaa toteutustapaan. [2, s. 356.]

2.3.5 Hyväksymistestaus

Hyväksymistestauksessa todetaan sovelluksen lopullinen vastaavuus määrittelyihin. Toisin kuin järjestelmätestaus, hyväksymistestauksen suorittaa yleensä sovelluksen tilaaja tai muu loppukäyttäjä. Tehokas tapa toteuttaa hyväksymistestaus on pyrkiä testaamisen keinoin osoittamaan, että sovellus ei täytä vaatimuksia, ja jos näin ei voida osoittaa, on sovelluksen luovutus hyväksytty. [3, s. 31.]

2.4 Regressiotestaus

2.4.1 Regressiotestauksen tarkoitus

Regressiotestauksella pyritään varmistamaan, että sovellus toimii ongelmitta ja määritysten mukaisesti vielä sovellukseen kohdistuvien muutosten jälkeen. Regressioita, eli muutoksesta johtuvia virheitä, voi aiheutua esimerkiksi uusien toiminnallisuuksien toteuttamisesta tai virhekorjausten tekemisestä. Kattavalla regressiotestauksella havaitaan muutoksen aiheuttamat odottamattomat ongelmat vanhoissa sovelluksen osissa – myös muutetun komponentin ulkoisilla osa-alueilla. Mitä laajemmasta sovelluksesta on kyse, sitä todennäköisemmin regressioita syntyy jossain ohjelman osassa. Tästä syystä regressiotestaus on hyvin tärkeää sovelluksen eheyden ja toimivuuden säilyttämiseksi. [3, s. 134, 162.]

2.4.2 Regressiotestauksen hyödyntäminen iteratiivisessa kehityksessä

Insinöörityön käsittelemää sovellusta kehitetään iteratiivisesti. Iteratiivinen sovelluskehitys on otettava huomioon sovellusta testattaessa, ja se korostaa regressiotestauksen tärkeyttä; regressiotestaus on tärkeä osa iteratiivisen sovelluskehityksen testausprosessia [5, s. 44]. Iteratiiviselle sovelluskehitykselle on ominaista nopeat ja suuret muutokset, joiden sivuvaikutukset pyritään havaitsemaan regressiotestaamalla. Muutettava sovellus on usein alttiimpi toteutuksesta johtuville virheille kuin uusi tyhjästä toteutettava sovellus [3, s. 22]. Tästä syystä regressiotestit tulisi suorittaa määrääjain sovelluksen laadun varmistamiseksi mahdollisimman usein. Ideaalitapauksessa koko regressiotestijoukko ajettaisiin jokaisen pienimmänkin muutoksen jälkeen, mutta käytännössä tämä on hyvin vaikeaa tai tietyissä tapauksissa mahdotonta, sillä kattava regressiotestaaminen vie runsaasti aikaa [6].

2.4.3 Automaation hyödyntäminen regressiotestauksessa

Regressiotestausta voidaan helpottaa automatisoimalla regressiotestaamista mahdollisimman paljon. Tämä tuo regressiotestaamiseen useita hyötyjä: työmäärä testien ajamiseksi on pienempi, testejä voidaan ajaa nopeammin ja useammin. Automaation ajamat testit toistetaan aina samalla tavalla ja automaatiolla voidaan toteuttaa testejä, jotka olisivat manuaalitestauksella vaikeita tai jopa mahdottomia toteuttaa. Tällaisia

voivat olla esimerkiksi testit, jotka vaativat epäinhimillisen nopean suoritusnopeuden tai testit, jotka seuraavat reaaliajassa taustajärjestelmien toimintaa. [7, s. 6.]

Automaatiotestien tärkeimpiin ominaisuuksiin kuuluu uudelleenkäytettävyys. Kerran toteutetut automatisoidut regressiotestit voidaan ajaa sellaisenaan koska tahansa uudelleen. Samoja automaatiotestejä voi käyttää esimerkiksi useiden sovellusversioiden samanaikaiseen testaamiseen; monta asiakasta käsittävässä projektissa voidaan samat automaatiotestit suorittaa erikseen jokaiselle asiakkaalle. Pitää kuitenkin ottaa huomioon, että vaikka automaatiotestit voidaan ajaa manuaalista regressiotestausta huomattavasti pienemmällä testaustyömäärällä, vaatii silti automaatiotestien toteuttaminen etenkin alussa paljon resursseja. Yhden automaatiotestin toteuttaminen vie enemmän aikaa ja vaivaa kuin usean manuaalisesti suoritettujen regressiotestien ajon puhumattakaan resursseista, joita vaaditaan kokonaisen automaatiotestaamisen pohjalle rakennettavan testauskehiksen toteuttamiseksi. Ajankäyttäminen kuitenkin kannattaa – hyvä pohjatyö mahdollistaa uusien automaatiotestien nopeamman toteuttamisen, ja laadukas automaatiotesti on helpompi ylläpitää ja vähemmän altis virheille kuin puolihuolimattomasti toteutettu testi. [7, s. 497.]

Automaatiotestausta puoltaa myös se, että automaatiotestien suorittamiseen kuluva aika on huomattavasti lyhyempi, kuin mitä se olisi kokonaan testitapauksen manuaalisesti suorittamalla. QAI Global Instituten marraskuussa 1995 toteuttamassa tutkimuksessa, jossa vertailtiin manuaalista ja automaattista testausta, todettiin automaation vähentäneen testaukseen kuluvan ajan noin neljännekseen. Tutkimuksessa vertailtiin testaamiseen kuluva aikaa työtunteina. Nykyaikana automaatiotestaustyökalujen kehittyä entisestään, voi automaation hyötysuhde olla vielä tätäkin korkeampi. [2, s. 49-50.]

Taulukko 1. QAI Global Instituten tutkimustulos automaattisen ja manuaalisen testauksen viemästä ajasta [6, s. 50.]

	Manuaalinen testaus (h)	Automaatio-testaus (h)	Automaation hyötysuhde
Testaus-suunnittelu	32	40	-25%
Testausmenettelyn kehittäminen	262	117	55%
Testien suorittaminen	466	23	95%
Tulosten analysointi	117	58	50%
Virheiden seuranta	117	23	80%
Raporttien luonti	96	16	83%
Kokonaiskesto	1090	277	75%

Suoritusajalliset syyt ja vähentynyt testaustyömäärä eivät ole ainoat syyt regressiotestauksen automatisoinnille. Automaatiotestaus varmistaa, että testit ajetaan samoin joka kerta. Tällöin ei pääse syntymään inhimillisiä virheitä testauksessa. Automaatiotesti vertailee muun muassa merkkijonoja ja liukulukuja määrittelyiden ja sovelluksen välillä tarkemmin ja nopeammin kuin yksikään ihminen. Hyvin toteutettu automaatiotesti huomaa virheet, jotka saattaisivat manuaalitestauksen yhteydessä jäädä huomaamatta; hyvin toteutettu automaatiotesti ajetaan joka ajokerralla täysin samalla tavalla, eikä yhtäkään testiaskelta jää epähuomiossa välistä. Automaatiotestaus parantaa testaamisen laatua ja sitä kautta taas sovelluksen laatutasoa. [7, s. 346.]

Automaatiotestaus on parhaimmillaan silloin, kun se vaatii mahdollisimman vähän manuaalista työtä tuottaen kuitenkin hyvää tulosta. Automaation tuloksellisuuden mittaamiseen soveltuvat useat eri mittarit, joita voi soveltaa muuhunkin testaamiseen. Tulosta voidaan mitata testien suorittamalla koodikattavuudella, määrittelyjen testauksen kattavuudella, havaittujen virheiden määrällä, uudelleenesiintyvyydellä ja korjausnopeudella. [7, s. 508-512.]

Automaatiota voidaan testien ajamisen ohella hyödyntää myös muilla testaamisen osalueilla, kuten esimerkiksi tulosten analysoinnissa, virheiden monitoroinnissa ja raporttien laatimisessa [2, s. 38]. Tämä tarkoittaa optimitilanteessa sitä, että uuden testattavan sovellusversion saapuessa automaatiotestit menisivät automaattisesti suorituk-

seen, ja viimeisen testin suorittamisen jälkeen tulokset ja kattavuusraportit generoituisivat automaattisesti.

3 Regressiotestauksen nykytila projektissa

Esittelen tässä luvussa lyhyesti projektissa toteutettavan sovelluksen regressiotestausprosessia ja tapamme hyödyntää automaatiota regressiotestauksessa. Automaatiotestausprosessia avataan tarkemmin luvussa 4 kartoittamalla automaatiotestien elinkaaren vaiheet yksityiskohtaisesti.

3.1 Regressiotestausprosessi

Insinöörityön käsittelemässä projektissa toteutettavan sovelluksen kehitys etenee vaiheittain siten, että joka vaiheessa määritellään ja toteutetaan tietty osa kokonaisuuteen vaadituista toiminnallisuuksista. Tämä osa pilkotaan erikseen vielä pienempiin iteraatioihin. Jokaiselle iteraatiolle on omat toiminnallisuutta testaavat järjestelmätestitapauksensa, joista tärkein osa valitaan regressiotestausjoukkoon. Valintaprosessissa testitiimin jäsenet käyvät läpi kaikki järjestelmätestitapaukset ja valikoivat tietyn prosenttimäärän kaikista testeistä regressiotestausjoukkoon. Testijoukon valinta pyritään tekemään siten, että valitut testitapaukset testaavat mahdollisimman monipuolisesti järjestelmän kaikkia osa-alueita, jolloin joukkoon valikoituvat testit, joilla on korkein todennäköisyys havaita suurin osa virheistä. [3, s. 41.]

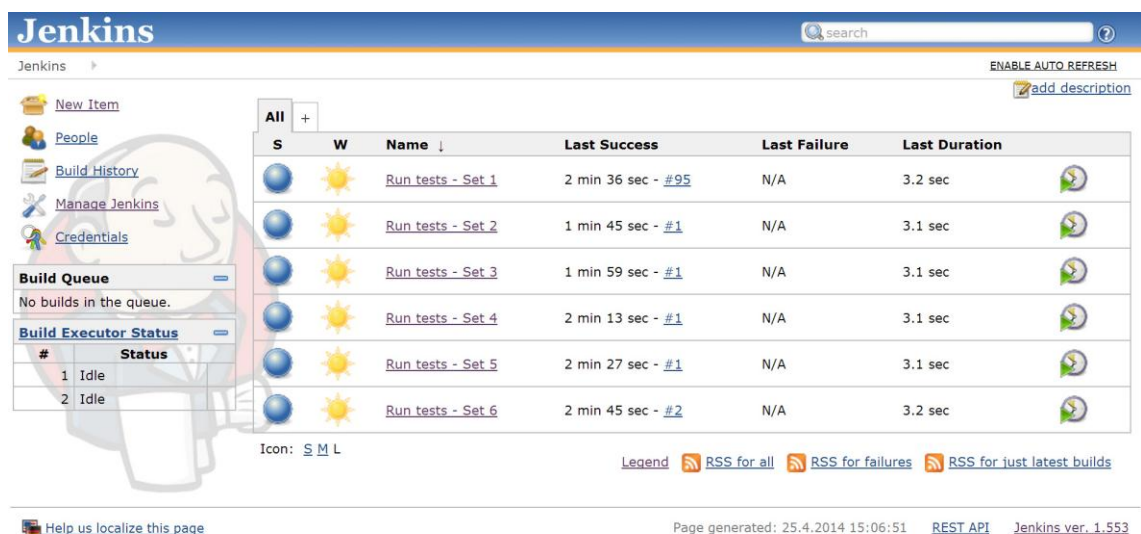
Projektissa sovelluksen regressiotestausta suoritetaan ajamalla uudelleen valikoitu osa järjestelmätestitapauksia sovellusversion käännöstä vasten. Valtaosa regressiotestitapauksista ajetaan automaatiota hyödyntämällä. Loput automatisoiduksi kelpaamattomat testit ajetaan erikseen manuaalisesti.

Regressiotestijoukon testitapaukset pyritään ensisijaisesti automatisoimaan, mutta ne tapaukset, joiden automatisointi ei ole mahdollista, ajetaan manuaalisesti sovellusversiota testattaessa. Tällaisia voivat olla esimerkiksi testitapaukset, joissa tarvitaan ihmismillemää sivun asettelun ja fonttien tarkastamiseksi tai tapaukset, joita ei voi teknisistä syistä toteuttaa automaattisesti ajettaviksi. Teknisiä esteitä testin automatisoinnille voi olla monia, kuten tarve ajaa testi vasten ulkoisia rajapintoja, joihin ei automaatioympäristöistä ole pääsyä.

3.2 Regressiotestitapausten automatisointi

Testattavan sovelluksen regressiotestijoukko käsittää suuren määrän testitapauksia – määrä liikkuu useissa sadoissa ja kasvaa jatkuvasti projektin edetessä. Lisäksi samat regressiotestit ajetaan useille asiakkaille, mikä lisää entisestään regressiotestauksen vaativuutta. Asiakaskohtaisuudet ja eroavaisuudet asiakkaiden toimintaprosesseissa tulee ottaa erikseen huomioon jokaisessa testitapauksessa. Testitapaukset ja niiden käsittämät skenaariot ovat keskimäärin hyvin laajoja ja kestoiltaan pitkiä, joten automaation hyödyntämiselle regressiotestauksessa on selkeät perusteet.

Automaatiotestien ajamiseen käytetään projektin regressiotestauksessa Jenkinsiä, joka mahdollistaa testien ajamisen ulkopuolisella Linux-palvelimella. Jenkins on avoimeen lähdekoodiin perustuva jatkuvan integraation palvelin [9]. Regressiotestijoukon automaatiotesteille on luotu Apache Ant -käännöstyökalua käyttämällä omat XML-muotoiset määrittelynsä, joita käyttäen Jenkins ajaa testit. Testien suorittaminen Jenkins-palvelimella ei ole vielä täysin automatisoitu, sillä toistaiseksi testit käynnistetään manuaalisesti yksi testikokonaisuus kerrallaan Jenkinsin verkkohallinnan käyttöliittymästä. Projektin tapauksessa yksi testikokonaisuus on yhden iteraation aikana toteutetut toiminnallisuudet testaava joukko. Manuaaliseen testien käynnistämiseen on päädytty syystä, että testit vaativat vielä jonkin verran manuaalista seuranta, mihin syvennytään tarkemmin insinööriyön myöhemmissä osissa.



The screenshot shows the Jenkins web interface. At the top, there is a search bar and a 'Jenkins' logo. Below the logo, there are navigation links: 'New Item', 'People', 'Build History', 'Manage Jenkins', and 'Credentials'. On the left, there is a 'Build Queue' section showing 'No builds in the queue.' and a 'Build Executor Status' table with two executors in an 'Idle' state. The main content area displays a table of test sets:

S	W	Name ↓	Last Success	Last Failure	Last Duration
🌐	☀️	Run tests - Set 1	2 min 36 sec - #95	N/A	3.2 sec
🌐	☀️	Run tests - Set 2	1 min 45 sec - #1	N/A	3.1 sec
🌐	☀️	Run tests - Set 3	1 min 59 sec - #1	N/A	3.1 sec
🌐	☀️	Run tests - Set 4	2 min 13 sec - #1	N/A	3.1 sec
🌐	☀️	Run tests - Set 5	2 min 27 sec - #1	N/A	3.1 sec
🌐	☀️	Run tests - Set 6	2 min 45 sec - #2	N/A	3.2 sec

At the bottom of the table, there are icons for 'S', 'M', and 'L'. Below the table, there are links for 'Legend', 'RSS for all', 'RSS for failures', and 'RSS for just latest builds'. At the very bottom, there is a footer with 'Help us localize this page', 'Page generated: 25.4.2014 15:06:51', 'REST API', and 'Jenkins ver. 1.553'.

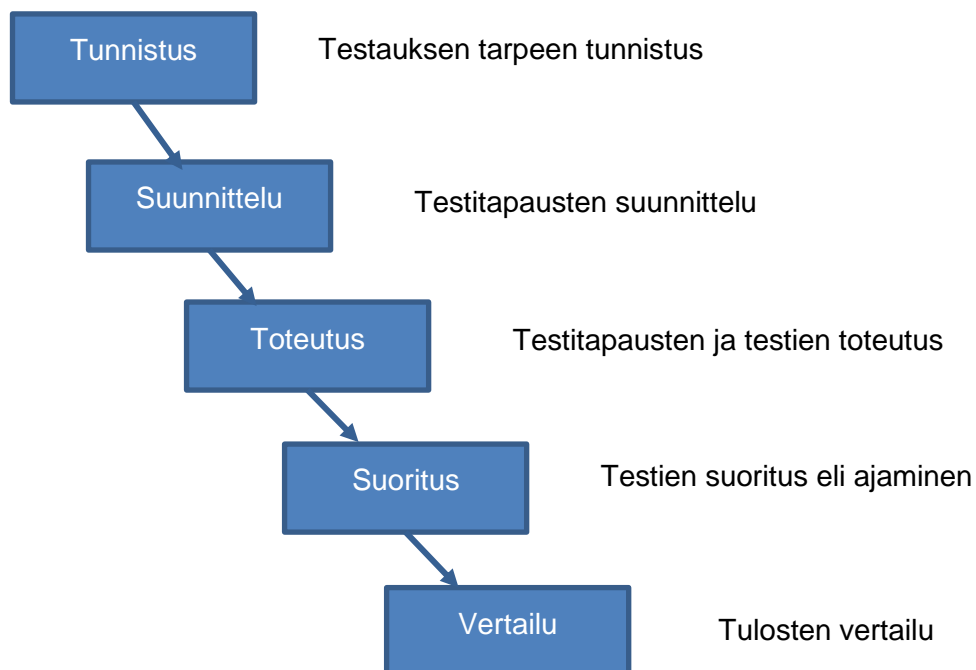
Kuva 4. Jenkins-integraatiopalvelimen verkkohallintanäkymä

4 Automaatiotestien elinkaari

Luvussa käydään läpi automaatiotestien koko elinkaari suunnittelu- ja toteutusvaiheesta ajoon ja ylläpitoon keskittyen projektin näkökulmaan. Luvussa esitellään teorian ohella myös automaatiotestauksen tekniset taustat, kuten käytetyt työkalut ja muut toteutukselliset ja ylläpidolliset asiat. Elinkaareissa kohdattuja haasteita ja kehitystarpeita kartoitetaan lukua 4 seuraavissa luvuissa.

4.1 Yleisesti automaatiotestien elinkaaresta

Testausprosessi etenee pääpiirteittäin alla olevan kuvan 5 mukaisesti, mutta voi myös poiketa kaaviossa esitetystä rakenteesta. Ensimmäisessä vaiheessa pyritään tunnistamaan testauksen tarve, eli löytämään sovelluksesta ne toiminnallisuudet, joiden testaus halutaan kattaa. Seuraavassa vaiheessa suunnitellaan testitapaukset tunnistettujen tarpeiden mukaisesti. Toteutusvaiheessa luodaan lopulliset testitapaukset, joiden pohjalta toteutetaan automatisoidut testit. Valmiit automaatiotestit suoritetaan eli ajetaan. Lopuksi testien syötteille ja lopputuloksille suoritetaan testitapauksessa määritetyt tarkistukset, joilla varmistetaan sovelluksen toimivuus. [7, s. 13-17.]



Kuva 5. Prosessikaavio automaatiotestien elinkaaresta [8, s. 13.]

Tässä projektissa testausprosessi etenee vastaavasti. Kaaviosta poiketen testitapausten ja automaatiotestien toteutuksen jälkeen testit siirtyvät jatkuvaan suoritus- ja vertailuvaiheeseen, jonka aikana testejä ylläpidetään. Prosessin kulkua ja testien elinkaarta on kuvailtu tarkemmin luvuissa 4.2-4.7.

4.2 Testitapaukset

Testitapaus on yksityiskohtainen dokumentaatio vaatimuksista, toimenpiteistä ja odotetuista tuloksista tietyn tavoitteen suorittamiseksi ja toiminnallisuuden oikeellisuuden varmistamiseksi [5, s. 139]. Kaikkia sovelluksen käyttötapauksia on käytännössä mahdotonta testata, joten testitapausten suunnittelu on tärkeää tehdä siten, että ne testaavat mahdollisimman kattavasti kaikki sovelluksen kriittisimmät osuudet, eli tärkeimmät ja virhealtteimmat toiminnallisuudet. Kriittisyyksiä ja virhealttiuksia voidaan kartoittaa esimerkiksi riskianalyysillä tai moduulien monimutkaisuuden perusteella. [3, s. 41, 104.]

Projektin järjestelmä- ja regressiotestauksessa käytettävät testitapaukset perustuvat aitoja sovelluksen käyttötilanteita jäljitteleviin skenaarioihin. Valtaosassa testitapauksista on pääosassa oikea henkilö, joka on sovelluksen tilaajan asiakas. Henkilöön viitataan sovelluksessa salatun nimen ja henkilötunnuksen perusteella.

Testitapaus voi käsittää esimerkiksi skenaarion, jossa sovellusta käytävä käsittelijä, eli tässä tapauksessa testin suorittaja, luo henkilölle uuden käsiteltävän asian, täyttää sovelluksessa asian lomakkeelle henkilön tiedot ja suorittaa sen käsittelyyn liittyviä toimenpiteitä. Tässä ohessa voidaan tehdä useita testitapauksessa kuvattuja tarkastuksia niin näytöllä kuin tietokannassakin: Näin varmistetaan, että prosessi etenee määritysten mukaisesti ja käyttöliittymäelementit näkyvät näytöllä oikein.

Osa projektin testitapauksista sisältää hyvinkin monimutkaisia ja yksityiskohtaisia skenaarioita, jotka voivat kestää ajallisesti hyvin kauan - korkeimmillaan jopa tunnin verran. Keskimäärin testitapaukset ovat kuitenkin noin muutaman minuutin kestoisia automaattisesti ajettuina. Manuaalisesti ajettuna kesto olisi jonkin verran pidempi.

Käytännössä testitapaukset on toteutettu vaihe- ja iteraatiokohtaisesti taulukkomuotoisina käsikirjoituksina eli testiskripteinä. Jokainen testitapaus vaatii kyseiseen testiin soveltuvan oikean maailman henkilön, johon viitataan insinööriyössä testiaineistona. Testitapaus ohjeistaa selkokielellä askel askeleelta testaajalle testin kulun, suoritettavat

toimenpiteet ja tarkistettavat asiat. Esimerkiksi testaajaa voidaan ohjeistaa painamaan tiettyä painiketta ja sen jälkeen pyytää tarkistamaan, että näytöllä näkyy testitapauksessa mainitut elementit oikein – mahdollisesti samoin kuin tietokannassa. Testitapauksissa suoritettavat tarkistukset pohjautuvat luonnollisesti määrittelyihin.

Taulukko 2. Hyvin yksinkertainen kuvitteellinen testitapaus

Ennakkoehto	Henkilöä ei löydy järjestelmästä
Askel 1	Kirjaudu sisään järjestelmään
Askel 2	Kirjoita asiakkaan henkilötunnus kenttään ”Henkilötunnus”
Askel 3	Paina ”Hae”-painiketta.
Askel 4	Varmista, että ollaan siirrytty sivulle ”Henkilön tiedot”.
Askel 5	Varmista seuraavien elementtien arvot: <ul style="list-style-type: none"> - Tiedot haettu: (nykyinen päivämäärä) - Asiakkaan nimi: (asiakkaan nimi)
Askel 6	Kirjaudu ulos järjestelmästä painamalla ”Kirjaudu ulos”-linkkiä

4.3 Automaatiotestien toteuttaminen

4.3.1 Automatisoinnin työkalut

Insinööriyön käsittelemän sovelluksen regressiotestien toteuttamiseen käytetään Selenium-testaustyökalua, joka tarjoaa hyvin monipuolisen rajapinnan verkkosovellusten testaamiseksi. Seleniumin käyttämiseen on useita eri tapoja: kehittäjä voi käyttää joko omaa kehitysympäristöään tarjoavaa selainlisäosaa Selenium IDE:ä tai toteuttaa testitapaukset haluamallaan ohjelmointikielillä käyttäen Selenium-laajennusta. Selenium on virallisesti saatavilla seuraaville suosituimmille ohjelmointikielille, joita ovat Java, C#, Ruby, Python, JavaScript. Lisäksi saatavilla on useita kolmannen osapuolen toteutuksia muillekin ohjelmointikielille. [9.]

Selenium suorittaa komentoja suoraan selaimessa Selenium WebDriver -ajurin avulla. WebDriver tukee suurinta osaa moderneista selainohjelmista. Ohjelmointikielillä käytetään Selenium-kirjaston komentoja, jotka testitapausta suoritettaessa välitetään WebDriver -ajuria käyttämällä selaimelle. Selenium-kirjaston komennoilla voi esimerkiksi klikata verkkosivustolla näkyviä elementtejä tai tarkistaa niiden tekstejä – pääsääntö-

sesti mikä vaan selaimen toimenpide, joka onnistuu ihmiskäyttäjältä, onnistuu myös Seleniumilla. [10.]

```
public class SeleniumTest {

    @Test
    public void test() {
        WebDriver driver = new FirefoxDriver();

        driver.get("http://www.domain.tld");

        WebElement searchInput =
            driver.findElement(By.name("searchInput"));

        searchInput.sendKeys("Hakusana");

        searchInput.submit();

        String h1Text = driver.findElement(
            By.xpath("//div[@id='content']/h1")).getText();

        assertEquals("Value of H1 element was not as expected",
            "Etsi sivustolta", h1Text);

        driver.quit();
    }
}
```

Koodiesimerkki 1. Yksinkertainen Selenium-testi

Projektissa testattavan sovelluksen ohjelmointi tapahtuu Java EE -alustalla. Regressiotestausjoukon Selenium-testitapaukset ovat Java-pohjaisiksi toteutettuja JUnit-testejä, jotka sijaitsevat samalla SVN-tietovarastolla kuin itse sovelluskin. Näin testeissä voidaan hyödyntää osittain samoja Hibernate-tietomalleja, joita itse testattava sovelluskin käyttää. Tämä tukee modulaarisuuden periaatetta ja poistaa tarpeen tehdä tietomallin määritysten mahdolliset muutokset erikseen automaatiotesteihin.

Projektissa on hyödynnetty jo alusta lähtien automatisoitua regressiotestausta. Alun perin testien toteutus alustaksi valikoitui Selenium IDE, jonka puutteiden takia se vaihtui nopeasti skriptipohjaiseen Selenium RC:hen, joka tunnetaan nykyisin Selenium Serverinä. Testauksen kannalta tärkeiden silmukkaoperaatioiden puute Selenium IDE:ssä olisi ollut ylittävissä hyödyntämällä testitapauksissa osittain JavaScript-skriptausta, mutta automaatiotestit olisivat käyneet nopeasti liian vaikeasti ylläpidettäviksi. Seleni-

um IDE ei myöskään mahdollista lasilaatikkotestausta, jota Selenium Server -pohjaisia testejä toteuttamalla on mahdollista hyödyntää.

Ohjelmistoympäristönä käytämme avoimen lähdekoodin Eclipseä, joka mahdollistaa muun muassa virallisen Jenkins-integraation Eclipse Mylyn -käyttöliittymäläajennusta käyttämällä. Näin voimme seurata automaatiotestien Jenkins-ajojen tuloksia suoraan kehitysympäristössä: Saamme esimerkiksi konsolin lokitiedostot ja tulokset näkymään suoraan Eclipsen JUnit-näkymässä. Mylyniä käyttämällä voi myös ajaa testijoukot Jenkinsissä Eclipsen käyttöliittymän kautta. [11.]

4.3.2 Testitapauksesta automaatiotestiksi

Ensimmäinen vaatimus testitapauksen automatisoinniksi on se, että testi voidaan ajaa vaihe vaiheelta onnistuneesti myös manuaalisesti. Kun testitapaus valikoituu regresiotestijoukkoon automatisoitavaksi, on se jo aiemmin ajettu järjestelmätestauksen yhteydessä. Järjestelmätestauksen suorittanut henkilö merkitsee testitapauksen käsikirjoitukseen havaintonsa testin suorittamisesta, ja jos automatisoinnille ei havaita esteitä, voi testin ohjelmoinnin aloittaa. Jos esteitä kuitenkin esiintyy, eli esimerkiksi testitapauksen testaamaa toiminnallisuutta ei ole vielä täysin toteutettu, tai siihen on kaavailtu muutoksia, siirtyy automaatiotestin toteuttaminen myöhemmäksi.

Jokainen uusi automaatiotesti perii yhteisen kantaluokan (base class), joka sisältää testin ajamiseen tarvittavia konfiguraatioita ja esimerkiksi JUnitin setUp()- ja tearDown() (@Before- ja @After) -metodit. SetUp()-metodi suoritetaan ennen testin suorittamista, ja siellä tehdään jokaiselle testille yhteiset toimenpiteet, esimerkiksi alustetaan Seleniumin WebDriver ja asetetaan muuttujien arvoja. TearDown()-metodia kutsutaan taas testin suorittamisen jälkeen riippumatta siitä, menikö testi läpi vai ei. Tässä metodissa palautetaan testiympäristö alkuperäiseen tilaan, jossa se oli ennen testin suorittamista. Muun muassa sovellukseen testin aikana tehdyt käyttöäoikeusmuutokset palautetaan alkuperäisiksi.

Automaatiotestin toteuttaminen pohjautuen selkokiehiseen testitapaukseen on hyvin johdonmukaista: jokainen testitapauksen askel toteutetaan automaatiotestissä, kuten se toteutettaisiin myös manuaalisesti ajamalla. Modulaarisuutta toteuttaen jokainen suurempi prosessi pyritään toteuttamaan yhteyskäyttöiseksi. Yhteiskäyttöiset prosessit sijoitetaan omiin näyttökohtaisiin luokkiinsa, joita kutsutaan erityisen Singleton-mallilla

toteutetun luokan avulla. Modulaarisuuden hyödyntäminen vaatii vain hyvin vähän ylimääräistä työtä, mutta helpottaa uusien testien toteuttamista ja ylläpidettävyyttä [8, s. 397].

Lisäksi automaatiotestauksen käyttöön on syntynyt paljon apuluokkia, jotka tarjoavat metodeita esimerkiksi erimuotoisten lukujen tai päivämäärien helppoon vertailuun tai jotka helpottavat monimutkaisten tietokantakutsujen tekemistä. Tällaisia tietokantakutsuja ovat muun muassa sovelluksen taustalla pyörivien prosessien tilan seuranta, jota voidaan hyödyntää esimerkiksi tapauksessa, jossa halutaan prosessin päättyvän ennen uuden toimenpiteen, kuten sivunlatauksen ja siihen liittyvien tarkastuksien, tekemistä.

Satojen jo toteutettujen automaatiotestien myötä projektin tietovarastoon on kasvanut erittäin suuri määrä näitä yhteiskäyttöisiä metodeita ja luokkia, joita hyödynnetään jatkuvasti myös uusissa testeissä; kaikkia rutiininomaisia prosesseja ei tarvitse kirjoittaa erikseen joka testiä varten. Tällaisia prosesseja voi olla esimerkiksi sovellukseen kirjautuminen, hakemusten täyttäminen tai hakutoiminnon käyttäminen. Testitapauksessa voidaan kutsua useiden erillisten Selenium-komentojen sijaan näitä yksittäisiä metodeita suoraan näyttökohtaisista luokista. Tämä edistää testien johdonmukaisuutta, tekee uusien testien ohjelmoinnista nopeampaa ja helpottaa ylläpitämistä: mahdollisten järjestelmämuutosten jälkeen jokaisen automaatiotestin sijaan ainoastaan niiden kutsumat yhteiskäyttöiset metodit tulee päivittää ajan tasalle [2, s. 266].

Yhteiskäyttöisten prosessien lisäksi olemme myös luoneet oman Selenium-kirjastoa laajentavan luokkamme, jossa on ylikirjoitettu monia Selenium-komentoja soveltumaan paremmin juuri tämän sovelluksen testaamiseen, ja luokkaan on lisätty myös useita uusia WebDriverä hyödyntäviä metodeita.

JUnit-kirjasto tukee useita eri assertointimetoodeja, jotka mahdollistavat testin keskeyttämisen, jos jokin testitapauksen askel epäonnistuu. Näitä metodeita on esimerkiksi `assertNotNull(Object o)`, joka olettaa, että parametrina saatu objekti on määritelty, eikä siis null, eli tyhjä. Muita metodeita ovat muun muassa `assertEquals(Object expected, Object actual)`, joka vertaa kahden objektin vastaavuutta, ja `assertFalse(boolean condition)`, joka olettaa, että parametrina saatu ehto palauttaa epätosi, eli false. JUnitin assertointimetoodeja voidaan hyödyntää yhdessä Selenium-komentojen kanssa esimerkiksi siten, että testi keskeytetään, jos Selenium palauttaa selaimen olevan väärällä sivulla tietyn painikkeen painamisen jälkeen. [12.]

4.4 Automaatiotestien ajaminen

Jokainen automaatiotesti ajetaan useamman kerran läpi jo toteutusvaiheessa. Kun automaatiotesti valmistuu, siirretään se katselmointiin, jossa varmistetaan testin vastaavuus testitapauksen käsikirjoitukseen. Jos testitapauksessa havaitaan puutteita, palautetaan se kehittäjälle. Katselmoinnin läpäissyt automaatiotesti on valmis siirrettäväksi ajoon integraatiopalvelimelle.

Automaatiotestit ajetaan Jenkins-integraatiopalvelimella osana kokonaisuutta, joka käsittää tiettyjen iteraatioiden aikana toteutettujen toiminnallisuuksien testaamisen. Yksi iteraatio saattaa esimerkiksi testata, että sovelluksessa määritettävät käyttöoikeudet toimivat oikein, ja toinen taas, että kaikki järjestelmän tarjoamat dokumentit muodostuvat kuten on määritetty. Jokainen näistä iteraatioista kuuluu sovelluskehityksen tiettyyn vaiheeseen. Vaihe on iteraatioiden kattava kokonaisuus ja voi sisältää kymmeniä iteraatioita.

Automatisoidut regressiotestit tulevat useimmiten toteutukseen jo järjestelmätestausvaiheessa, mutta niiden jatkuva ajaminen alkaa vasta sovellusversion vaiheen lähestyessä hyväksymistestausta. Tällöin kaikki vaiheen testauslaajuuteen kuuluvat automaatiotestit ovat valmiina ja niille luodaan Ant-tiedostoihin omat target-määrittämisensä. Jenkins-palvelimella luodaan jokaiselle asiakkaalle iteraatiokohtaisesti omat ajokokonaisuudet projektinsa, jotka käsittävät kaikkien iteraation testien ajamisen Ant-tiedostojen määrittysten mukaisesti. Jenkinsillä testien ajaminen on hyvin yksinkertaista: konfiguroidun projektin käänös käynnistetään manuaalisesti verkkokäyttöliittymästä tai ajastetusti, ja testien suorituksen valmistuttua Jenkins näyttää testien graafisen tuloksen ja trendin.

```
<target name="tests" depends="compile">
  <junit printsummary="yes" haltonerror="false" haltonfailure="false" fork="yes">
    <classpath path="${lib.dir}">
      <path refid="classpath" />
    </classpath>

    <formatter type="xml" />

    <batchtest fork="yes" todir="${reports}">
```

```

        <fileset dir="test">
            <include
name="**/tests/iteration01/Test*.java"/>
        </fileset>
    </batchtest>
</junit>
</target>

```

Koodiesimerkki 2. Ant-target testien ajamiseksi

4.5 Automaatiotestien ylläpidettävyys

Automaatiotestien tapauksessa on syytä kiinnittää huomiota ylläpidettävyyteen heti alusta alkaen - ylläpitäminen on merkittävä kuluerä. Mitä suurempi määrä automatisoituja testitapauksia regressiotestijoukkoon kuuluu, sitä vaikeammin hallittavaksi kokonaisuus muuttuu. Testien määrän ja testikattavuuden välillä täytyy löytää sopiva tasapaino, jolloin testit ovat helposti ylläpidettävissä käytettävissä olevilla resursseilla kuitenkin regression ollessa tuloksellista. [7, s. 194-195.] Sama koskee automaatiotestien käyttämää testidataa tai -aineistoa, joka tässä tapauksessa voi tarkoittaa esimerkiksi olemassa olevien henkilöiden tietoja. Näitä tietoja käsitellään ei-selkokielisiksi salattuihin, jotta tiedot eivät yksilöisi ketään ja henkilö pysyisi testiajille tuntemattomana.

Myös hyvin toteutetut automaatiotestit vaativat elinkaarensa aikana ylläpitämistä. Sovellukseen ja sen määrittelyihin tehtävät muutokset vaikuttavat testitapausten ajamiseen ja tulokseen. Voikin olla, että testin ajo epäonnistuu ja dokumentaatiota tutkimalla huomataan, että syynä ovat muuttuneet määrittelyt. Tällöin automaatiotesti korjataan uusien määrittelyiden mukaiseksi. Usein samat sovellusmuutokset vaikuttavat moniin testeihin, mutta testien modulaaristen komentojen myötä jokaista testiä ei tarvitse korjata yksitellen. Syynä muutoksiin voi olla esimerkiksi puutteet määrittelyissä, testitapauksessa tai automaatiototeutuksessa, virheiden korjaus tai asiakkaan tekemät muutospyyntö.

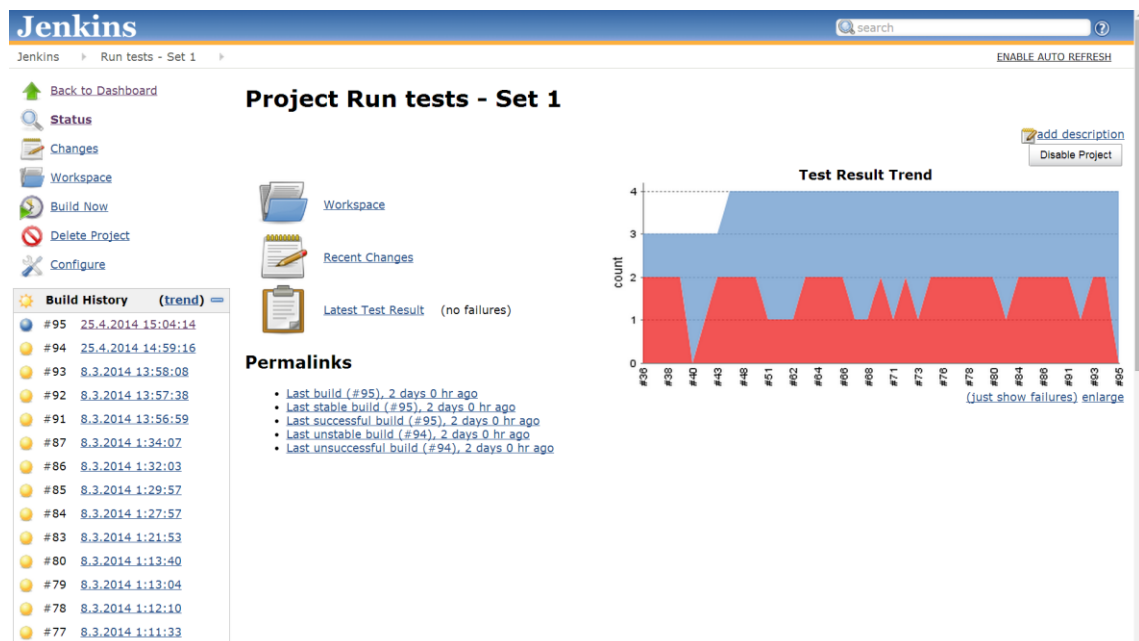
Sovelluksen tai testitapausten muutokset eivät ole ainoa syy testien korjaustarpeelle. Kuten jo mainittu, projektin regressiotestauksen testitapaukset perustuvat oikeiden henkilöiden salakirjoitettuihin tietoihin asiakkaan toimialalla. Tietojen muuttuessa, muut-

tuvat mahdollisesti myös askeleet testitapauksen suorittamiseksi. Aiheeseen perehdytään tarkemmin luvussa 5.1.

Automaatiotestejä päivitetään myös suorituskyvylisistä ja laadullisista syistä. Joissain tapauksissa saatetaan huomata Jenkins-palvelimen tuloksista, että testin suorittamiseen kuluu jatkuvasti huomattavan suuri määrä aikaa. Toisinaan syynä voi olla virhe järjestelmässä tai testitapauksessa, mutta usein ongelmat voidaan ratkaista testitapauksen optimoinnilla.

4.6 Tulosten seuranta ja luotettavuus

Iteraatiokohtaisten Jenkins-projektien piirtämät graafit esittävät iteraatioiden yleistilanteen yhdellä vilkaisulla. Graafeista käy nopeasti ilmi iteraatioiden toiminnallisuuksissa tai taustajärjestelmissä ilmenevät ongelmat heikentyneiden testien tulosten myötä. Jokaisen käännöksen erilliset tulokset kertovat epäonnistuneiden testien syyt yksityiskohtaisemmin.



Kuva 6. Jenkins tarjoaa graafeja tulosten seurantaan

Trendit eivät kerro suoraan verkkosovelluksen laadusta. Automaatiotestien skenaarioiden ollessa vaativia ja laajoja, kuten suurimmassa osassa projektin testitapauksista, ovat testit alttiita epäonnistumisille myös erilaisista satunnaisista syistä. Monet ylei-

simmistä projektin automaatiotestauksessa kohdatuista testien epäonnistumisista on jo ratkaisu.

Jenkinsin tarjoamia tuloksia on toisinaan vaikea hyödyntää sellaisenaan edellä mainituista syistä; automaatiotestien epäonnistumisyyden oikeellisuutta voi olla vaikea tulkitella ilman tarkempaa testikohtaista analyysia.

Analyysin helpottamiseksi testin epäonnistumiskohdassa tallennetaan kuvankaappaus palvelimelle. Jo kuvankaappauksesta saattaa nähdä viitteitä testin epäonnistumisen syyille, esimerkiksi sen, onko testi päätyntä oikealle sivulle, tai onko tiedot näytetty kyseisellä sivulla määritysten mukaisesti. Useimmiten testit tulee kuitenkin ottaa yksittäin tarkempaan tarkasteluun.

4.7 Automaatiotestauksessa havaitut virheet

Testaamisen tarkoituksena on ylläpitää sovelluksen laatua: pääkeino tähän on virheiden havaitseminen ja siihen liittyvien toimenpiteiden tekeminen. Virhe voi tarkoittaa sovelluksen toteutuksellisia eroja määrittelyihin nähden tai sovelluksessa havaittavia järjestelmävirheitä. [5, s. 257.]

Myös automatisoidulla regressiotestaamisella pyritään havaitsemaan testitapauksiin liittyvät virheet sovelluksessa. Prosessi virheiden havaitsemiseen ja käsittelyyn on kuitenkin erilainen kuin manuaalitestauksessa. Tästä johtuen automaatiotestin debuggaaminen ja virheiden syyn analysointi on pääsääntöisesti vaikeampaa kuin manuaalitestin tapauksessa. Manuaalitestin suorittaja tietää tasan tarkkaan, mitä oli tekemässä ja mitä testitapauksen askelta suorittamassa, kun testissä ilmeni virhetilanne. Automaatiotestin tulosten tarkastelijalla on käytössään vain konsolilogit ja JUnitin tarjoama asertointipointti, joka voi olla hyvinkin tarkka kuvaus virheestä tai huonoimmillaan hyvin epämääräinen. Paljon riippuu automaatiotestin laadullisesta toteutuksesta. [7, s. 196.]

Automaatiotestin suorituksen epäonnistuessa tarvitaan tapauskohtaista analyysia. Testin epäonnistumisen syyn selvittämiseksi tulee usein turvautua testitapauksen ainakin osittaiseen manuaaliseen suorittamiseen, jotta sovelluksen virhetilanteen voi itse todeta ja tarkemmat askeleet virheen havaitsemiseksi selkenevät. Jos syyksi testin epäonnistumiselle havaitaan sovellusvirhe tai eroavaisuudet määrittelyyn ja sovelluksen toi-

minnan välillä, raportoidaan virheestä projektin käyttämään vianseurantajärjestelmään. Tällaisia järjestelmiä ovat esimerkiksi Atlassian JIRA [13], IBM Rational ClearQuest [14] ja Bugzilla [15].

Havaitusta virheestä on hyvä tehdä kirjaus myös automaatiotestiin, jossa virhetilanne on havaittu, esimerkiksi tunnisteena assertointiviestiin. Näin seuraavalla ajokerralla testin tuloksista näkyy heti, että toiminnallisuutta vaivaa virhe, jos sitä ei ole ehditty korjata ennen tätä.

Virheen käsittely etenee projektissa sovitun prosessin mukaisesti. Kun virheeseen on tehty korjaus, pyritään järjestelmän toimivuus validoimaan ensiksi manuaalisesti. Myös automaatiotestin pitäisi nyt päästä tuon virhetilanteen yli; assertointi ei enää keskeytä testiä. Jos sama virhe kuitenkin toistuu myöhemmin uudelleen, jää se kiinni automaatiotestiin aiemmin tehtyyn assertointiin, ja assertointiviestissä mainittu tunniste helpottaa uuden tai toistuneen virhetilanteen selvittämistä.

5 Projektissa kohdatut automaation haasteet

Automatisoidun testausprosessin luotettavuutta projektissa heikentävät tietyt regressiotestausprosessissa kohdatut yleisesti esiintyvät haasteet. Haasteita kartoittamalla ja ratkaisuja ongelmiin löytämällä automaatiotestauksesta saadaan enemmän irti - samalla myös manuaalisen työn määrä vähenee tulosten parantuessa.

Tässä luvussa kartoitetaan sovelluksen automatisoidun regressiotestausprosessin haasteita ja esitellään prosessille tyypillisiä piirteitä, jotka tulee ottaa huomioon jo testausta suunnitellessa. Projektikohtaisiin haasteisiin esitellään ratkaisuja teoriatasolla ja toteutuksellisesti luvussa 6.

5.1 Testiaineisto ja sen muutokset

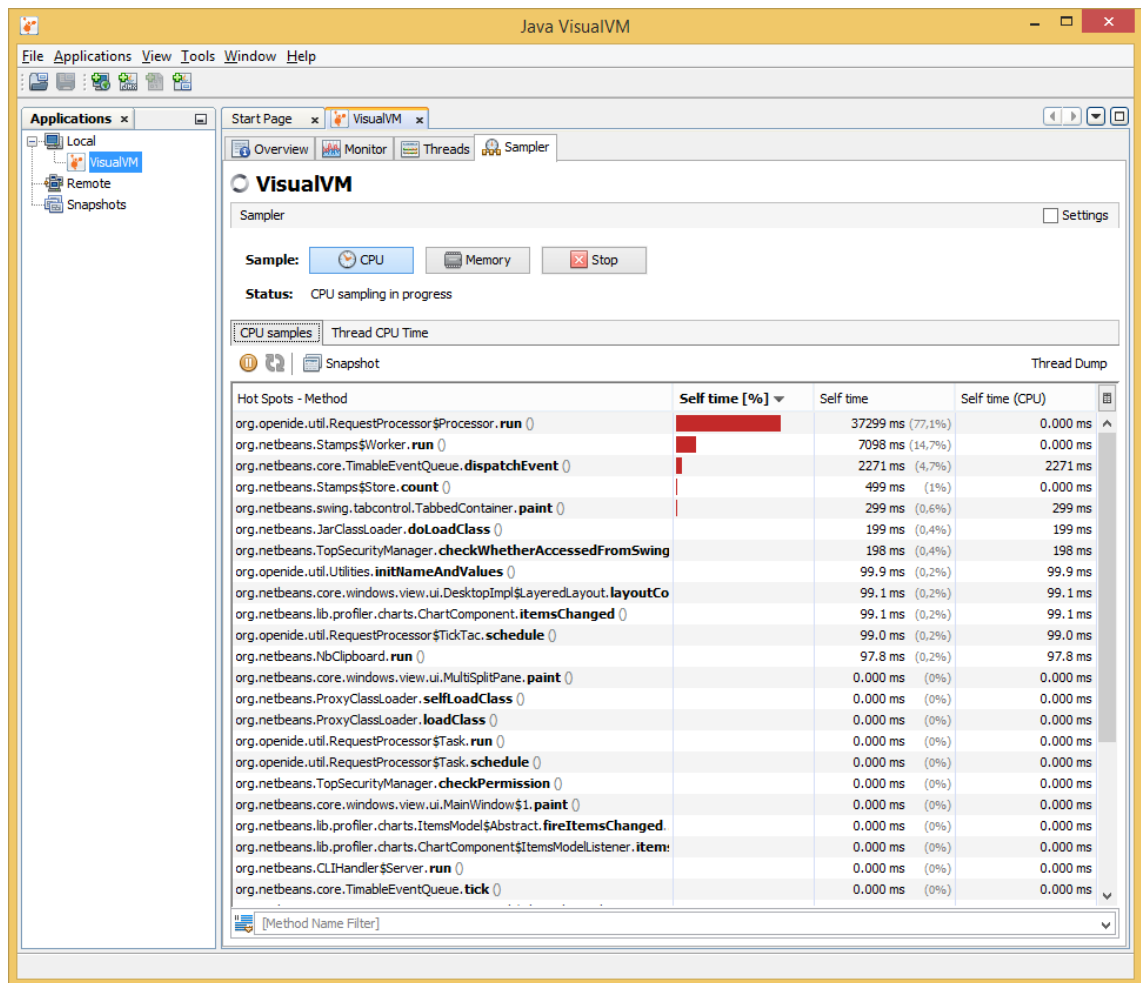
Testiaineisto käsittää terminä kaiken testaamisessa käytetyn materiaalin eli testitapaukset, syötteet ja lähtöarvot, tiedon, dokumentaation ja oletetun lopputuleman [7, s. 144]. Tässä termillä testiaineisto viitataan testaamisessa käytettyyn tietoon, eli henkilöihin ja heidän ominaisuuksiinsa.

Testitapauksissa, joissa lähtökohtana on tarkastella sovelluksen toimintaa aitoja muuttuvia tietoja vastaan, tulee ottaa huomioon monia asioita. Testattavan sovelluksen tapauksessa valtaosa testeistä käsittelee aitoja henkilöitä. Järjestelmässä ja sen ulkopuolella suoritetaan kyseisille henkilöille erilaisia operaatioita, kuten esimerkiksi erilaisien lomakkeiden käsittelyprosesseja. Käsittelyprosessin askeleet ovat hyvin riippuvaisia testiin käytetystä testiaineistosta, ja henkilölle tapahtuvat todelliset muutokset voivat usein vaikuttaa testin suorittamiseen kriittisesti. Voikin olla niin, että henkilön taustatietojen muutokset, kuten työpaikan tai asuinpaikkakunnan vaihtuminen, vaikuttavat käsittelyprosessiin, eikä se enää etenekään kuten testitapauksessa on oletettu. Todellisuudessa henkilö on usein edelleen täysin soveltuva testitapauksen suorittamiseksi, mutta testitapauksessa tulisi suorittaa uusia askeleita, joita on hyvin hankala ennakoida etukäteen. Automatisoidun regressiotestauksen kannalta tämä testiaineiston tietojen muuttuminen on ollut yksi suurimpia haasteita projektissa, ja siten havaitut virheet pois lukien suurin yksittäinen syy testien epäonnistumiselle.

5.2 Automaatiotestien laadulliset ja suoritusajalliset tekijät

Suuri vaikutus automatisoituun regressiotestausprosessiin on myös testien laadullisilla ja suoritusajallisilla tekijöillä. Testien ollessa toteutettu modulaarisesti, ja samoja operaatioita käytettäessä monia kertoja useissa testeissä, kumuloituvat näennäisesti pienetkin viiveet yksittäisissä metodeissa. Esimerkiksi, yhdessä testitapauksessa voidaan tehdä henkilöhaku kymmeniä kertoja – jos henkilöhaun toteutus pysäyttää säikeen suorituksen turhaan useiksi sekunneiksi, kumuloituvat nuo harmittomilta tuntuvat sekunnit yksittäisissä testeissä minuuteiksi ja koko testijoukossa jopa useiksi tunneiksi.

Modulaarisuuden johdosta tämän kaltaiset virheet on kuitenkin helppo korjata, kunhan ne pystytään ensiksi havaitsemaan. Suoritusajallisten vaikutusten havaitseminen testeissä voi usein olla vaikeaa. Suoritusajan tulkintaan voidaan hyödyntää esimerkiksi profilointia. Java JDK:n tarjoama graafinen VisualVM-työkalu soveltuu erinomaisesti tähän JUnit-testien tapauksessa. VisualVM:n reaaliaikaisesti muodostamasta listauksesta näkee, kuinka paljon kunkin sovelluksen, tai tässä tapauksessa testitapauksen, metodit ovat vieneet suhteellista suoritusaikaa.



Kuva 7. Java VisualVM -profilointityökalu

Automaatiotestaajia on projektissa useita, joten myös ohjelmointitavat vaihtelevat. Kaikkiin asioihin ei ole vakiintunut selkeitä käytäntöjä, vaan jokainen ohjelmoija tekee toteutuksensa oman näkemyksensä mukaisesti. Tällä voi olla laadullisia vaikutuksia automaatiotesteihin. Yksi olennaisimmista projektin automaatiota koskevista laadullisista tekijöistä on oikeanlainen ja kuvaava assertointiviesti epäonnistumisen syyksi. Tällä hetkellä tapa toteuttaa assertointi on hyvin kirjavaa, ja joissain tapauksissa jopa epäselkeää.

5.3 Testien häiriönsieto

Automaatiotestausprosessissa suureksi tekijäksi tulosten kannalta on noussut testien häiriönsietokyky (eng. robustness). Häiriönsietokyvyllä tarkoitetaan testin kykyä testata haluttua asiaa ilman satunnaisten häiriöiden vaikutusta. Tällaisia häiriöitä voi olla esimerkiksi yhteysongelmat tai automaatiotestin toteutuksesta johtuvat epävakaudet, jois-

ta johtuen esimerkiksi sivunlatauksen valmistumista ei odoteta ennen seuraavien toimintojen suorittamista. Häiriönsietokyky käsittää myös terminä sen, että testien yksittäiset sovelluksessa havaitut virhetilanteet eivät saisi ketjureaktiomaisesti rikkoa useita testejä. Käytännössä kuitenkin usein ketjureaktioiden syntymistä ei voi täysin estää. [7, s. 224.]

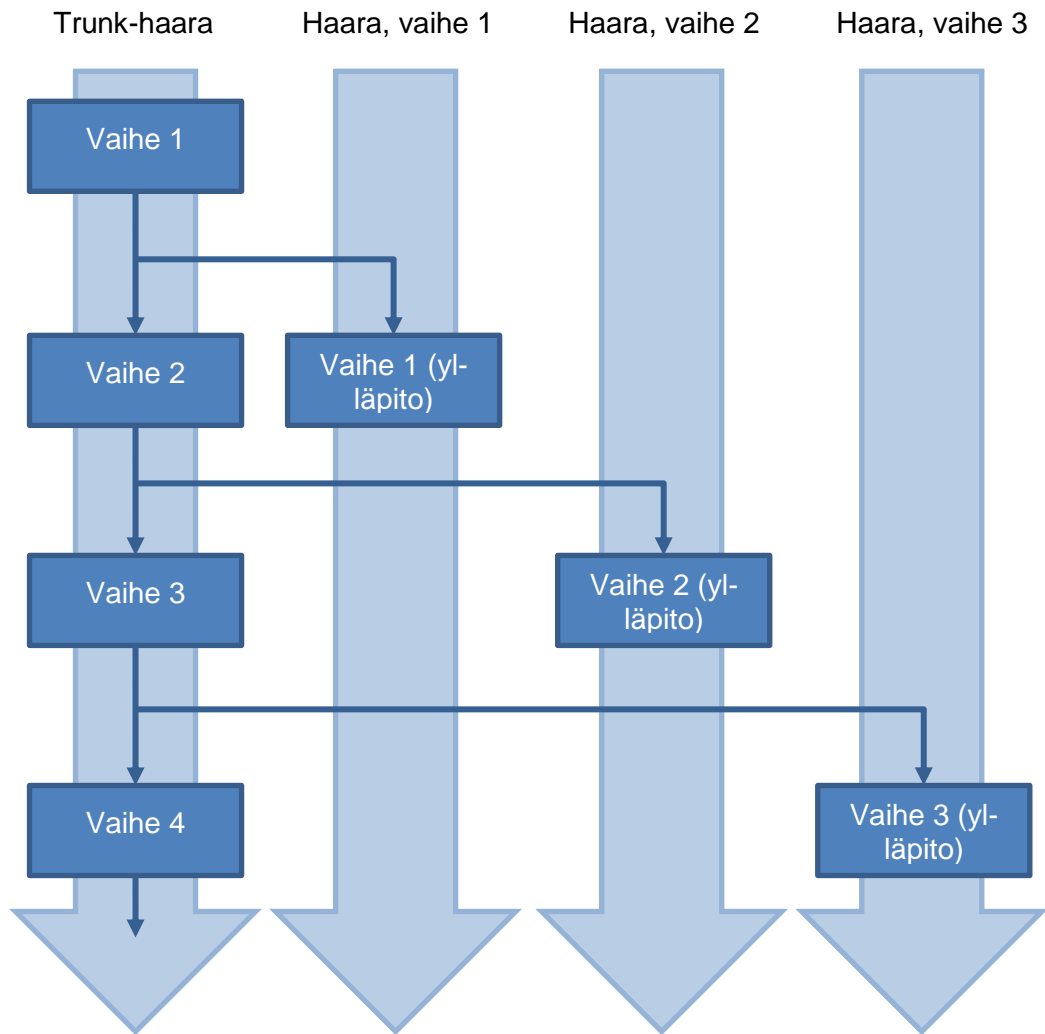
Testin häiriönsietokyvyn puute voi aiheuttaa suuria ongelmia testien ajamisessa ja tulosten analysoinnissa. Ei-oleelliseen virheeseen epäonnistuneen testin tulos ei kerro testaajalle mitään, vaan testi on ajettava toiseen kertaan. Jos häiriönsietokyvyssä ilmenneitä puutteita ei korjata, voidaan sama ongelma kohdata uudelleen.

5.4 Sovelluksen arkkitehtuuri ja haarautuminen

Testattava sovellus on loogisesti versioitu, eli sen arkkitehtuuri on haarautunut (eng. branching) useaan eri versioon. Omat kehityshaaransa (eng. branch) on projektin eri asiakkaille, mutta myös vaiheille. Eräs vaihe voi olla esimerkiksi asiakastestauksessa ja ylläpidossa, kun samanaikaisesti toinen uudempi on kehityksessä. Haarautumisella on vaikutuksia myös automatisoituun regressiotestaukseen.

Käytännössä haarautuminen tarkoittaa automaatiotestauksen kannalta usein sitä, että osa myös automaatiotestaukseen kuuluvista luokista on myös haarautettava. Projektissa automaatiotestaus hyödyntää muun muassa itse sovelluksen Hibernate-malleja, ja kehityshaarojen suurten erojen takia jokainen testattava haara sisältää omat kopionsa automaatiotesteistä. Yksinkertaistettuna, toisen haaran automaatiotestejä ei voi ajaa toisesta haarasta käännettyä sovellusta vasten koodipohjan erojen takia.

Automaatiotestaajille haarautuminen näkyy siinä, että jokaisessa haarassa olevat automaatiotestiluokat on pidettävä ajan tasalla; testien muutokset ja korjaukset on toteutettava, tai yhdistettävä (eng. merge) jokaiseen kehityshaaraan. Luonnollisesti kaikkia muutoksia ei toteuteta kaikkiin haaroihin, sillä osa muutoksista on asiakas- tai vaihekohtaisia.



Kuva 8. Vaiheiden haarautumista havainnollistava kuva

Yksinkertaistetussa kuvassa 8 esitetään sovelluksen haarautumista eri vaiheisiin. Jokainen nuoli kuvaa yhtä haaraa, jossa suoritetaan regressiotestausta. Sovellusversioon tapahtuu muutoksia jokaisessa haarassa, esimerkiksi asiakkaan muutospyyntöjen tai ylläpidollisten syiden takia. Muutosten vaikutukset automaatiossa on tehtävä samaten jokaisessa haarassa. Insinööryön käsittelemässä sovelluksessa trunk-haara sisältää sovelluksen uusimman vaiheen koodit, ja muut haarat ovat siitä eriytettyjä versioita.

Eri vaiheiden määrittämisessä on suuria eroja, jotka vaikuttavat myös automaatiotestien toteutukseen. Nämä eroavaisuudet vaikeuttavat osaltaan muutosten yhdistämisen suorittamista, mutta ongelmaa on pyritty vähentämään toteuttamalla vaiheriippuvaiset muutokset testeissä omiin ulkoisiin luokkiinsa.

5.5 Ulkoiset tekijät

Sovellus käyttää taustallaan useita ulkoisia rajapintoja. Rajapintojen kehitys- ja huolto-työt, yhteysongelmat tai palauttavat virheet vaikuttavat sovelluksen toimivuuteen ja siten myös testaamiseen. Automaatiotestauksessa rajapintojen odottamaton toiminta voi aiheuttaa suuria haasteita.

Manuaalitestauksessa sovellusta havaitaan välittömästi, mikäli jokin rajapinta palauttaa virheitä, joiden syy voi olla moninainen: Toisinaan palautuvista virheistä on ilmoitettu rajapintojen ylläpidosta etukäteen, ja toisinaan ne johtuvat esimerkiksi hetkellisistä teknisistä ongelmista tai meneillä olevista päivityksistä. Manuaalitestauksen tapauksessa sovelluksen osa-alueen testaaminen keskeytetään, kunnes rajapinnat mahdollistavat taas testaamisen jatkumisen.

Automaation kannalta rajapintojen aiheuttamia ongelmia ei välttämättä heti huomata. Projektissa automaatiotestit ajetaan aiemmin kuvaillulla tavalla integraatiopalvelimella, eikä siten niiden tulosta seurata reaaliajassa. Rajapinnoista aiheutuvien virheiden havaitsemiseen voikin mennä useita tunteja ja käytännössä tänä aikana ajettut automaatiotestit ovat ajettu turhaan.

Osa sovelluksen käyttämisestä rajapinnoista on päällä vain tietyn osan päivästä; yöaikaan tai viikonloppuisin ei kutsuja näihin rajapintoihin voi tehdä. Tämä hankaloittaa automatisoitujen regressiotestien ajamista: käytännössä voimme ajaa suuren osan testeistä vain toimistoaikoina, sillä ne vaativat pääsyä ulkoisiin rajapintoihin. Tämä aiheuttaa haasteita etenkin silloin, jos testien ajamisessa havaitaan muita ongelmia, joista johtuen kokonaisia iteraatioita pitäisi ajaa useaan kertaan uudelleen.

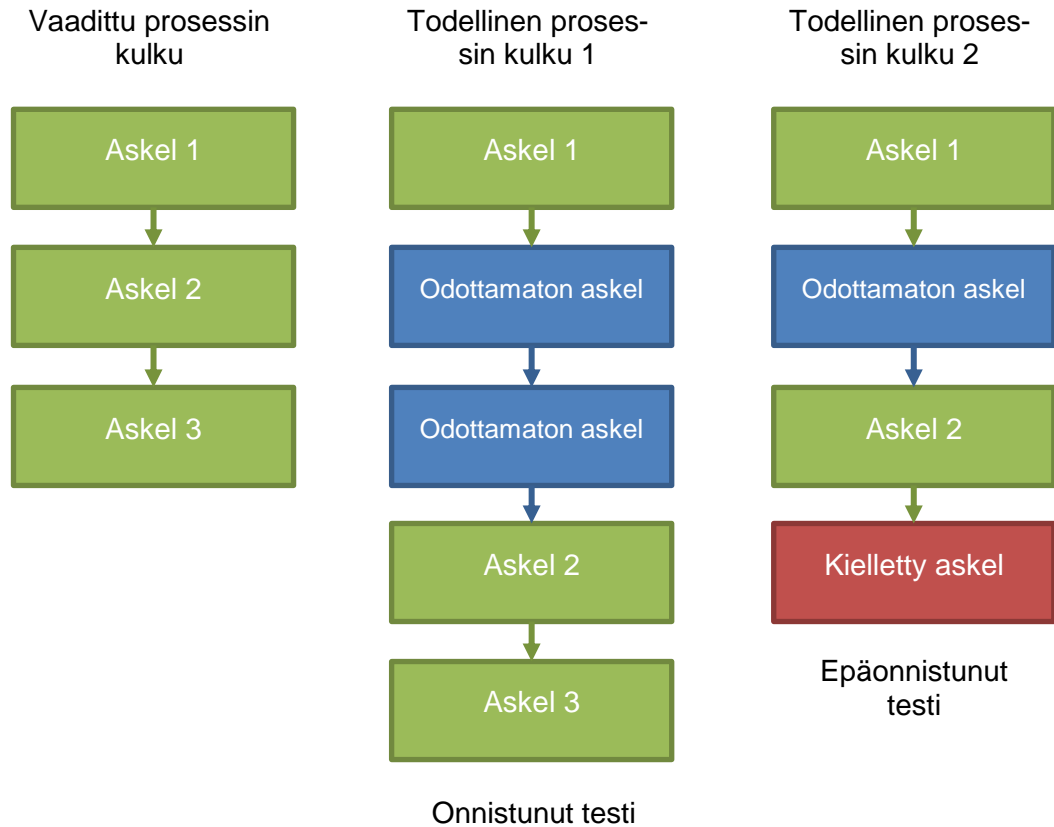
6 Automaation kehittäminen projektissa

Tässä luvussa syvennytään regressiotestausprosessin kehittämiseen projektin näkökulmasta edellisessä luvussa tehdyn kartoituksen mukaisesti. Luvussa esitellään insinööriyden aikana testausprosessissa havaittuihin haasteisiin löydettyjä ratkaisuja ja niiden toteutuksia. Lukua seuraa lyhyt katsaus automaatiotestien tulosten vertailusta ja raportoinnista.

6.1 Sopeutuminen testiaineistojen muutoksiin

Testiaineistojen muuttumisen ongelmaa on ratkottu uuden loogisesti toimivan proseduurin toteuttamisen kautta. Tämä proseduri on suunniteltu ajamaan testin kriittisimmät kohdat siten, että kaikki testitapauksessa mainitut operaatiot suoritetaan ja niiden oikeellisuus varmistetaan. Samalla myös testitapauksen ohella havaittavat testin kannalta odottamattomat ja merkityksettömät askeleet kulussa suoritetaan automaattisesti ilman, että niiden ilmeneminen keskeyttää koko testin epäonnistuneena. Käytännössä tämä tarkoittaa sitä, että proseduurille annetaan askeleet siitä mitä halutaan ja myös siitä, mitä ei haluta testitapauksen suorittaessa tapahtuvan. Testitapauksessa voidaan esimerkiksi vaatia, että testin aikana nousee huomautus annettujen tietojen tarkistamiseksi ja lopulta vahvistamiseksi, mutta huomautusta, jossa ilmoitetaan sovellusvirheestä, ei haluta nähdä.

Sellaisten huomautusten, jotka eivät vaikuta oleellisesti testitapauksen suorittamiseen, käsittelyyn ei oteta yksittäisessä automaatiotestissä kantaa. Tällaisia voivat olla huomautukset, joissa pyydetään tarkastamaan testiaineiston muuttuneita tietoja. Näille testin kannalta ei-merkittäville huomautuksille on proseduurin toteutuksessa olemassa omat toimenpiteensä, jotka suoritetaan, jotta prosessi pääsee etenemään. Proseduurille voi myös antaa ohjeita siitä, kuinka tietyt ongelmat tarpeen tullen selvitetään, riippumatta siitä kohdataanko niitä ollenkaan koko prosessin aikana.



Kuva 9. Proseduurin testiaineistojen muutoksiin sopeutumiseksi

Proseduurin toimintaa on pyritty selventämään kuvassa 9. Ensimmäisessä sarakkeessa esitetään toivottu prosessin kulku, ja muut sarakkeet esittävät mahdollisia todellisia prosessin kuluja. Ensimmäinen todellisista prosessinkuluista ei johda testin epäonnistumiseen, vaikka sen aikana nousee odottamattomia huomautuksia, sillä nämä ovat testin kannalta merkityksettömiä. Toinen esitetyistä kuluista sisältää askeleen, joka on määritetty proseduurin asetuksissa kielletyksi; tämä johtaa heti testin epäonnistumiseen. Ilman proseduurin hyödyntämistä molemmat näistä testeistä johtaisivat epäonnistumiseen heti ensimmäisen odottamattoman askeleen kohdatessa.

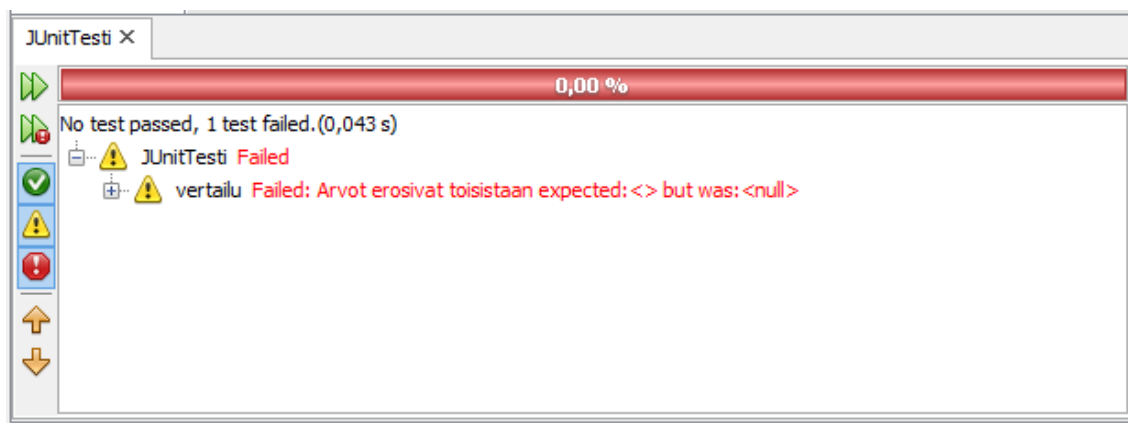
6.2 Modulaarisuuden ja laadun parantaminen

Projektin regressiotestauksen automaation modulaarisuuden taso on pääosin hyvä. Joissain tapauksessa yhteiskäyttöisyyden lisäys kuitenkin toisi eheyttä testaamiseen ja etenkin tulosten analysointiin.

Yksi suuria haasteita automaation tapauksessa on ollut testien puutteellinen assertointien käyttäminen, tai vielä useammin assertoinnin tapahtuessa annettavan kuvauksen epäselvyys. Puutteelliset assertoinnit voivat vaikuttaa testin epäonnistumiseen siten, että ajon tuloksista saatavat tiedot eivät ole riittävän kuvaavia tai esimerkiksi vääriä. Joissain tapauksissa voi esimerkiksi näkyä testin epäonnistumisen syynä tietyn elementin puuttuminen ruudulla tai prosessin väärä tila. Todellinen syy usein saattaakin olla puutteellisissa assertoinneissa. Toisin sanoen, jokin testitapauksen askelista on epäonnistunut jo aikaisemmin, mutta automaatiotestissä ei ole lisätty tarpeellisia assertointilauseita, joilla testitapauksen suorittaminen keskeytettäisiin jo tuolloin. Konkreettinen esimerkki tällaisesta tapauksesta on jonkun suuren tietoryhmän, kuten tietokannan rivien tarkastaminen kerralla. Tiedot saatetaan tarkistaa testissä silmukkaa hyödyntämällä, ja jonkun tiedon poiketessa odotetusta arvosta asetetaan boolean-arvoinen muuttuja epätodeksi. Vasta kaikkien tarkistusten jälkeen tarkistetaan asertoimalla, että tuo muuttuja ei ole asetettu epätodeksi. Virheen tapauksessa automaatiotestin assertointiviesti kertoo ainoastaan, että jokin näistä tarkastuksista epäonnistui, mutta tarkemman syyn selvittämiseksi on testi suoritettava vaihe vaiheelta uudestaan.

Toinen havaittu haaste assertointeihin liittyen projektin tapauksessa on se, että JUnitin assertointimetodeita ei hyödynnetä parhaalla mahdollisella tavalla. Usein kahta objektia, esimerkiksi tietokannan tekstiä näytöllä näkyvään tekstiin verratessa, vertailu toteutetaan niin kutsutun avustajaluokan metodilla, joka vertailee kahden merkkijonon vastaavuutta toisiinsa ja palauttaa tosi tai epätosi riippuen siitä, ovatko merkkijonot samat. Tuolle metodille assertointi tehdään `assertTrue(boolean condition)` -metodilla, jolloin testin epäonnistuessa näytetään vain ilmoitus, että arvot erosivat toisistaan, muttei sitä, miten ne erosivat. Testin tuloksesta ei voi näin päätellä, epäonnistuiiko testi esimerkiksi kirjoitusvirheen takia tai edettiinkö prosessissa väärin.

Vaihtoehtoisesti kahden objektin, tai tässä tapauksessa merkkijonon, vertailun voisi tehdä JUnitin `assertEquals(String expected, String actual)` -metodilla, joka kertoisi suoraan testin epäonnistuessa, miten metodille annetut parametrit eroavat toisistaan. Syy miksei alun perin tätä metodia ole hyödynnetty automaatiotesteissä, on se, että avustajaluokan merkkijonoja vertailevat metodit osaavat verrata myös määrittelemättömiä arvoja; tietokannasta palautuva null (olematon arvo) tulkitaan samaksi kuin näytöltä haettu tyhjä merkkijono. Assertointimetoja käyttämällä nämä kaksi tulkitaan eri merkkijonoiksi, kuten ohjelmoinnin perussääntöjen mukaisesti kuuluukin: Null, eli olematon, on eri asia kuin tyhjä merkkijono, joka on olemassa ja määritelty tyhjäksi.



Kuva 10. Olemattoman ja tyhjän merkkijonon vertailu assertEquals()-metodilla

Ylläkuvattuun ongelmaan on monta ratkaisua. Yksi mahdollisuus olisi käsitellä assertEquals()-metodin parametrit siten, että olematon arvo tulkitaan olevan määritelty tyhjäksi. Tähän voi hyödyntää esimerkiksi Apachen tarjoamaa StringUtils-luokkaa [16].

Projektin automaatiossa olemme pyrkineet ratkaisemaan assertointiin liittyvät ongelmat toteuttamalla uuden avustajaluokan arvojen vertailua ja objektien assertointia varten. Tämä AssertHelper-niminen apuluokka tukee assertointia vastaavasti kuin JUnitin tarjoamat metodit, mutta niihin on tehty testausautomaation kannalta tärkeitä muutoksia. Esimerkiksi ylläkuvattuun arvojen vertailuun, jossa olematon ja tyhjä halutaan tulkita käytännön syistä samaksi, on luokassa metodinsa, joiden käyttäminen on yksinkertaista. Parametrina annetaan viesti, joka näytetään jos vertailu epäonnistuu, havaittu ja oletettu arvo. Mahdollinen null alustetaan metodin sisällä tyhjäksi merkkijonoksi ja muunnetuilla arvoilla kutsutaan JUnitin alkuperäistä metodia.

AssertHelper-luokan toteuttamisella on myös muita hyötyjä. Voimme toteuttaa yksittäisiä assertointimetoodeja sellaisille tapahtumille, jotka ovat yleisiä useissa testitapauksissa. Näin voidaan yhtenäistää esimerkiksi testien epäonnistumisten yhteydessä näytettyjä kuvauksia staattisiksi määritellyin merkkijonoin.

Myös yhteiskäyttöisiin metodeihin on tehty automaation edetessä jatkuvasti laatua ja suorituskykyä parantavia muutoksia. Esimerkiksi aiemmin näytön piirtämisen ajoittainen hitaus on saattanut aiheuttaa sen, että testi epäonnistuu yrittäessään klikata elementtiä, jota ei vielä näy ruudulla vielä käynnissä olevan AJAX-pyyntönsä johdosta. Seleniumin vakiokomennot eivät tarjoa mahdollisuutta odottaa dynaamisten AJAX-pyyntöjen valmistumista - ainoastaan sivulatausten odottamiselle löytyvät omat komen-

tonsa. Muun muassa tähän liittyvät ongelmat on ratkaistu laajennusluokalla. Tässä tapauksessa uudelle sivulle siirtymisen jälkeen kutsutaan testeissä oman laajennusluokan metodia, joka odottaa sivunlatauksen valmistumista. Tämä metodi kutsuu ensiksi Seleniumin vakiokomentoa sivunlatauksen odottamiseksi ja sen jälkeen sovelluskoh-
taisesti toteutettua komentoa, joka osaa odottaa myös kaikkien tarpeellisten AJAX-
pyyntöjen valmistumista.

6.3 Yhtenäinen ohjeistus ja kommunikaatio

Kuten muussakin sovelluskehityksessä, myös automaatiotestauksen toteutuksessa on tärkeää noudattaa ohjelmointikäytäntöjä. Yhtenäisten käytäntöjen käyttäminen tukee pyrkimystä parempilaatuiseen koodiin, millä on tärkeä rooli parantaa muun muassa koodin ylläpidettävyyttä, uudelleenkäytettävyyttä, luettavuutta ja ymmärrettävyyttä. [17, s. 464-465.]

Sovelluskehityksessä yleisesti käytettyjen ohjelmistokäytäntöjen hyödyntämisen ohella on myös tärkeää sopia tapauskohtaiset käytännöt, joita automaatiotestaustiimi noudattaa uusien testien ohjelmoinnin tai vanhojen ylläpitämisen yhteydessä. Näitä käytäntöjä ovat muun muassa projektissa sovitut nimeämiskäytännöt: esimerkiksi testiluokkien nimiä voi edeltää sana "Test" ja yhteiskäyttöisten näyttöluokkien nimiä sana "Screen". Testin nimessä voi olla hyvä ilmaista myös esimerkiksi testattavan käyttötapauksen tai näytön tunniste, jolloin ulkopuolinenkin näkee jo testin nimestä yleiskuvauksen siitä, mitä kyseisessä automaatiotestissä testataan.

Muita yhtenäisyyden kannalta tärkeitä asioita on muun muassa se, että uudet luokat tai muut tiedostot sijoitetaan oikeisiin Java-paketteihin tai tiedostopolkuihin: rutiininomaiset prosessiluokat luodaan esimerkiksi flow-nimiseen pakettiin, josta niitä voidaan tarvittaessa käyttää helposti myös muissa testeissä.

Koko automaatiotestaustiimin on tärkeää noudattaa sovittuja käytäntöjä, jotta automaatiotestauksen laatu ja testien häiriönsietokyky pysyy korkeana. Käytännöistä poikkeaminen voi johtaa testien pirstaloitumiseen esimerkiksi siten, että joitain yleisiä testauksen toimintoja toteutetaan turhaan useita kertoja testikohtaisesti. Jos jokin tavoista todetaan virheelliseksi tai epävakaaaksi, joudutaan kaikki testikohtaiset toteutukset muuttamaan erikseen, kun yhteiskäyttöisiä komponentteja hyödyntämällä olisi muutos yh-

teen paikkaan riittänyt. Tällaisten erillisten komponenttien vakauden seuranta ja virheiden havainnointi on myös hankalaa.

Ohjeistuksen puute tai siitä poikkeaminen voi aiheuttaa ongelmia. Esimerkiksi tässä projektissa eräässä testikohtaisesti toteutetussa metodissa kohdattiin toistuvasti päättymätön silmukka. Metodissa odotettiin tietyn tiedoston luomista palvelimen tiedostorakenteeseen, mutta tiedostoa ei sovelluksen konfiguraatiovirheestä johtuen koskaan ilmestynyt oletettuun sijaintiin. Syynä loppumattomaan silmukkaan oli yksinkertainen virhe toteutuksessa, jossa ei ole otettu huomioon, että tiedostoa ei välttämättä koskaan ilmestykään; toisin sanoen silmukkaa toistettiin niin kauan kunnes tiedosto oletettavasti löytyisi halutusta sijainnista. Tämä aiheutti testien ajamisessa sen, että koko sen hetkinen ajo keskeytyi päättymättömäksi ajaksi tuohon yksittäiseen testiin, kunnes ajon kävi manuaalisesti pysäyttämässä. Jos tiedostorakenteen seuranta olisi testissä toteutettu kutsumalla loogisemmin suunniteltua yhteiskäyttöistä metodia, joka oli jo olemassa, ei ongelmaa olisi ilmennyt.

Ohjeistuksen ohella myös testaustiimin välinen kommunikaatio on tärkeää. Projektin tapauksessa automaatiotestaajia sijaitsee useissa eri toimipisteissä, eli henkilökohtaista kanssakäymistä kaikkien testaajien välillä ei ole. Puutteellinen kommunikaatio voi johtaa siihen, että eri toimipisteiden työntekijät selvittävät samaa automaatiossa kohdattua ongelmaa tai analysoivat esimerkiksi samoja testitapauksia samaan aikaan.

Kommunikaatiota automaatiotestaustiimin välillä on lisätty nimittämällä selkeät vastuuhenkilöt eri toimipisteiden välille. Nämä vastuuhenkilöt toimivat tiiviissä yhteistyössä ja pitävät toiset vastuuhenkilöt kartalla oman toimipisteensä työntekijöistä. Olemme myös ottaneet käyttöön koko automaatiotestaustiimille yhteisen Microsoft OneNote -muistiinpanosovelluksen, jota kaikki testaajat voivat täyttää reaaliajassa. Sovellukseen kirjataan epäonnistuneiden testien analyysit ja ongelmat reaaliajassa.

6.4 Testien automaattisen ajamisen kehittäminen

Projektin nykytilassa automatisoidun regressiotestauksen testit ajetaan integraatiopalvelimella. Automaatiotestejä sisältävät testiprojektit käynnistetään Jenkinsin verkkokäyttöliittymästä. Automaatiotestit testaavat usein toiminnallisuuksia siten, että yksittäisen testin suorittaminen vaikuttaisi myös toisen testin suorittamiseen esimerkiksi siten, että testi saattaa muuttaa suorituksen aikana sovelluksen käyttöoikeuksia, tai omistaa

samoja testiaineistoriippuvuuksia kuin muut testit. Tästä johtuen samoja testiympäristöjä vasten ei ajeta useita testejä samaan aikaan.

Nykyisin Jenkins on konfiguroitu siten, että vain yhden ympäristökohtaisen iteraation suorittaminen sallitaan kerralla. Jos muita iteraatioita liipaistaan käyntiin, päätyvät ne jonoon odottamaan vuoroaan, kunnes edellinen testi-iteraatio on suoritettu loppuun.

Testien samanaikaista ajamista voidaan kehittää siirtymällä iteraatioperusteisista jaotteluista loogisimpiin testiaineisto- ja toiminnallisuusriippuvaisiin jaotteluihin. Tämä tarkoittaisi sitä, että automaatiotestejä ei suoritettaisi kerralla iteraatioittain, vaan toiminnallisuuksittain. Tällöin useita toiminnallisuusryppäitä voisi suorittaa testausympäristössä samanaikaisesti ilman, että ajettavat testit vaikuttaisivat toistensa suoritukseen.

Yllä kuvatun muutoksen toteutus vaatii paljon resursseja, mutta säästää merkittävästi aikaa koko testijoukkojen ajamisen tapauksessa. Vaikutus korostuu entisestään, kun projekti etenee ja testitapausten määrä kasvaa. Mahdollisuus suorittaa koko testijoukko nopeammin alusta loppuun parantaa myös automatisoidun regressiotestauksen häiriönsietoisuutta: testijoukot ehdittäisiin ajamaan helpommin täydellisenä loppuun, vaikka ulkoisissa järjestelmissä tai sovellusversion tilassa havaittaisiin ongelmia.

Ulkoisissa järjestelmissä vaikuttavien ongelmien havaitsemiseksi on suunnitelmissa toteuttaa erillisen rajapintojen testaustyökalu, joka testaa kaikkien automaatiotesteihin vaadittavien ulkoisten rajapintojen toimivuuden ennen kaikkien testien automaattista ajamista. Tällöin rajapintariippuvaisten testien ajamiseen ei tarvitsisi käyttää resursseja tilanteissa, jolloin testien epäonnistumiset eivät olisi päteviä ulkoisista syistä.

7 Automaatiotestien tulokset

Luvussa käydään läpi yksityiskohtiin syventymättä automaatiotestien vertailuvaihe ja esitellään automaatiotestien analysoinnin tulevaisuuden näkymiä projektissa. Lisäksi luvussa esitellään insinööriyön aikana toteutettu sovellus tulosten analysoinnin automatisoimiseksi ja manuaalisen työn vähentämiseksi.

7.1 Manuaalinen työ

Automaatiotestauksen tarkoituksena on kasvattaa testauksen kattavuutta ja laatua samalla pienentäen manuaalisen työn tarvetta tulosten saavuttamiseksi. Testiautomaatio toteutetaan käytännössä siten, että kaikki testaaminen, johon menisi manuaalisesti enemmän aikaa ja vaivaa kuin automaation keinoin, pyritään automatisoidaan. Testejä, jotka voisi suorittaa manuaalisesti pienemmin resurssein koko sovelluksen kehityskaaren ajan, ei ole järkevää automatisoida. [18; 6, s. 229.]

Valmiilla automaatiollakaan ei päästä kokonaan eroon manuaalisesta työstä. Automatisoidussa regressiotestauksessa kohdatut virheet kaipaavat usein manuaalista analyysiä, joka voi vaatia hyvinkin paljon työtä. Testien laatu on tässä merkittävä tekijä ja erityisesti testien häiriönsietokyvyn rooli on suuri. Jos testit ennakoivat testien kannalta merkityksettömiä virheitä hyvin, ei testaajan tarvitse käyttää aikaa turhien epäonnistumisten tutkimiseen.

7.2 Tulosten analysoinnin uudet menetelmät

Tulosten analysointiin vaadittavan manuaalisen työn vähentämiseksi on tehty tiettyjä toimenpiteitä. Muun muassa turhan analysoinnin määrää on onnistuttu välttämään testien laadun ja erityisesti häiriönsietoisuuden parantamisella.

Testien määrän ollessa hyvin suuri vie koko regressiotestijoukon automaatiotestien päivitys paljon resursseja ja aikaa. Tästä johtuen aivan koko testijoukkoa ei ole ehditty vielä insinööriyön aikana päivittämään hyödyntämään uusia tekstissä esitettyjä toiminnallisuuksia. Tästä ja muista satunnaisista testien epäonnistumiseen johtavista syistä on insinööriyön aikana toteutettu Java-kielinen sovellus, joka analysoi suoraan Jenkins-sovelluspalvelimelta kaikki tietyn vaiheen testitulokset.

Tulokset analysoiva sovellus luo SSH-yhteyden Jenkins-palvelimelle ja noutaa XML-muotoiset ajojen testitulokset määritetystä tiedostopolusta. XML-tiedostot jäsennetään oliomalliin, joista voidaan suoraan generoida esimerkiksi Ant-targetit kaikkien tiettyyn syyhyn epäonnistuneiden testien ajamiseksi. Testit voidaan myös jakaa useisiin Ant-targetteihin ennakoidun suoritusajan perusteella. Nämä testit voidaan ajaa uudestaan Jenkins-palvelimella käyttäen generoitua Ant-määritelmää. Sovellus mahdollistaa tule-

vaisuudessa myös kattavamman epäonnistumissyiden analyysin, kun kaikki testit saatetaan ajantasalle hyödyntämään yhtenäisiä JUnit-assertointiviestejä.

Lisäksi nykyään testin päättyessä tulostetaan lokitiedostoon suuri määrä tietoja muun muassa prosessin aikana kohdatuista huomautuksista. Lisäksi muistissa olevien muuttujien arvot tallennetaan lokiin. Nämä lokitiedot auttavat selvittämään epäonnistuneiden testien syytä helpommin.

7.3 Raportointi

Sovellusversion testaustulokset julkaistaan johdolle ja asiakkaille Excel-muotoisena taulukkona, joka sisältää lukumäärät ja prosentuaaliset osuudet onnistuneista, epäonnistuneista ja ajamattomista testeistä, ja listan kaikista testeistä ajotuloksineen. Tähän asti tulokset on koottu taulukkoon manuaalisesti Jenkins-sovelluspalvelimen tuloksista siten, että pohjalla on käytetty edellisten ajojen taulukkoa. Ymmärrettävästi tämä vie resursseja, ja tämän vaiheen automatisointia on jo aloitettu hyödyntäen edellisessä luvussa esiteltyä analysointisovellusta.

8 Yhteenveto

Insinöörityön tarkoituksena oli tuottaa tilaajalle dokumentaatio sovelluksen regressiotestauksesta ja edistää sen automaatiota. Dokumentaation ohella kartoitettiin automaatiotestauksen haasteita, joihin pyrittiin löytämään ja toteuttamaan ratkaisuja.

Sovelluksen automaatiotestaus on aloitettu jo ennen insinöörityön aloittamista, mutta sen toimintaa ja vakautta on kehitetty vahvasti insinöörityön aikana. Kaikkia havaittuja muutostarpeita ja kehitysehdotuksia ei ole työn aikana ehditty kokonaan toteuttaa, mutta niiden kehitystyötä jatketaan vielä insinöörityöprojektin päättymisen jälkeen. Suurin haastavuus olemassa olevan testausprosessin kehittämisessä oli siinä, että automaatiotestien määrä oli jo valmiiksi hyvin suuri, ja testausprosessi jo pitkälle toteutettu. Tästä johtuen muutosten toteuttaminen vei huomattavasti enemmän resursseja kuin tapauksessa, jossa refaktoroitavia automaatiotestejä ei olisi ollut vielä useita.

Kehittämiskohteita on edelleen jäljellä: testien häiriönsietoon ja ylläpidettävyyteen pyritään vielä löytämään entistä tehokkaampia ratkaisuja. Myös tulosten analysoinnin ja raportoinnin manuaalisen työn määrää pyritään vähentämään entisestään.

Perehdyin työn aikana perusteellisesti automaatiotestaukseen kirjallisuuden kautta - tämä tuo konkreettista hyötyä minulle testaajan työssäni. Tiedän automaatiotestauksen haasteista, vaatimuksista ja parhaista käytännöistä nyt huomattavasti enemmän kuin ennen insinööriyden aloittamista.

Olisin toivonut voivani perehtyä työssäni enemmän automaation tulosten analysoinnin kehittämiseen siten, että suoraan sovelluspalvelimelta ajettujen testien tuloksista saisi generoitua konkreettiset raportoinnit sovellustestauksen seurantaan varten. Automaatiotestien refaktorointi ja testijärjestelmän kehittäminen vaativat kuitenkin resursseja oletettua enemmän, jolloin tulosanalyysiin liittyviin toimenpiteisiin ei jäänyt niin paljoa aikaa. Runko automatisoitua raportointia varten on kuitenkin toteutettu.

Koen insinööriyden hyödyttäneen itseni ohella myös projektia. Automaatiotestauksen tila on entistä parempi ja tulee vielä kehittymään kartoitettujen kehitysehdotusten myötä jatkossakin.

Lähteet

- 1 Haikala & Mikkonen. 2011. Ohjelmistotuotannon käytännöt. Helsinki: Talentum Media Oy.
- 2 Dustin, Rashka & Paul. 1999. Automated Software Testing: Introduction, Management, and Performance. 13. painos. Boston: Addison-Wesley.
- 3 Myers, Sandler & Badgett. 2012. The Art of Software Testing. 3. painos. Hoboken, New Jersey: John Wiley & Sons, Inc.
- 4 Software Business Competence. 2006. Testausstrategiat. Viitattu: 25.3.2014. <https://www.oamk.fi/sbc/testaus/testausstrategiat.htm>.
- 5 Homés. 2012. Fundamentals of Software Testing. Hoboken, New Jersey: John Wiley & Sons, Inc.
- 6 Porter. Testing. Viitattu 15.3.2014. <http://www.cs.umd.edu/~aporter/html/currTesting.html>.
- 7 Fewster & Graham. 1999. Software Test Automation: Effective use of test execution tools. Boston: Addison-Wesley.
- 8 Selenium HQ. Selenium Downloads. Viitattu: 22.3.2014. <http://docs.seleniumhq.org/download/>.
- 9 Selenium HQ. Selenium Documentation: Introduction. Viitattu: 22.3.2014. http://docs.seleniumhq.org/docs/01_introducing_selenium.jsp.
- 10 Jenkins CI. 2012. Setting up Eclipse to build Jenkins. Viitattu: 25.3.2014. <https://wiki.jenkins-ci.org/display/JENKINS/Setting+up+Eclipse+to+build+Jenkins>.
- 11 Graham & Fewster. 2012. Experiences of Test Automation: Case Studies of Software Test Automation. Boston: Addison-Wesley.
- 12 JUnit.org. JUnit API JavaDocs: Class Assert. Viitattu: 25.3.2014. <http://junit.sourceforge.net/javadoc/org/junit/Assert.html>.
- 13 Atlassian. Issue & Project Tracking Software. Viitattu: 11.5.2014. <https://www.atlassian.com/software/jira>.
- 14 IBM. Rational ClearQuest. Viitattu: 11.5.2014. <http://www-03.ibm.com/software/products/fi/clearquest>.
- 15 Bugzilla. About. Viitattu: 11.5.2014. <http://www.bugzilla.org/about>.

- 16 Apache Software Foundation. Commons Lang 3.1 API: Class StringUtils. Viitattu. 25.3.2014. <http://commons.apache.org/proper/commons-lang/javadocs/api-3.1/org/apache/commons/lang3/StringUtils.html>.
- 17 McConnell. 2004. Code Complete 2: A Practical Handbook of Software Construction. 2. painos. Redmond, Washington: Microsoft Press.
- 18 Chillarege. Software Testing Best Practices. IBM Research. Viitattu: 22.3.2014. <http://www.chillarege.com/authwork/TestingBestPractice.pdf>.