Janne Suomalainen

www-sovelluspalveluiden integraatiotestien automatisointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

28.4.2013

Metropolia

Tekijä Otsikko Sivumäärä Aika	Janne Suomalainen www-sovelluspalveluiden integraatiotestien automatisointi 63 sivua + 6 liitettä 28.4.2013
Tutkinto	insinööri (AMK)
Koulutusohjelma	tietotekniikka
Suuntautumisvaihtoehto	ohjelmistotekniikka
Ohjaajat	lehtori Olli Hämäläinen sovellusarkkitehti Matti Lehtomäki

Insinöörityö käsittelee sovellustestauksen automatisointia ja testien integrointia osaksi automaattisesti ajettavia regressiotestejä. Työn ensimmäisenä tavoitteena oli mahdollistaa www-sovelluspalveluiden testaukseen luotujen testitapausten automatisointi ja niiden suorituksen mahdollistaminen Enoro Oy:n kehittämästä Autotester-sovelluksesta. Käytännössä tällä haluttiin liittää SoapUI-nimisellä www-sovelluspalveluiden testaukseen keskittyvällä ohjelmistolla tehtyjä testejä osaksi automaattisesti ajettavia regressiotestejä.

Ensimmäinen tavoite ratkaistiin kehittämällä SoapUiRunner-niminen ohjelmakomponentti, joka integroitiin osaksi Autotester-sovellusta. SoapUiRunner mahdollistaa SoapUI-projektissa olevan testisarjan tai yksittäisen testitapauksen ajamisen suoraan Autotesterille kirjoitetulla AT-Skriptillä. Lisäksi SoapUiRunner muodostaa suorituksen jälkeen testituloksista selkeän html-muotoisen testiraportin.

Insinöörityön toisena tavoitteena oli tehdä selvitystyö siitä, miten Enoro Oy:n käynnissä olevan SAP-MDUS-integraatioprojektin testit voitaisiin tulevaisuudessa liittää myös osaksi automaattisesti ajettavia regressiotestejä. Selvitystyö keskittyi olemassa olevan testausratkaisun, Enoron kehittämän SapSim-sovelluksen tulevaisuuden roolin selvittämiseen. Siinä haettiin SapSim:ille järkevintä hyödyntämistapaa osana regressiotestien suorittamista. Selvitystyön johtopäätöksenä suositellaan SapSim:in toiminnallisuuksien liittämistä suoraan Autotester-sovellukseen, jolloin ylläpidettävänä olisi vain yksi testaussovellus.

SoapUiRunnerin kehityksen jälkeen useita www-sovelluspalveluita testaamaan tehtyjä SoapUI-testisarjoja voidaan jatkossa suorittaa Autotesterin kautta, jolloin niitä voidaan ajaa osana automaattisia regressiotestejä. Siten automaattisesti toistettavien regressiotestien määrä lisääntyy huomattavasti. Lisääntyneen testauksen seurauksena yhä suurempi osa kehitettävissä sovelluksissa ilmenneistä virheistä havaitaan ja voidaan korjata sovelluskehityksen aikaisemmassa vaiheessa. Lopputuloksena testauksen kattavuus paranee, joka mahdollistaa laadukkaampien ohjelmistoratkaisujen tarjoamisen asiakkaille.

Avainsanat	Autotester, regressiotestaus, SoapUI, testien automatisointi, www-sovelluspalvelu

Author	Janne Suomalainen
Title	Automation of web service integration tests
Number of Pages	63 pages + 6 appendices
Date	28 April 2013
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software technology
Instructors	Olli Hämäläinen, Senior Lecturer Matti Lehtomäki, Software Architect

The main themes of this thesis are software test automation and integration of automated test cases with united automated regression tests. The first objective of the project described in this thesis was to enable automating existing test cases created for web service testing, and enable their execution from a program called Autotester.

The first objective was achieved by developing a program component called “SoapUiRunner” and adding the component into the existing Autotester solution. SoapUiRunner is a software component that executes SoapUI test suites or single test cases in SoapUI test projects.

The second objective of the thesis is to describe how the testing of an on-going project called “SAP-MDUS integration” could be integrated into regression tests in the future. The main study was about determining the best way to utilize the SapSim application that was currently used to execute the tests in the test integration process. Based on the study, it is safe to recommend that the SapSim features should be integrated into the Autotester solution having only one application to maintain.

At the time being multiple test cases regarding web service testing can be executed through Autotester because of SoapUiRunner. Therefore, the amount of automated regression tests can be increased significantly. As a result, the total coverage of software testing is on the increase, which results in better quality software products for customers.

Keywords	Autotester, regression testing, test automation, SoapUI, web services

Sisällys

Lyhenteet ja määritelmät

Autotester — Enoron käyttämä, GENERISin testaamiseen kehitetty testiohjelma, josta voidaan ajaa Autotesterin komentosarjakielellä tehtyjä komentosarjoja (AT-skriptejä).

GENERIS — Enoro Oy:n modulaarinen järjestelmäratkaisu energia-alan tiedonhallintaan.

MDUS — Meter Data Unification and Synchronization. Mittaridatan yhdistäminen ja synkronisointi. (Advanced Meter Infrastructure 2012.)

Regressiotestaus

Testausta, joissa pyritään tutkimaan, toimivatko aiemmin kehitetyt/testatut ominaisuudet edelleen, kun tuotekomponenttia on kehitetty edelleen tai se on liitetty yhteen muiden toiminnallisuuksien kanssa. Regressiotestit on tarkoitus ajaa uudelleen aina uuden toiminnallisuuden valmistuttua. (Standard glossary of terms used in Software Testing 2012.)

SAP — Yhden maailman suurimmista ohjelmistovalmistajista, SAP AG:n yritysohjelmisto, jota käytetään toiminnanohjausjärjestelmiin liittyviin tuotteisiin.

Testauskomentosarja

Automatisoitu ”käsikirjoitus” eli skripti, joka suorittaa ohjelmassa tietyn käyttötarkoituksen mukaisen toiminnallisuuden ilman, että ihmisen täytyy tehdä testiä manuaalisesti.

www-sovelluspalvelu

Verkkopalvelimessa toimiva ohjelma, joka tarjoaa standardoitujen internetyhteyskäytäntöjen avulla palveluja sovellusten käytettäväksi (Tietotekniikan termitalkoot 2012).

1 Johdanto

Insinöörityö käsittelee www-sovelluspalveluiden testauksen automatisointia. Tarkoituksena on mahdollistaa pohjoismaisen energia-alan IT-yrityksen, Enoro Oy:n sovelluskehitykseen liittyvien testien automatisointi ja niiden liittäminen osaksi jatkuvasti toistettavia testauskierroksia. Siihen liittyen käsitellään testauksen automatisointiin kuuluvia käsitteitä ja testauksen taustalla olevaa termistöä. Niiden pohjalta rakennetaan varsinainen sovellusratkaisu, jolla vastataan testauksen automatisoinnin synnyttämiin tarpeisiin.

Insinöörityölle on asetettu tavoitteita. Ensimmäisenä tavoitteena on kehittää Enoro Oy:n olemassa olevia testaustyökaluja siten, että www-sovelluspalveluiden välistä tiedonsiirtoa hyödyntävien sovelluksien toimintaa pystytään testaamaan automaattisesti testejä suorittavalla työkalulla. Tämän päämäärän tavoittelu aloitetaan perehtymällä manuaalisesti ajettaviin testeihin ja sitä kautta suunnitellaan, miten testien automaattinen suorittaminen mahdollistettaisiin käytössä olevasta automaatiotestausympäristöstä käsin.

Ensimmäisen tavoitteen täyttyessä olemassa olevat testit voidaan hyödyntää osana testien automatisointia, jolloin niitä ei tarvitse rakentaa uudelleen. Lisäksi automatisoitujen testien ajaminen ei vaadi jatkossa enää manuaalista käynnistämistä. Testejä voidaan myös ajaa helposti osana laajempia regressiotestejä. Regressiotestaus on testausta, joita toistetaan ohjelmiston kehityksen aikana. Niillä varmistetaan, että aiemmin luodut sovelluskomponentit toimivat edelleenkin, kun ohjelmistoa tai sen osia on kehitetty eteenpäin (Pressman 2001, s.491). Ohjelma- tai sovelluskomponentilla tarkoitetaan tässä työssä jotain ohjelmiston osaa tai jopa yksittäistä luokkaa, joka mahdollistaa jonkin yksittäisen toiminnallisuuden toteuttamisen.

Toisena insinöörityön tavoitteena on tehdä selvitystyö Enoro Oy:n SAP-MDUS-integraatioprojektiin liittyen. Projektin keskeisessä osassa toimivat www-sovelluspalvelut, joiden kautta kaksi erillistä järjestelmää kommunikoivat keskenään. SAP-MDUS-integraatioprojektiin liittyvät testit halutaan myös tulevaisuudessa liittää osaksi regressiotestejä. Niille on kehitetty testaustyökalut mutta nykyiset työkalut eivät ole yhteensopivia käytössä olevan testiympäristön kanssa. Selvitystyössä pyritään selvittämään, mitä olemassa oleville työkaluille tulisi tehdä ja miten testit voitaisiin jatkossa

saada mukaan automaattisiin regressiotesteihin. Ensimmäisen tavoitteen ratkaiseminen tuonee myös synergiaetuja toisen tavoitteen taustalla olevan ongelman ratkaisemisessa. Molemmissa tapauksissa käytössä on osittain samoja testityökaluja. Toisen tavoitteet hyötyjä ei voida helposti näyttää toteen lyhyellä aikavälillä, koska sen tuottamat hyödyt realisoituvat myöhemmin.

Insinöörityön käytännön työosuuksien jälkeen tavoitteena on arvioida työn onnistumista ja tavoitteiden saavuttamista. Olettamuksena on, että toteutettava työ tuo helpotusta ja parannusta testaajien työskentelyyn, Enoron ohjelmistojen testaukseen ja osaltaan auttaa kehittämään entistäkin tehokkaampia ja kattavampia automaatiotestauksen ratkaisuja. Testien automatisoinnin lopputuloksena testaukseen kuluvan ajan pitäisi vähentyä, jolloin projektit pystytään joko viemään loppuun nopeammalla tahdilla tai vaihtoehtoisesti testauksessa säästyvä aika voidaan hyödyntää varsinaiseen kehitystyöhön.

2 Ohjelmistotestaus ja sen automatisointi

Ohjelmiston kehityksen elinkaareen kuuluu oleellisena osana ohjelmiston testaus. Ohjelmiston testauksessa tutkitaan kehitetyn ohjelman tai sen osien toimintaa erilaisissa tilanteissa ja erilaisilla syötteillä. Testausta voidaan tehdä joko manuaalisesti tai vaihtoehtoisesti automatisoiduin testein. Ohjelmistotestaus vaatii suunnittelua, ja sitä varten yrityksissä on usein määritelty testausstrategia, jota ohjelmistokehityksessä noudatetaan. (Dustin ym. 2009, s.129–130; Pressman 2001, s.21.)

Ohjelmiston testaukseen käytettävä aika ja testausajankohta suhteessa projektiin riippuvat käytettävästä ohjelmistokehitystavasta ja testausstrategiasta. Erilaisia ohjelmistokehitystyylejä ovat esimerkiksi testivetoinen kehitys (Test-driven development, TDD) ja ketterä ohjelmistokehitys (agile software development). Tässä dokumentissa ei perehdytä sen tarkemmin erilaisiin ohjelmistokehitystapoihin, vaan sen sijaan testaukseen. Yhteistä kaikelle ammattimaiselle ohjelmistokehitykselle on se, että testaaminen on oleellinen osa kehitystä ja että lopputulos vastaa laadullisesti asetettuja päämääriä.

Testausstrategialla määritellään testausmenetelmät, -tavoitteet ja ne vaiheet, joilla ohjelmiston ominaisuudet varmistetaan toimiviksi ja riittävän laadukkaiksi käyttöönottovaiheeseen mennessä. Testausstrategia sisältää niin kutsutun ”testauskartan” (testing roadmap). Se määrittää testit, jotka pitää olla suoritettu, että ohjelmaprojektin voidaan katsoa olevan valmis. Lisäksi testausstrategia sisältää päätöksen siitä, kuinka paljon testaukseen halutaan uhrata aikaa ja resursseja. Testausstrategiassa on myös tieto siitä, minkä asioiden toiminnasta halutaan erityisesti varmistua eli niin kutsutut korkean riskin komponentit. (Pressman 2001, s.477.)

2.1 Peruste ohjelmiston testaamiselle

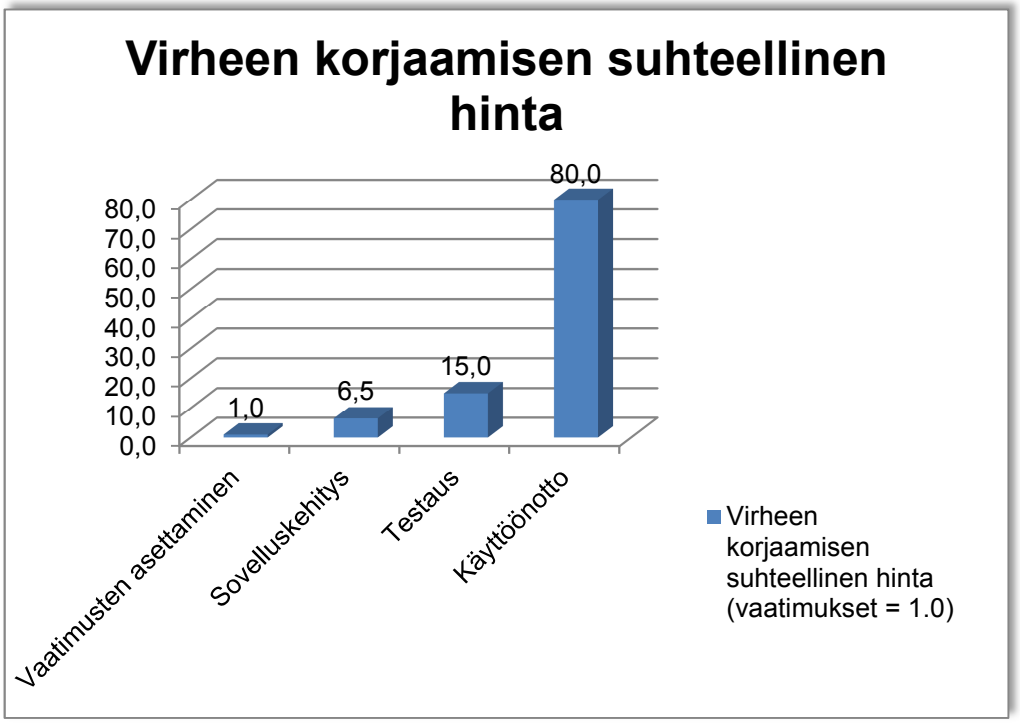
Ohjelmiston testaamisella on päämääränä vahvistaa kaksi asiaa. Ensimmäinen asia on se, että ohjelmisto tekee sen, mitä sen on tarkoitus tehdä ja toinen asia on se, että ohjelmisto ei tee mitään sellaista, mitä sen ei ole tarkoitus tehdä. Ensimmäisellä päämärällä tavoitellaan sitä, että ohjelma toimii odotetulla tavalla virheettömästi. Sen lisäksi ohjelman tulisi vastata sille asetettuja vaatimuksia, kun ohjelmaa käytetään sen käytölle tyypillisellä tavalla. Jälkimmäisen tehtävä on varmistaa, että ohjelma toimii hallitusti

myös poikkeuksellisissa tilanteissa. Poikkeukselliset tilanteet eivät välttämättä vastaa tapaa, jolla ohjelmaa on tarkoitus käyttää.

Monessa tapauksessa sovelluksen testauksen hoitaa testaukseen perehtynyt henkilö, joka ei ole sama henkilö, kuin sovelluksen testattavan ominaisuuden kehittäjä. Ohjelmiston testaajan tehtävä on etsiä virheitä, joita ohjelmasta löytyy. Lisäksi hän raportoi löydetyt virheet eteenpäin kehittäjille, jotka hoitavat virheiden korjaamisen. Testaajien tehtävä on myös etsiä tietoa ohjelmasta, miten se käyttäytyy erilaisissa tilanteissa. Saatava tieto on yksi arvokas syy testata ohjelmistoa. (Kaner ym. 2002, s.2, 16.)

Toinen arvokas syy harjoittaa ohjelmistotestausta on kustannusperusteinen. Ohjelmistojen virheistä aiheutunutta haittaa on tutkittu paljon ja virheiden on todettu aiheuttavan huomattavia taloudellisia menetyksiä. Esimerkiksi yhdysvaltain kauppaministeriön alainen National Institute of Standards and Technology (NIST) on arvioinut, että ohjelmavirheet aiheuttavat USA:n taloudelle vuosittain 59,5 miljardin dollarin kulut. Summaa olisi mahdollisuus pienentää jopa kolmanneksella pelkästään siten, että ohjelmistojen testausta kehitetään ja parannetaan. (Dustin ym. 2009, s.24.)

Ohjelmavirheen hinnan on tutkittu olevan sitä pienempi, mitä aikaisemmassa vaiheessa se havaitaan ja poistetaan. Kuvassa 1 on havainnollistettu kertoimin, kuinka paljon virheen hinta nousee, kun sen havaitseminen tapahtuu vasta myöhemmissä vaiheissa. Virheen korjauksen hinnan referenssipisteenä käytetään ensimmäistä vaihetta, jolloin ohjelmistoa vasta suunnitellaan, sen toimintaa spesifioidaan ja sille asetetaan vaatimuksia toiminnan suhteen. Kun ohjelman toteutuksen suunnitteluun käytetään riittävästi aikaa ja mahdollisia ongelmatilanteita ratkaistaan ennen varsinaista kehitystyötä, päästään huomattavasti halvemmalla kuin, jos virheitä aletaan korjata vasta kehitysvaiheessa saati sitten myöhemmin. Virheellä tarkoitetaan tässä yhteydessä paitsi ohjelman ohjelmointivirhettä (bug), myös sen toteutukseen liittyvää heikkoutta, joka estää tai hankaloittaa ohjelman kehittämisen, käyttämisen tai toimimisen järkevällä tavalla. (Pressman 2001, s.203; Ritscher 2010, s.2.).



Kuva 1. Virheen korjaamisen suhteellinen hinta sovelluskehityksen eri vaiheissa (Pressman 2001, s.203; Ritscher 2010, s.2).

Kaikkia virheitä on hyvin hankalaa havaita suunnitteluvaiheessa, eikä ohjelmistokehityksenkään aikana yleensä havaita kaikkia oleellisia virheitä. Tästä johtuen testauksesta ei pitäisi tinkiä, vaan sille pitäisi varata riittävän suuri aika. Kuten kuvasta 2 on nähtävissä, virheet, jotka havaitaan vasta käyttöönoton yhteydessä, tulevat monin kerroin kalliimmaksi suhteessa testauksessa paljastuviin virheisiin. Lisäksi ohjelmiston julkistamisen ja käyttöönoton jälkeen ilmenneet virheet aiheuttavat ohjelmistolle ja sen kehittäjälle laadullisen imagon heikkenemistä. (Pressman 2001, s.203; Ritscher 2010, s.2.)

Ohjelmiston testauksella ei voida kuitenkaan koskaan täydellisesti estää siitä, ettei virheitä jäisi huomaamatta ja sitä kautta päätyisi julkaistavaan versioon. Testaaminen vaatii paljon osaamista ja neuvokkuutta testaajalta ja sen kattavuuden lisääminen vaatii aina enemmän aikaa ja rahaa. Osaamista ja rahaa ei yleensä ole rajattomasti tarjolla. Siksi testauksessa tehdään usein kompromisseja ja lopputuloksella pyritään saavuttamaan riittävän hyvä laatu, joka tyydyttää kaikkia osapuolia. (Kaner ym. 2002, s.6, 8, 15.)

2.2 Ohjelmistotestausmenetelmiä

Erilaisia ohjelmistotestauksen menetelmiä on lukuisia. Käytetyt testausmenetelmät riippuvat siitä, mikä on testien ensisijainen tavoite. Eri tavoitteisiin tähtääviä testejä ovat esimerkiksi validaatio- ja verifikaatiotestit, regressio-, yhteensopivuus-, suorituskyky-, stressi- ja optimointitestit. (Dustin ym. 2009, s.11.; Pressman 2001, s.477)

Ohjelmistotestauksen menetelmät voidaan jakaa kahteen pääkategoriaan: mustan laatikon testeihin sekä lasilaatikkotesteihin eli toiselta nimeltään valkoisen laatikon testeihin. Mustan laatikon testaukseen (black-box testing) kuuluvat järjestelmän testit, jotka hyödyntävät käyttöliittymää ja ulkoisia rajapintoja. Oleellinen tekijä mustan laatikon testeissä on se, ettei testaaja tiedä, eikä hänen tarvitsekaan tietää, mitä järjestelmä sisältää tai miten se on toteutettu. Vastaavasti valkoisen laatikon testaus (white-box testing) sisältää järjestelmän sisäisen testaamisen eli muun muassa ohjelmakoodin testaamisen, kuten esimerkiksi ohjelmakomponenteille tehtävät yksikkötestit (unit tests). Valkoisen laatikon testaus vaatii jonkinasteista ymmärrystä ohjelmiston suunnittelumallista ja toimintalogiikasta. (Dustin ym. 2009, s.11–12; Gross 2005, s.76.)

Mustan ja valkoisen laatikon testien väliin jääviä testausmenetelmiä kutsutaan harmaan laatikon testeiksi (gray-box testing). Se, mitä kaikkea harmaan laatikon testaamiseen kuuluu, vaihtelee käytettävän määritelmän mukaan. Harmaan laatikon testauksena voidaan pitää esimerkiksi testejä, jotka suoritetaan kuten mustan laatikon testit, mutta joissa on käytössä tarkempaa informaatiota järjestelmän toiminnasta, kuin mitä käyttöliittymässä olisi tarjolla. Joskus voi olla tilanne, ettei käyttöliittymä anna minkäänlaista virheilmoitusta epäonnistuneesta tapahtumasta, jolloin mustan laatikon testin tulos tulkittaisiin hyväksytyksi. Harmaan laatikon testauksessa voitaisiin kuitenkin todeta testin epäonnistuneen, jos järjestelmän sisältä voidaan todeta, ettei toivottua lopputulosta ole syntynyt. Harmaan laatikon testaus vaatii sekä syvällisempää ymmärrystä järjestelmän toiminnasta että siihen vaikuttavista komponenteista. Tämäntyyppinen testaus rajataan usein koskemaan tiettyjä, kapea-alaisia osioita sillä testit ovat usein varsin monimutkaisia toteuttaa. (Dustin ym. 2009, s.12–13.)

Yleisesti tunnetuista testausmenetelmistä savu- ja regressiotestit kuuluvat pääosin valkoisen laatikon testeihin. Vastaavasti yhteensopivuus-, suorituskyky-, stressi- ja optimointitestit ovat yleensä pääosin toteutettu mustan laatikon menetelmillä. Harmaan

laatikon menetelmiä hyödynnetään enemmän tapauskohtaisesti. (Dustin ym. 2009, s.12–13.)

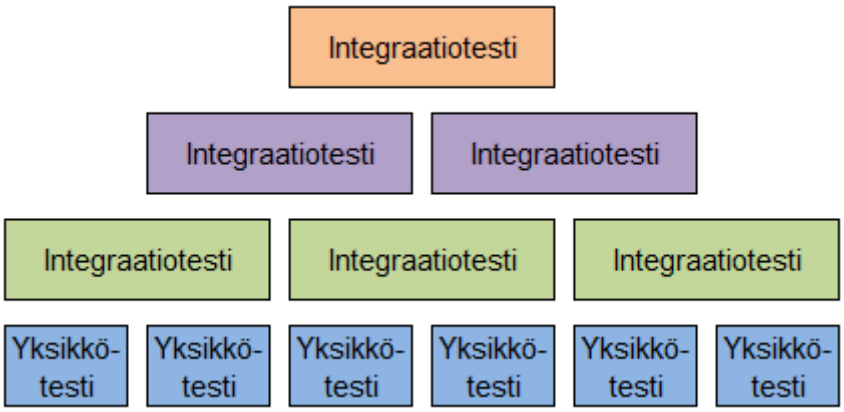
Hyvä esimerkki järjestelmän alimman tason testauksesta on yksikkötestaus, joka kuuluu valkoisen laatikon testeihin. Vastaavasti järjestelmän ylimmän tason testaukseen liittyvä esimerkki on tehdä testitapaus käyttötapauksen perusteella, jossa testataan järjestelmän toimivuutta jossakin todellisessa tilanteessa, miten käyttäjä käyttäisi järjestelmää. Tämänlainen testaaminen lukeutuu mustan laatikon testeihin. Seuraavissa luvuissa käsitellään näitä eri ääripäiden testaustapoja syvällisemmin.

2.2.1 Yksikkö- ja integraatiotestaukset

Yksikkötestauksella tarkoitetaan pienimmän mahdollisen sovelluskomponentin toiminnan testaamista. Olio-ohjelmoinnin yhteydessä yksikkötestauksella tarkoitetaan yleensä yhden luokan testaamista mutta yksikkötestaus voitaisiin rajata myös käsittämään vain yksittäinen metodi/funktio. Yksikkötestauksella pyritään varmistamaan, että yksittäiset komponentit toimivat yksin virheettömästi. (Pressman 2001, s.485, 636.)

Integraatiotestit ovat usein osa regressiotestejä. Niissä pyritään varmistamaan, että yksistään toimivat komponentit toimivat myös silloin, kun ne liitetään yhteen muiden sovelluskomponenttien kanssa. Integraatiotestejä suoritetaan sen jälkeen, kun sovelluskomponentit ovat läpäisseet yksikkötestauksen. Jotta integraatiotestaus olisi järkevää, ensin on täytynyt varmistaa komponenttien toimiminen yksistään.

Integraatiotestejä voidaan tehdä pyramidimallisesti siten, että ensin testataan kahden komponentin toimiminen yhdessä, jonka jälkeen testattuja kokonaisuuksia yhdistellään kuvan 2 mukaisesti. Yksikkö- ja integraatiotestejä voidaan tehdä sovelluskehityksen ollessa vielä kesken aina silloin, kun uusia sovelluskomponentteja valmistuu. (Pressman 2001, s.488–489, 636.)



Kuva 2. Yksikkö- ja integraatiotestien suorittamishierarkia (Pressman 2001, s.482).

2.2.2 Järjestelmätestaus

Jos sovelluksen testausta suoritettaisiin vain ohjelmakoodin osalta, saataisiin tieto siitä, että koodi tekee sen, mitä sen on haluttu tekevän. Tällöin ei vielä ole varmistettu, että varsinainen sovellus tekisi kokonaisuutena sen, mitä sille on asetettu vaatimukseksi (requirements). Sovelluksen vaatimukset luodaan siinä vaiheessa, kun sovelluksen sisältöä suunnitellaan. Vaatimukset muodostavat yhdessä syyn sille, miksi sovellus ylipäätään tehdään. Niihin liittyvät käyttötapaukset luodaan kuvaamaan sovellukselle asetettuja vaatimuksia eli sitä, mitä vaatimuksen mukaisella toiminnolla halutaan käytännössä saada aikaiseksi. Vasta käyttötapausten testaaminen kertoo todella sen, tekeekö sovellus sen, mitä siltä vaaditaan. (Gross 2005, s.73.)

Käyttötapaukset pohjautuvat usein käyttöskenaarioihin. Käyttöskenaario on lyhyt kertomus käyttäjän näkökulmasta katsottuna siitä, mikä on käyttäjän tilanne ja mitä hän haluaa saavuttaa. Käyttöskenaarion ei ole tarkoitus kuvata, miten jokin asia saavutetaan, vaan yksinkertaisesti vain mitä halutaan saavuttaa. Käyttöskenaarioista selviää, mitä sovelluksella pitäisi pystyä tekemään. Käyttötapaus on yksi polku tai kuvaus siitä, miten käyttäjä saavuttaa haluamansa tilanteen. (Ritscher 2010, s.3.)

Käyttötapausten perusteella rakennetut testitapaukset ovat erittäin hyödyllisiä mittaamaan sitä, pystytäänkö sovelluksella tekemään kaikki se, mitä sillä pitäisi pystyä tekemään. Toisin sanoen ne mittaavat sitä, onko kaikki sovellukselle asetetut vaatimukset täyttyneet. Käyttötapausten perusteella sovellusta tarkastellaan käyttäjän näkökulmasta. Niistä voidaan hahmottaa ohjelmiston käytetyimmät eli kriittisimmät osiot ja sen

avulla testausta voidaan kohdentaa nimenomaan kriittisiin toiminnallisuuksiin. (Ritscher 2010, s.3.)

2.3 Ohjelmistotestauksen automatisointi

Käsite ”sovelluksen automaattinen testaus” voidaan ymmärtää monin eri tavoin. Toisinaan sen ymmärretään koskevan testivetoista kehitystä tai automaattista yksikkötestausta. Toisinaan sen taas käsitetään tarkoittavan komentosarjaa eli skriptiä, jossa testausohjelmisto suorittaa järjestyksessä tietyt toiminnot sovelluksessa. Käsite voidaan myös ymmärtää rajoittuvan pelkästään suorituskykytestaukseen, jossa testiohjelma ajaa samanaikaisesti satoja tai tuhansia rinnakkaisia käyttötestejä. Sovelluksen automaattinen testaus tarkoittaa tässä yhteydessä kaikkia yllä mainittuja asioita. (Dustin ym. 2009, s.3.)

Ohjelmiston automaattinen testaaminen eroaa manuaalisesta ohjelmistotestauksesta muun muassa siinä, että automaation avulla pystytään tekemään testejä, jotka olisivat manuaalisesti testaten erittäin haastavia, elleivät jopa mahdottomia. Testien automatisointi on osa sovelluskehitystä. Testien automatisointiaikeet tulee usein huomioida jo sovellusta kehitettäessä, että sovelluksella olisi rajapinta tai tuki testausympäristölle, jonka avulla testejä voidaan automatisoida. Testien automatisointi ei kuitenkaan poista täysin manuaalisen testaamisen tarvetta. Kaikkea testausta ei yleensä voi automatisoida. Lisäksi testien automatisoinnin osaaminen vaatii ymmärrystä siitä, miten testaus manuaalisesti suoritettaisiin. (Dustin ym. 2009, s.4, 81, 83–84.)

2.3.1 Testauksen automatisointiin käytettäviä työkaluja

Testauksen automatisointiin liittyen on kehitetty lukuisia ohjelmistoja ja työkaluja. Niitä löytyy vapaan lähdekoodin sovelluksina, kuten esimerkiksi Harness, jUnitum ja Software Testin Automation Framework. Lisäksi ohjelmistoja löytyy myös kaupallisina tuotteina, kuten esimerkiksi SmartBearin TestComplete ja SoapUI Pro. Osa testauksen automatisointiin tarkoitetuista työkaluista integroituu osaksi sovelluskehitysympäristöä. Näitä testausliitännäisiä (plugins) löytyy esimerkiksi Microsoftin Visual Studioon (Visual Studio Test professional) sekä suosittuun avoimeen lähdekoodiin perustuvaan Eclipse-kehitysympäristöön (Jubula, Maveryx jne). Lisäksi Microsoftin Visual Studion Enterprise- ja Ultimate-versioihin on sisäänrakennettu yksikkö- ja käyttöliittymätestaustyökalu-

ja. Useimmiten tarjolla olevat yksittäiset testaustyökalut ovat keskittyneet pääasiallisesti yhdentyyppiseen testaukseen eli ne toimivat periaatteella ”yksi työkalu yhteen tehtävään”. (Automated Testing – TestComplete. 2013; Functional testing 2013; Tour Visual Studio Test Professional 2010 2013.)

Testaus, jossa suoritetaan monia erilaisia testejä, vaatii useimmiten useiden eri testityökalujen käyttöä. Niiden käyttöä varten on hyvä pystyttää testausympäristö, josta eri testejä voidaan ajaa. Kun ryhdytään suunnittelemaan testauksen automatisointia, olisi hyvä kartoittaa, minkä tyyppinen testiympäristö sopii omiin testaustarkoituksiin. Lisäksi tulisi selvittää, olisiko sopivaa testiympäristöä saatavilla valmiina ratkaisuna. Silloin kun päädytään kehittämään oma testiympäristö, olisi se kannattavaa toteuttaa siten, että siihen voidaan tarvittaessa integroida mahdollisimman monia testausohjelmia. Tällöin testitarpeiden muuttuessa tai jos markkinoille tulee parempia testaustyökaluja, ne voidaan helposti liittää osaksi omaa testausympäristöä. (Dustin ym. 2009, s.5.)

Testien automatisointiin tarkoitetulle testiympäristölle olisi järkevää asettaa seuraavat vaatimukset: Järjestelmän testaaminen pitäisi olla mahdollista ilman, että testikoneelle asennetaan mitään ohjelmia, koska ohjelmien asentaminen välttämättä ole aina mahdollista. Lisäksi testiympäristön pitäisi tukea sekä yrityksen olemassa olevia että kehitteillä olevia sovelluksia. Testiympäristön pystyttämisessä tulisi myös pyrkiä siihen, että testiympäristö pystyisi hoitamaan kaikki testauksen elinkaareen sisältyvät testit alusta loppuun, jolloin kaikki testaus olisi mahdollista suorittaa yhden käyttöliittymän avulla. Lisäksi testauksen kannalta olisi hyödyllistä, jos testiympäristö tukisi testauksen hajauttamista useille eri tietokoneille tai palvelimille, jolloin testausta voitaisiin tehostaa myös suorituskyvyn osalta. Samalla testiympäristössä käytettävät työkalut olisi järkevää valita siten, että niitä voidaan hyödyntää mahdollisimman monissa testeissä, jolloin testiympäristö pysyisi mahdollisimman yksinkertaisena. Huomioon otettavia asioita on siis kokonaisuutena varsin paljon, ja ne vaativat syvällistä perehtymistä. (Dustin ym. 2009, s.5.)

2.3.2 Testauksen automatisoinnin tavoitteet ja siitä saatavat hyödyt

Kun ohjelmistotuotantoyritys harkitsee testausprosessiensa automatisointia, sisältyy siihen merkittäviä aloituskustannuksia. ”Miksi automatisoida?” on se kysymys, johon täytyy löytyä perusteltu vastaus silloin, kun mietitään, kuinka paljon kustannuksia syntyy testauksen automatisointiin perehtymisestä, käytännön toteutukseen vaadittavien

resurssien kiinnittämisestä ja testauksen automatisoinnin vaatimien välineiden ja järjestelmien hankkimisesta. Näin ollen testauksen automatisoinnin tulisi tuoda mukanaan sellaisia hyötyjä, joilla aloituskustannukset voidaan perustella. Aloituskynnyksen ylittämiseksi saavutetut hyödyt tulisi pystyä konkretisoimaan luvuiksi ja sitä kautta rahassa mitattaviksi, jotta testauksen automatisoiminen voitaisiin todeta kannattavaksi toteuttaa. (Dustin ym. 2009, s.23–24, 69–70; Ritscher 2010, s.3.)

Useissa tapauksissa testausautomaatioon panostaminen, aloitus- ja ylläpitokustannuksista huolimatta, on varsin perusteltua ja hyväksyttyä. Testaukseen kuluu merkittävä osa koko ohjelmistokehitysprojektiin kuluvasta ajasta. Vaadittavan laadun varmistaminen ohjelmistoa testaamalla on paitsi kallista, se tuo myös mukanaan monia haasteita, jotka tulevat aina vain monimutkaisemmiksi käyvien sovelluksien sivutuotteina. Testausautomaatio tarjoaa ratkaisua moniin merkittäviin haasteisiin, joita ohjelmistoala kohtaa tänä päivänä. Testausautomaation mukana tuomia etuja ovat muun muassa testaukseen kuluvan ajan lyheneminen, ohjelmiston laadun parantuminen, manuaalisen testauksen tehostuminen, testauksen kattavuuden paraneminen sekä sellaisten ongelmien tutkimisen mahdollistaminen, jotka olisivat lähes mahdottomia suorittaa manuaalitestauksen keinoin. (Dustin ym. 2009, s.23–26, 56–57.)

Automatisoidun testitapauksen suorittaminen kestää yleensä huomattavasi lyhyemmän ajan verrattuna siihen, että testi tehtäisiin manuaalisesti ihmistestaajan toimesta. Lisäksi, kun testitapaus on kerran automatisoitu, sitä voidaan helposti suorittaa myös myöhemmin, kun ohjelmistoa on kehitetty edelleen. Samojen testien toistamisella varmistetaan, ettei uusi kehitys ole rikkonut aiemmin toiminutta toiminnallisuutta. (Dustin ym. 2009, s.26–27.)

Testitapausten automatisoinnilla on myös mahdollista vaikuttaa lopputuotteen kokonaislaatuun ja testien kattavuuteen. Kerran automaattiseksi testiksi tehtyjä testirunkoja voidaan helposti hyödyntää siten, että testiä muokataan hieman esimerkiksi syötteiden tai muun testidatan osalta, ja tuotoksesta tehdään uusi testitapaus aiempien rinnalle. Tällöin toiminnan testaus on kattavampaa mutta automatisointiin käytetty aika ei lisäänny yhtä merkittävästi. Lopputuloksena suoritettavia testejä on useampia, ja sitä kautta on odotettavissa, että ohjelmiston laatu paranee. Näin ollen virheitä löydetään yhä enemmän ja ohjelmisto toimintaa on testattu tarkemmin, jotta se toimisi useimmissa erilaisissa tilanteissa odotusten mukaisesti. (Dustin ym. 2009, s.38–39.)

Useimmiten yhden komponentin perinpohjainen testaaminen testaajan toimesta ei ole resurssi- ja aikataulusyiden vuoksi mahdollista. Sen sijaan kerran automatisoitua testiä voidaan ajaa vuorokauden ajasta riippumatta, jolloin testeihin kuluvalla ajalla ei ole yhtä suurta merkitystä kuin manuaalisesti testatessa. Totuus kuitenkin on se, että mitä enemmän ja kattavammin ohjelmistoa testataan, sen vähemmän siihen jää havaitsemattomia virheitä. (Dustin ym. 2009, s.40–41, 45–46.)

Kun rutiininomaiset yksinkertaiset testit on saatu automatisoitua, testaajat voivat käyttää aikansa hyödyllisemmin. He voivat keskittyä sellaisiin testeihin, joita ei ole syystä tai toisesta järkevää automatisoida tai he voivat lisätä ja kehittää uusia testausmenetelmiä. Hankaliin testitapauksiin voidaan perehtyä entistä syvemmin ja niiden toimintaa tutkia entistä tarkemmin, kun aikaa säästyy yksinkertaisimpien testien tekemiseltä. Lisäksi testaajilla jää enemmän aikaa testausprosessin suunnitteluun ja valvomiseen. Näin ollen testaajien työstä tulee myös monipuolisempaa, motivoituneempaa ja mielenkiintoisempaa, kun rutiininomaiset työt vähenevät.

Eräs testauksen automatisoinnin puolesta puhuva seikka on se, että testausautomaation avulla pystytään tekemään paljon asioita, joita ilman automaatiota ei olisi mahdollista testata. Näitä asioita ovat esimerkiksi erilaiset suorituskyky- ja stressitestit, jotka testaavat järjestelmää tai ohjelmistoa ääritilanteissa. Niissä järjestelmää rasitetaan sellaisin keinoin, jotka olisivat ihmisvoimin mahdotonta saavuttaa. Lisäksi ihmistestaaja on inhimillinen ja hän voi sokeutua testattavalle asialle, jos samoja testejä toistetaan kerta toisensa jälkeen. Testaaja ei välttämättä enää havaitse testattavan ohjelmiston puutteita samalla tavalla, kun ensimmäisellä testauskerralla. Kun testi on automatisoitu, voidaan olla varmoja siitä, että se suoritetaan aina samalla tavalla, kerta toisensa jälkeen. Lisäksi automaattisesti suoritettavat testit eivät ole niin riippuvaisia testaukseen erikoistuneiden eksperttien läsnäolosta, kuin vastaavasti monet manuaaliset testit olisivat. Näin ollen testejä voidaan ajaa yhtä lailla esimerkiksi lomakaudella. (Dustin ym. 2009, s.44–45.)

2.3.3 Testauksen automatisoinnin luomat haasteet

Testausautomaation käyttöönotto tuo mukanaan uusia haasteita. Haasteita syntyy testiympäristön vaatimusten, nykyisten ja tulevien testitapausten kuin myös sovellusten osalta. Haasteiden ohittaminen ei sinänsä ole erityisen hankalaa mutta sovelluskehittäjiltä ja -testaajilta tulee löytyä aito halu ja tahtotila testien automatisoimiseksi.

Ensimmäinen testauksen automatisoinnin mukanaan tuoma haaste on automatisoinnin käyttöönottoon liittyvä haaste. Monissa tutkituissa tapauksissa testaamisen automatisoimishanke on päättynyt ongelmiin, jotka juontuvat siitä, ettei testausautomaatioon siirtymiseen vaadittavalle työlle anneta riittävästi resursseja. Epäonnistumisen syy on varsin ymmärrettävä: luonnollisestikin uusien käytäntöjen ja prosessien käyttöönotto vaatii aikaa ja perehtymistä. (Dustin ym. 2009, s.69–70, 83–84.)

Toinen automatisointiin liittyvä haaste liittyy testiympäristöön. Automaatiotestiympäristö vaatii jatkuvaa ylläpitoa. Kun sovellukset ja tekniikat vaihtuvat, tarvitaan niiden testaamiseen uusia testitapoja, työkaluja ja testiympäristön kehitystyötä. Testiympäristön ylläpitoon olisi hyvä varata resursseja ja nimetä vastuuhenkilö tai -henkilöitä, jotka vastaavat testiympäristön ylläpidosta. Lisäksi automaattista testausprosessia olisi hyvä pyrkiä jatkuvasti kehittämään aivan kuten muitakin prosesseja, jotta se ei jäisi jälkeen testausmenetelmien kehityksestä. Nämä asiat kuuluvat usein osaksi sovelluskehitysyrityksessä laadittua testausstrategiaa. (Dustin ym. 2009, s. 4, 80–81.)

Kolmas automatisointiin liittyvä haaste liittyy testitapauksiin. Periaatteessa kaikki manuaaliset testitapaukset voitaisiin automatisoida juuri sellaisina kuin ne ovat olleet. Siitä huolimatta olisi järkevää pohtia, kannattaisiko testitapauksia mukauttaa paremmin automatisointiin sopivaksi. Tällöin testauksesta voidaan usein saada paljon enemmän irti ja testi voi muokattuna palvella paljon paremmin testauksen tarkoitusta. Lisäksi testauksia automatisoidessa ja erilaisia testauskomentosarjoja käytettäessä tulee huomioida se, että ne voivat lakata toimimasta, jos käyttöliittymään tai käyttölogiikkaan on tullut muutoksia edellisen testauskerran jälkeen. Siksi on tyypillistä, että testitapauksia joudutaan ylläpitämään ja muokkaamaan jatkuvasti. (Dustin ym. 2009, s. 4, 80–81.)

Neljäs automatisointiin liittyvä haaste liittyy testaajien osaamiseen. Testauksen automatisoiminen asettaa uusia haasteita testaajille. Heiltä vaaditaan entistä enemmän ymmärrystä sovelluskehityksestä ja siitä, miten testit toimivat. Automaattisten testien lopputuloksia ei voida hyväksyä olematta varma siitä, että testit toimivat oikein ja että ne todellakin testaavat sen, mitä niiden on tarkoitus testata. Yhtä lailla testien automatisointi vaatii enemmän ymmärrystä sovelluskehittäjiltä siitä, miten testaaminen tapahtuu. Sovelluksiin joudutaan usein kehittämään toiminnallisuuksia nimenomaan automaattista testaamisprosessia silmällä pitäen. (Dustin ym. 2009, s. 81, 83–84.)

Jotta tehokkaita testiskriptejä voitaisiin tehdä, tulisi sovellukseen olla tarjolla jokin käyttöliittymä testausohjelmaa varten. Käyttöliittymän avulla testaustyökalun tulisi saada palaute tehdyistä toimenpiteistä. Siten niistä voitaisiin päätellä, onko testi läpäisty vai jäikö jokin asia toimimatta tai testaamatta. Automaattisesta testaamisesta saadaan huomattavasti tehokkaampaa, kun siihen liittyvät vaatimukset otetaan huomioon jo sovelluksen osia kehittäessä. (Dustin ym. 2009, s. 83–84.)

Testauksen automatisointi tuo siis mukanaan monia haasteita, joiden ratkaiseminen vaatii miettimistä, prosessien kehittämistä ja testauksen huomioimisen jo sovellusta kehittäessä. Näiden haasteiden ratkaiseminen ei ole kuitenkaan kohtuuttoman vaikeaa tarjolla olevaan hyötypotentiaaliin nähden. Kun testauksen automatisointiprosessit on kertaalleen kehitetty ja vaadittavat toimet ovat omaksuttu osaksi sovelluskehitystä, on tulevien testien automatisointi jo huomattavasti helpompaa ja vähemmän resursseja vaativaa.

2.3.4 Testauksen automatisoinnin kannattavuus

Kuten aiemmissa luvuissa on jo todettu, testauksen automatisoinnilla voidaan saavuttaa merkittäviä sovelluskehityskulujen alennuksia sekä ajan ja resurssien säästöä. Mitään takeita ei kuitenkaan ole olemassa että, testauksen automatisoinnilla lopulta saavutettaisiin haluttuja päämääriä. Saadakseen vastauksen kysymykseen, kannattaako tiettyä testiä tai testausvaihetta automatisoida, täytyy pystyä analysoimaan investoinnille saatavaa pääoman tuottoastetta eli ROI-mittaria. (Dustin ym. 2009, s.4–5.)

Se, että taloudellista hyötyä pystytään arvioimaan etukäteen ROI-mittarien avulla, mahdollistaa automatisoinnin kannattavuuden analysoinnin ennen varsinaisia käytännön toimenpiteitä. Taloudellisia ROI-mittareita on testauksen automatisoinnin yhteydessä suhteellisen helppoa tutkia. Osuvien johtopäätösten tekeminen vaatii kuitenkin asiantuntijuutta ja kokemusta. Kun esimerkiksi pystytään tekemään suhteellisen tarkkoja arvioita lukujen muodossa siitä, kuinka paljon nopeammin saman asian testaaminen onnistuu yhdellä kerralla tai projektin aikana kertyneiden testauskertojen yhteenlaskettuna tuloksena, saadaan myös kohtalaisen relevanttia tietoa siitä, kuinka suuri taloudellinen hyöty testauksen automaatiosta syntyy. (Dustin ym. 2009, s.4, 26.)

Tutkimukset ovat osoittaneet, että noin puolet sovelluskehitykseen kuluvasta ajasta menee testaukseen. Niinpä testauksen voidaan katsoa synnyttävän merkittävän osan

sovelluskehityksestä syntyvistä kustannuksista. Jos testauksen keston lasketaan esimerkiksi puolittuvan testauksen automatisoinnin avulla, saavutettaisiin jopa 25 prosentin hyöty kokonaiskustannuksissa. (Dustin ym. 2009, s.26–27.)

Testauksen automatisointi kannattaa aloittaa eniten toistettavista osista tai testeistä, jotta saavutettaisiin suurin taloudellinen hyöty. Juuri toistettavuuden helppous on vahvin testien automatisoinnin puolestapuhuja. Näin ollen, jos testin automatisointiin kuluu esimerkiksi tunti ja testin manuaaliseen suorittamiseen kuluu 30 minuuttia, tuo jokainen testaus taloudellista säästöä kolmannesta testauskerrasta lähtien. Tuolloin testaaminen ei enää sido testaajan aikaa kovinkaan merkittävästi alkuponnistelujen jälkeen. Tämän lisäksi testiin tuskin kuluu enää yhtä kauan aikaakaan, kun suorituksen vaiheet tapahtuvat automaattisesti.

Kaikkia testien osia ei välttämättä kannata ryhtyä automatisoimaan, kun halutaan saavuttaa mahdollisimman suuri taloudellinen hyöty. Tiettyjen tutkimusten mukaan vain noin 40–60 prosenttia sovelluksen kehityksen elinkaaren aikana tehtävistä testeistä kannattaisi automatisoida. Se, mitä testejä ne missäkin sovelluksessa on, opitaan yleensä kokemuksen karttuessa. (Dustin ym. 2009, s.27.)

Testien automatisoinnilla haetaan yleensä testausajan nopeutumista sekä kykyä testata joitain osioita tarkemmin ja monipuolisemmin. Näitä tavoitteita silmällä pitäen tulisi tehdä erillinen ROI-analyysi jokaisen sovelluskomponentin tai testattavan kokonaisuuden testauksen automatisointia päätettäessä. Analyysissä tulisi miettiä vastausta kysymyksiin: ”kuinka suuri osa testeistä voidaan automatisoida?”, ”mikä on odotettu ajallinen hyöty, joka testin automatisoinnilla saavutetaan suhteessa manuaaliseen testaamiseen?”, ”kuinka paljon suuremman osan toiminnasta testaus kattaa ja kuinka paljon testit kohottavat tuotteen laatua?” ja ”onko mukana muita oleellisia seikkoja, joihin päätöksellä on vaikutusta?”. (Dustin ym. 2009, s.27.)

Esimerkkilaskelma automatisoinnin tuomasta tuotosta

Kuvassa 3 on ROI-laskelma tuotosta, jonka testien automatisointi tuo tietyssä tapauksessa, kun yhdelle työtunnille lasketaan hinnaksi 100 euroa. Testijärjestelmän konfiguroinnissa säästetty aika riippuu siitä, kumman testijärjestelmän, manuaalisen vai automaattisen, pystyttämiseen menee enemmän aikaa. Testijärjestelmän kehitykseen kuluu luonnollisesti enemmän aikaa, kun kyse on automatisoinnista. Siksi kyseinen luku nä-

kyy kuvassa negatiivisena. Vastaavasti automaattisten testien ajossa säätyy huomattava määrä aikaa suhteessa manuaaliseen testaajaan. Mitä enemmän testeille on toistokertoja, sitä suurempi on ajallinen säästö kyseisen arvon osalta. (Dustin ym. 2009, s.55–57.)

Kokonaisarvio testauksen automaation tuomista säästöistä			
Tehtävä	**Automatisoidun testin säästämä aika (h)**	**Tehtävä**	**Automatisoidun testin säästämä summa (€)**
Testijärjestelmän konfiguroimisessa säästetty aika	xxx	Testijärjestelmän konfiguroimisessa säästetty aika	xxx
Testien kehityksessä säästynyt aika	-250	Testien kehityksessä säästyneet kulut	-25 000
Testien ajossa säästetty aika	1 583,33	Testien ajossa säästetyt kulut	158 333
Testien arvionnissa / tutkimisessa säästetty aika	2 250	Testien arvionnissa / tutkimisessa säästetyt kulut	225 000
		Muut automatisoinnin kulut	-25 000
Tunnit	**3 583,33**	Euroja	**333 333 €**
Päivät (8h/pv)	448		
Kuukaudet (20 pv/kk)	**22**		

Kuva 3. ROI-laskelma testauksen automatisoinnista, kun työtunnin hinta on 100 euroa. (Dustin ym. 2009, s.57.)

Esimerkkilaskelmassa on tehty arvio tuhannella testillä, joista jokaisen suorittaminen kestää kymmenen minuuttia manuaalisesti suoritettuna ja puoli minuuttia automaattisesti suoritettuna eli kahdeskymmenesosan manuaaliseen suoritukseen verrattuna. Jokainen testi on laskettu toistettavaksi kymmenen kertaa. Näin ollen testien ajo manuaalisesti kestäisi 1 666,66 tuntia ja automaattisesti ajettuna vain 83,33 tuntia. (Dustin ym. 2009, s.57–61.)

Testitulosten analysointiin voidaan käyttää aikaa hyvinkin huomattavasti. Tämän takia on hyödyllistä myös automatisoida testitulosten arviointi ja diagnostiikka. Esimerkkilaskelmassa on automatisoitu testien tuottamat tulosteet. Tuloksia on laskettu olevan 3000 kappaletta, joista yhden kappaleen manuaaliseen arviointiin on laskettu kuluvan aikaa viisi minuuttia. Sama aika toistuu testien jokaisella toistokerralla eli kymmenen kertaa. Vastaavasti automaattiseen tulosten analysointiin on arvioitu kuluvan puoli minuuttia eli kymmenesosa manuaaliseen analysointiin kuluvasta ajasta. Näin ollen manuaalisesti suoritettuihin arviointeihin menisi kokonaisuudessaan 2 500 tuntia, kun au-

tomaattisesti sama asia hoituisi kymmenesosassa ajasta eli 250 tunnissa tuoden ajallista säästöä jopa 2250 tuntia eli yli kolme kuukautta. (Dustin ym. 2009, s.61–63.)

Muihin automatisoinnin kuluihin on huomioitu hankittavista automatisointityökaluista, henkilöstön koulutuksesta ja järjestelmien ylläpidosta aiheutuvat kustannukset. Automatisoinnin tuomat lisäkulutkin huomioiden, kuvan laskelmasta on nähtävillä, että automatisointi tuo esimerkkitapauksessa todella huomattavia säästöjä. Lisäksi on paljon sellaisia positiivisia vaikutuksia, joiden hyötyä on haastavaa laskea numeroina. Näitä ovat esimerkiksi testien kattavuuksien lisääntyminen ja sitä kautta saavutettu laadullinen hyöty. Kehitystyöhön saattaa kulua jatkossa huomattavasti vähemmän aikaa, kun virheet havaitaan tehokkaammin ja sovelluskehityksen aikaisemmassa vaiheessa. Kaikki automatisointitapaukset eivät kuitenkaan aina välttämättä tuo laskelmaan positiivista lopputulosta. Joissain tapauksissa voikin osoittautua, ettei testien automatisointi sovellukaan aiottuun projektiin. (Dustin ym. 2009, s.63–65.)

3 Testiympäristö

Sovelluksen testaamista varten täytyy pystyttää sopiva testiympäristö. Insinöörityön yhteydessä tehtävää sovelluskomponenttia ajetaan olemassa olevissa testiympäristöissä, joihin asennetaan vain tarvittavat lisäohjelmistot. Lisäohjelmilla tarkoitetaan asennettavia tai päivitettäviä ohjelmia, joita otetaan käyttöön, jotta olemassa olevia testejä voitaisiin ajaa automaattisesti Autotesterille laadittujen skriptien avulla.

Vaikka testiympäristöä ei insinöörityötä varten tarvitsekaan koota tyhjästä, halutaan kuitenkin esittää ja dokumentoida ne testausympäristön ominaisuudet, joita käytettävältä testiympäristöltä vaaditaan. Lisäksi käydään läpi ne vaatimukset ja testimenetelmät, joiden ympärille testiympäristölle laadittavat ominaisuudet tulisi perustaa. Tämä dokumentaatio mahdollistaa periaatteessa sen, että vastaavanlainen testiympäristö pystytään toteuttamaan myös alusta alkaen. Lisäksi, kun perusteet ja vaatimukset testiympäristön taustalla tunnetaan, pystytään helpommin selvittämään mahdollisuudet muuttaa jotakin yksittäistä ominaisuutta, kun sellainen tarve tulee eteen.

Optimaalinen tilanne on se, jossa testien ajoon liittyvä ympäristö on suunniteltu siten, että se voidaan asentaa yhdelle testipalvelimelle. Testattavat palvelut voivat tosin sijaita toisaallakin, samassa tai julkisessa verkossa. Testiympäristön pystytykseen lasketaan tässä yhteydessä kuuluvan testien automatisointiin vaadittavien ohjelmien ja palveluiden asentaminen ja testaus sekä konfiguraatioiden asettaminen tarkoitukseen sopivaksi. Kun testipalvelinta on onnistuneesti testattu ja siinä on ajettu tarkoituksenmukaisia testitapauksia onnistuneesti, voidaan pystytyksen katsoa olevan valmiina.

Käytettävän testiympäristön tulee tukea projektin käyttöön luotujen testien automaattista ajoa. Siksi sen suunnittelussa tulee ottaa huomioon kaikki testausstrategian puolesta ja testitapauksien osalta asetetut vaatimukset. Lisäksi testiympäristön suunnittelussa ja toteutuksessa tulee huomioida käytettyjen tekniikoiden asettamat vaatimukset.

Tässä työssä testiympäristöllä tarkoitetaan niiden ohjelmien kokonaisuutta, joilla testaus voidaan suorittaa. MDUS-integraatioprojektin testien automaattisen suorittamisen mahdollistava testiympäristö ei vaadi omaa palvelinta toimiakseen, vaan siinä voidaan hyödyntää samaa testiympäristöä, jolla on aiemmin ajettu testejä manuaalisesti. Toisaalta tämän insinöörityön yhteydessä kehitettävä komponentti liitetään osaksi Autotesteriä, jolloin se kulkee mukana kaikille testipalvelimille, joihin Autotesterin tuleva versio

asennetaan. Silloin kyseisiltä testipalvelimilta voidaan ajaa www-sovelluspalveluihin perustuvien järjestelmäkokonaisuuksien testaukseen tarkoitettuja testejä, kunhan testipalvelimilta on vain pääsy kyseisiin www-sovelluspalveluihin.

3.1 Testiympäristön vaatimukset

Yleensä testiympäristöllä pyritään mukailemaan todellista tuotantoympäristöä, jossa kehitettävää ohjelmistoa tullaan lopulta käyttämään. Näin pystytään paljastamaan myös kaikki mahdolliset konfiguraatioista ja ympäristöllisistä tekijöistä peräisin olevat virheet. Edellä mainittuja virheitä voivat olla esimerkiksi palomuurien tai klusterijärjestelmien luomat ongelmat, jotka voivat estää käytettyjen tekniikoiden ja protokollien toimintaa osittain tai kokonaan. (Dustin ym. 2009, s.108–109.)

Testiympäristön suunnittelussa on pyritty käyttämään mahdollisimman pitkälle yrityksen sisällä jo käytössä olevaa tekniikkaa ja hyväksi havaittuja ohjelmistoja, joiden käyttöehdot sallivat vapaan käytön liiketoiminnassa tai joihin on jo olemassa valmiiksi maksetut lisenssit. Myös asetettavat vaatimukset on luotu tätä seikkaa silmällä pitäen. Vain erittäin perustelluista syistä voitaisiin poiketa edellä mainitusta ohjelmistoista. Alustavan tarkastelun perusteella nämä asetetut tavoitteet on mahdollista saavuttaa ilman, että uusia ohjelmistolisenssejä tarvitsisi hankkia.

Testiympäristö perustuu seuraavissa luvuissa olevien vaatimusten ja menetelmien asettamiin ehtoihin. Osa vaatimuksista saattaa tuntua itsestäänselvyyksiltä, mutta niiden on silti täytyttävä, jotta testiympäristö toimisi sille tarkoitetussa käytössä. Tarkoitukseen sopiva testiympäristö pitäisi olla mahdollista toteuttaa siten, että se huomioi kaikki sille asetetut vaatimukset.

3.1.1 Testausstrategian asettamat vaatimukset

Kuten aiemmissa luvuissa on mainittu, olisi järkevää, että testaus olisi järjestelmällistä ja organisoitua ja siitä olisi laadittuna testausstrategia, jota testauksen yhteydessä noudatetaan. Sama asia koskee testauksen automatisointiakin. Olisi järkevää, että siihenkin olisi luotuna strategia, joka tukisi testausstrategiaa ja automatisoinnille asetettuja tavoitteita. Testauksen automatisoinnin strategia sisältää automatisoinnin osalta

testauksen laajuuden, päämäärät, lähestymistavat, testikehyksen ja -ympäristön, työkalut, aikataulun ja henkilöstövaatimukset. (Dustin ym. 2009, s.129.)

Automatisoitavat testitapaukset kuuluvat järjestelmätestauksen piiriin, koska niissä testataan järjestelmän toimintaa käyttötapausten perusteella. Kyseisissä käyttötapauksissa tehdään yleensä yksi yksinkertainen toimenpide, joka testaa tyypillistä toiminnallisuutta järjestelmässä. Enoron testausstrategia määrittelee, että kaikki yksikkötestit tulee olla suoritettuna ja että vaatimukset ja käyttötapaukset toiminnallisuuden osalta ovat tehtyinä ennen järjestelmätestien suorittamista. Lisäksi testaajan tulee olla tarkastanut testitapauksien tulokset.

Testausstrategialla on myös vaikutus käytettävien ohjelmien valintaan. Testausstrategia suosii käytettäväksi sellaisia ohjelmia, kuten SoapUI. SoapUI on mainittu Enoron testausstrategiassa yhdeksi neljästä ohjelmasta, joilla testausta harjoitetaan. Sitä suositellaan käytettävän juuri www-sovelluspalveluiden testaamisessa.

Testiympäristö tulee pystyä toteuttamaan samaan ympäristöön, missä suoritetaan muutkin testit kehitettävän GENERIS-ohjelmiston osalta, koska se tukee ohjelmiston laajamittaista, automaattisesti ajettavaa ja usein toistettavaa testausta. Testausstrategian mukaan korkean riskin toiminnalliset alueet tulisi pyrkiä automatisoimaan ja vain siinä tilanteessa, ettei automatisointi onnistu, voidaan niitä suorittaa manuaalisesti. Näin ollen testausstrategia tukee tämän projektin yhteydessä tarkastelun alla olevien testitapauksien automatisointia. Lisäksi se tukee ohjelman käyttöä, jonka integraatiota Autotesteriin tämän insinöörityön ohessa kehitetään. Nämä asiat eivät tietenkään ole yhteensattumia, vaan yhtenä tämän projektin toteuttamisen taustatekijänä toimii luonnollisesti Enoron testausstrategia.

3.1.2 Tekniikan asettamat vaatimukset

Käytössä olevat tekniikat asettavat myös omat vaatimukset kehitettävälle testausautomaatiolle ja sitä varten pystytettävälle testijärjestelmälle. Nykyiset testijärjestelmät toimivat Windows Server -ympäristössä, joka tarkoittaa sitä, että myös pystytettävän testiympäristön tulee olla täysin Windows Server -yhteensopiva. Lisäksi, kuten jo aiemmissa luvuissa on mainittu, tulee testijärjestelmän tukea www-sovelluspalveluiden käyttöä, jotta niihin kohdistuvia testejä voidaan suorittaa.

Enoro Oy:n sisällä kehitetty Autotester-sovellus asettaa myös rajoituksia sille, minkälaista ohjelmistoa voidaan harkita käytettäväksi yhdessä sen kanssa. Autotester tarjoaa esimerkiksi mahdollisuuden käynnistää komentoriviohjelmia. Se mahdollistaa kohtalaisen laajan tuen Windows-maailman ohjelmistoille. Lisäksi Autotesteriä voidaan kehittää siten, että sen kautta voidaan ajaa ohjelmia, jotka tarjoavat jonkinlaisen liityntärajapinnan komentojen vastaanotolle ja raportoinnille. Useat ohjelmat tarjoavat joko komentorivipohjaisen käynnistimen tai vaihtoehtoisesti komentoriviparametreja, joilla graafisen ympäristön ohjelma saadaan käynnistettyä ja ajettua haluttujen toimintojen kera. Niinpä tässä yhteydessä voidaan hyödyntää vain sellaisia ohjelmistoja, jotka tarjoavat jonkinlaisen liitynnän ohjelman ajamiselle toisesta ohjelmasta käsin. Lisäksi käytettävästä ohjelmasta tulee pystyä raportoimaan saatu testidata jossakin tarkoitukseen sopivassa formaatissa.

Insinöörityön yhteydessä toteutettava käytännön sovellus on sidoksissa projekteihin, joissa käytetään www-sovelluspalveluiden välillä tapahtuvaa kommunikointia. Kommunikointi tapahtuu SOA (Service Oriented Architecture) -käytäntöjen mukaisesti. Siinä kahden erilaisen järjestelmän välillä tapahtuvalle kommunikaatiolle on sovittu aina formaatti, jota molemmat järjestelmät osaavat tulkita. SOA:n tapauksessa tieto muodostetaan tietynlaiseen xml-formaattiin, jonka mukaisia tiedostoja siirretään järjestelmien välillä. Tuolloin järjestelmä, joka haluaa vastaanottaa tietoa, lähettää ensin palvelupyynnön (service request) kohdejärjestelmän yhteen liittymään (interface) ja ilmaisee pyynnössä, mitä tietoa kohdejärjestelmän halutaan toimittavan. Sen perusteella kohdepalvelu palauttaa vastauksen (service response) pyynnön esittäneen järjestelmän www-sovelluspalvelupyynnön lähettäneeseen liittymään. Tiedon siirtoon käytetään verkkoliikenteestä tuttua http-protokollaa (Hyper Text Transfer Protocol). (Sprott & Wilkes 2004.)

WWW-sovelluspalveluiden välillä tapahtuvan kommunikaation testaukseen vaaditaan testijärjestelmältä tuki http-protokollan mukaiselle verkkoliikenteelle sekä avoin yhteys palomuurin läpi niihin www-sovelluspalveluihin, joita halutaan testata. Lisäksi testiohjelman tulee ymmärtää testattavien SOA-käytännön toteuttavien palveluiden yhteydessä käytettävää SOAP (Simple Object Access Protocol) -formaattia ja sen sisältöön liittyvän datan asettamia vaatimuksia, jotka yleensä määritetään testitapauksien yhteydessä niitä testattavalle ohjelmalle, jotta ohjelma voi päätellä testin lopputuloksen.

3.1.3 Raportoinnin asettamat vaatimukset

Testien ajon jälkeinen raportointi tulee huomioida kokonaisuutta ajateltaessa. Kun testi on ajettu onnistuneesti läpi, ei ajosta välttämättä tarvita juuri muuta tietoa onnistumisen lisäksi. Sen sijaan testistä, jota ei läpäisty, tulisi antaa lisätietoa siitä, mikä on testin läpäisemättömyyden syy. Automaattiseksi tehdyn testin epäonnistuminen ei aina välttämättä tarkoita sitä, että ohjelmiston toiminnassa olisi puutteita. Syy virheeseen voi löytyä myös siitä, että ohjelmisto on muuttunut joltain osin eikä automaattinen testi siksi enää vastaa ohjelmiston toimintaa.

Hyvin raportoidut virheet edesauttavat testin epäonnistumisen todellisen syyn selvittämisessä. Lisäksi ne edesauttavat sitä, että sovelluskehittäjä pystyy nopeammin paikallistamaan virheen alkuperän. Tuolloin sovelluskehittäjä pystyy paremmin korjaamaan juuri oikean virheen ja varmistamaan, ettei korjaus riko samaa toiminnallisuutta jollain toisella tavoin. Siksi on erityisen tärkeää, että virheistä kerrotaan riittävän tarkasti ja mukana on tieto kaikista oleellisista asioista, jotka ovat saattaneet vaikuttaa virheen ilmenemiseen. (Tatham 1999; Whitmill 2010.)

Sovelluksen puutteiden tehokas raportointi saavutetaan, kun raportoinnissa otetaan huomioon muutamat asiat. Puutteet tulisi raportoida selkeästi mutta lyhyesti. Kaikista puutteista tulisi kuitenkin tarjota riittävästi informaatiota, jotta virheestä paljastuisi muun muassa se, onko vika ollut annetussa syötteessä vai toimiiko kohteena oleva ohjelmisto väärin. Lisäksi virheraporttien tulisi olla ammattimaisia, eikä niissä tulisi viljellä ylimääräisiä selittelyitä. Virheraportista tulisi myös ilmetä, miten virhe voidaan toistaa, mitä virheestä on selvinnyt tähän asti ja tarvittavat lisätiedot mitkä auttavat virheen korjaamisessa. (Whitmill 2010.)

Virheiden raportointi asettaa vaatimuksia myös testaukseen käytettävälle ohjelmistolle. Jotta testaajat voisivat koostaa hyviä virheraportteja (defect reports) ajetuista testikokonaisuuksista ja havaita helposti ne virheet, jotka eivät ole peräisin ohjelmiston toiminnan puutteista, tulee Autotesteriin integroitavan ohjelman pystyä raportoimaan ajettujen testien tiedot tehokkaasti ja siten, että mukana on riittävä määrä informaatiota. Autotester odottaa, että mahdollisen virheen sattuessa, tieto virheen sattumisesta raportoidaan ja mukana voidaan antaa tekstinä lisäinformaatiota. Onnistunutta testitapausta ei tarvitse raportoida mitenkään, koska testin ajaminen ilman virheestä ilmoittamista tulkitaan onnistuneeksi testiksi.

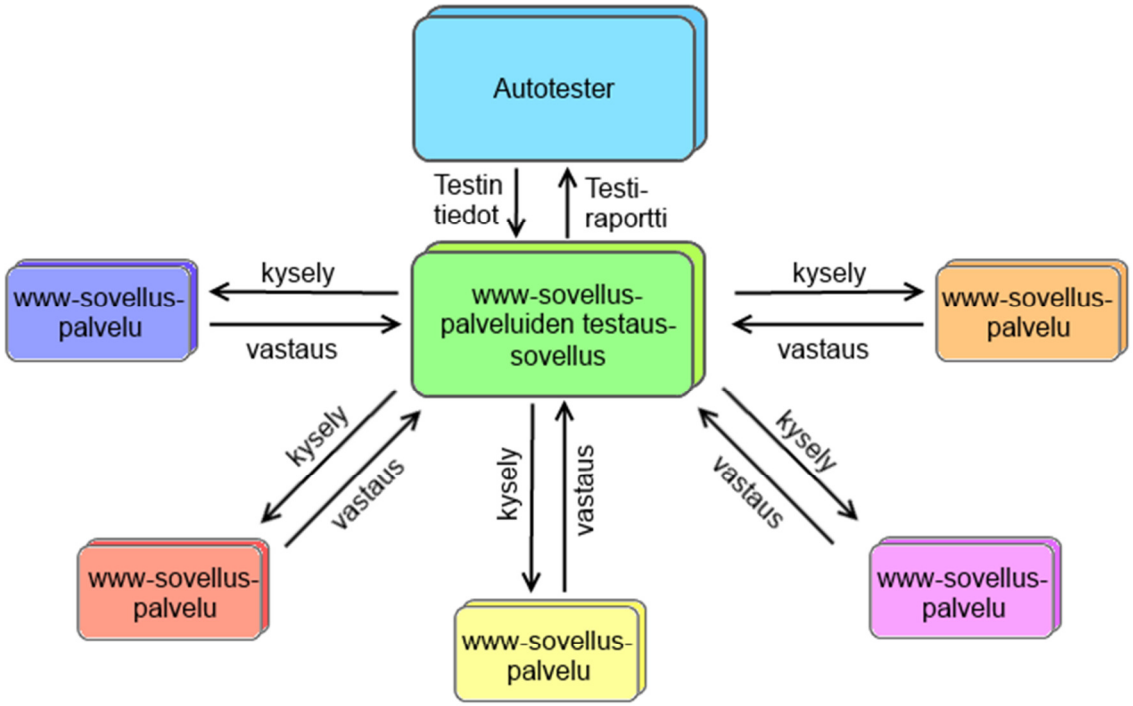
Kuten odottaa saattaakin, ei nykyisten, tämän insinöörityön kontekstiin liittyvien testien ajoon käytettävästä ohjelmistosta raportoida testin tulosdataa tarkoitukseen sopivassa formaatissa. Siksi testiohjelmasta saatu tieto tulee prosessoida, suodattaa ja lopulta muuttaa Autotesterille sopivaan muotoon kulloisenkin tilanteen mukaan. Koska ennen testin suorittamista ei tiedetä, mitä informaatiota lopulta Autotesterille halutaan välittää, tulee testiohjelmalta pyrkiä keräämään informaatiota mahdollisimman laajasti, jotta kaikki oleellinen tieto on käytettävissä testin epäonnistuessa. Lisäksi ennen testiohjelman käynnistämistä tulisi tarkistaa syötetyn tiedon oikeellisuus sekä se, että kaikki tarvittava informaatio on annettu, jotta testi voitaisiin ajaa onnistuneesti. Testi voidaan tällöin hylätä nopeammin ja mahdolliset puutteet raportoida jo ennen kuin varmasti epäonnistumiseen päätyvää testiä tarvitsee edes yrittää ajaa testiohjelmalla. Lisäksi näin toimiessa voidaan vaikuttaa siihen, että virheet paikallistetaan ja raportoidaan riittävällä tarkkuudella, jotta niiden korjaaminen olisi mahdollisimman helppoa ja tehokasta.

3.2 Suunnitelma testijärjestelmän toteuttamisesta

Insinöörityön yhteydessä kehitettävän sovelluskomponentin testaaminen ja käyttäminen vaatii www-sovelluspalvelu (web service) -tyyppisen kommunikaation toteuttamista. Kehitettävän komponentin lopullinen järjestelmätestaus sisältää Enoron olemassa oleviin projekteihin liittyvien testitapausten suorittamista ja tulosten validoimista suhteessa vastaavien testien manuaalisten suoritusten tuloksiin. Testaaminen käytännössä on toteutettu siten, että testijärjestelmien liittymärajapintoihin lähetetään testidataa erilaisilla pyynnöillä varustettuna, ja siitä vastaanotetaan vastauksia, joiden tuloksia analysoidaan. Jos vastaukset ovat haluttuja, on testi suoritettu onnistuneesti, muussa tapauksessa testin merkitään päättyneen virheellisesti.

Kuvassa 4 on visualisoitu suunnitelma siitä, miten testijärjestelmän tulisi toimia käytännössä. Se on laadittu olemassa olevien tarpeiden ja vaatimusten perusteella. Kuvassa Autotester antaa testien tietoja www-sovelluspalveluiden testaussovelluksella, joka suorittaa niiden perusteella testitapauksia, joissa kysellään testattavilta www-sovelluspalveluita tietoja ja niiltä odotetaan takaisin vastauksia. Saadut vastaukset analysoidaan ja niiden perusteella lähetetään testiraportti takaisin Autotesterille. Yksi testisarja saattaa sisältää useita kyselyitä saman www-sovelluspalvelun useisiin liittymiin tai vastaavasti useita testitapauksia, joissa testataan eri www-sovelluspalveluita. Riippu-

matta siitä, miten testikokonaisuudet on koottu, tulisi niiden testauksen onnistua ongelmitta Autotesteristä käsin.



Kuva 4. Suunnitelman mukainen testijärjestelmä.

SOA-tyyppisen (Service Oriented Architecture) kommunikoinnin testaukseen on tehty erilaisia ohjelmia. Yksi IT-alan yrityksissä laajalti käytössä oleva testausohjelma on ilmainen, avoimeen lähdekoodiin perustuva SoapUI, joka soveltuu tämänkaltaiseen tarkoitukseen erittäin hyvin. SoapUI mahdollistaa automaattisen testitapauksen tai -sarjan ajamisen yhtä painiketta painamalla. SoapUI:n graafista käyttöliittymää on helppo käyttää ja se voidaan integroida osaksi esimerkiksi Maven-, NetBeans-, Jboss- ja Eclipse-kehitysympäristöjä. Lisäksi SoapUI:hin tehtyjä testitapauksia voidaan ajaa komentorivityökalun, SoapUI TestRunnerin avulla. Se tekee SoapUI:n testitapausten ajosta helposti automatisoitavia. (Patton 2008.)

GENERIS-järjestelmään liittyvien toiminnallisuuksien testaaminen vaatii GENERIS-järjestelmän asennuksen ja siihen liittyvän Oracle-tietokannan olemassaolon testijärjestelmässä. Kun halutaan testata GENERIS-järjestelmään liittyviä www-

sovelluspalveluita, tulee GENERISin lisäksi olla asennettuna ja konfiguroituna Generis Information Service. Se on GENERISin www-sovelluspalvelu, joka hoitaa kommunikointia muiden www-sovelluspalveluiden ja GENERISin välillä. Näin ollen Generis Information Service on se www-sovelluspalvelu, jota testausohjelmalla voidaan esimerkiksi testata.

4 GENERIS-järjestelmään liittyvä testaaminen

Autotester on Enoron keskeinen testaustyökalu tai -ympäristö. Se on alun perin kehitetty Enoron toimesta GENERIS-järjestelmän monipuoliseen testaukseen. Sittemmin Autotesteriä on haluttu kehittää yhä kattavammaksi ja kykeneväksi testaamaan yhä laajemmin GENERISin yhteydessä käytettäviä tekniikoita. Autotester on Enoron testisuunnitelmassa keskeinen testiympäristö, josta regressio- ja kertaluontoisia testejä ajetaan.

Autotester toimii siten, että se ajaa yhden tai useamman Autotesterin formaatissa tehdyn ATScript-komentosarjatiedoston, joista saadaan tiedot läpimenosta ja mahdollisista virheistä. Autotesteriä varten on kehitetty oma skriptauskieli, jolla ATScriptejä kirjoitetaan. Se on hyvin yksinkertainen kieli, joka koostuu pääosin määritettävistä vakioista ja Autotesterin tarjoamien funktioiden suorittamisesta. Kutsuttaville funktioille annetaan parametreja, joiden perustella ne toimivat.

Uuden toiminnallisuuden integroiminen Autotesteriin tarkoittaa siis käytännössä sitä, että Autotesteriin lisätään funktioita, jotka mahdollistavat uusien testityyppien suorittamisen. Funktioiden vastaanottamat parametrit määrittävät esimerkiksi sen, mitä testejä ajetaan tai niiden avulla voidaan määrittää tarkka toimintaketju, joka suoritetaan GENERISin käyttöliittymässä. Siten voidaan esimerkiksi luoda testi, jolla navigoidaan johonkin tiettyyn sijaintiin GENERISin sisällä ja tehdä haluttuja toimenpiteitä, kuten nappien painalluksia ja arvojen syöttämisiä aivan, kuin ne olisi tehty suoraan GENERISin käyttöliittymästä. GENERIS-järjestelmään on sisällytetty toiminnallisuuksia, jotka pystyvät palauttamaan tiedon tehtyjen toimintojen onnistumisesta, jolloin testin lopputulos pystytään määrittelemään saadun tiedon avulla.

Autotesteriä kehitetään edelleen ja siitä halutaan luoda järjestelmä, joka kykenee täysin automaattisesti ajamaan kaikkia Enorolle oleellisia testejä. Lisäksi Autotesterin tulisi tulevaisuudessa kyetä raportoimaan suoritetuista testeistä itsenäisesti, esimerkiksi sähköpostilla, ilman, että sen toimintaan tarvitsisi välttämättä puuttua päivittäin. Siksi Autotesteriin halutaan nyt integroida ominaisuuksia, joilla voidaan ajaa esimerkiksi www-sovelluspalveluita testaavia testejä ilman ylimääräisiä välikappaleita. Silloin www-sovelluspalveluiden testauksen osalta on päästy lähemmäksi sitä pitkän tähtäimen päämäärää, jota kohden testausta Enorolla halutaan viedä.

4.1 www-sovelluspalveluiden testaukseen käytetyt testaustyökalut

Kaikkia testien suorittamiseen vaadittavia toiminnallisuuksia ei ole järkevää kehittää tyhjästä, jotta Autotesterillä voitaisiin testata kyseisiä testejä. Joidenkin testien suorittamiseen on luontaista käyttää olemassa olevia, esimerkiksi avoimen lähdekoodin ohjelmia silloin, kun kyseiset kolmannen osapuolen ohjelmat voidaan ikään kuin sulauttaa yhteen Autotesterin kanssa. Tämä tarkoittaa käytännössä sitä, että Autotesteristä voidaan suorittaa testejä ilman, että ulkoisten ohjelmien käyttöliittymiä tarvitsisi avata erikseen.

Autotesterillä ei ole tähän mennessä pystytty testaamaan kaikkia tarvittavia toiminnallisuuksia. Koska meneillä oleviin projekteihin liittyviä ominaisuuksia pitää pystyä testaamaan, eikä aina ole mahdollista kehittää Autotesteriä vastaamaan kaikkiin tarpeisiin ainakaan samalla aikataululla, on Enorolla otettu käyttöön testien suorittamiseksi muitakin tarjolla olevia testausohjelmia. Pitkän tähtäimen ratkaisuna ei haluta kuitenkaan ylläpitää useampaa järjestelmää, josta testejä ajettaisiin, ellei sille ole erittäin hyvää perustetta.

Testejä on jo tehty huomattava määrä kolmansien osapuolien ohjelmistoille, joita Enoron kehittämien sovellusten testauksessa on otettu käyttöön. Olemassa olevien testien käyttäminen halutaan mahdollistaa jatkossakin, jottei jo tehty työ menisi hukkaan. Siksi on luontaista pyrkiä integroimaan juuri ne työkalut Autotesteriin, jotka ovat jo olleet käytössä testaajilla. Näitä kolmansien osapuolien työkaluja on useita, mutta seuraavissa luvuissa käsitellään vain tämän insinöörityön kontekstille oleelliset työkalut.

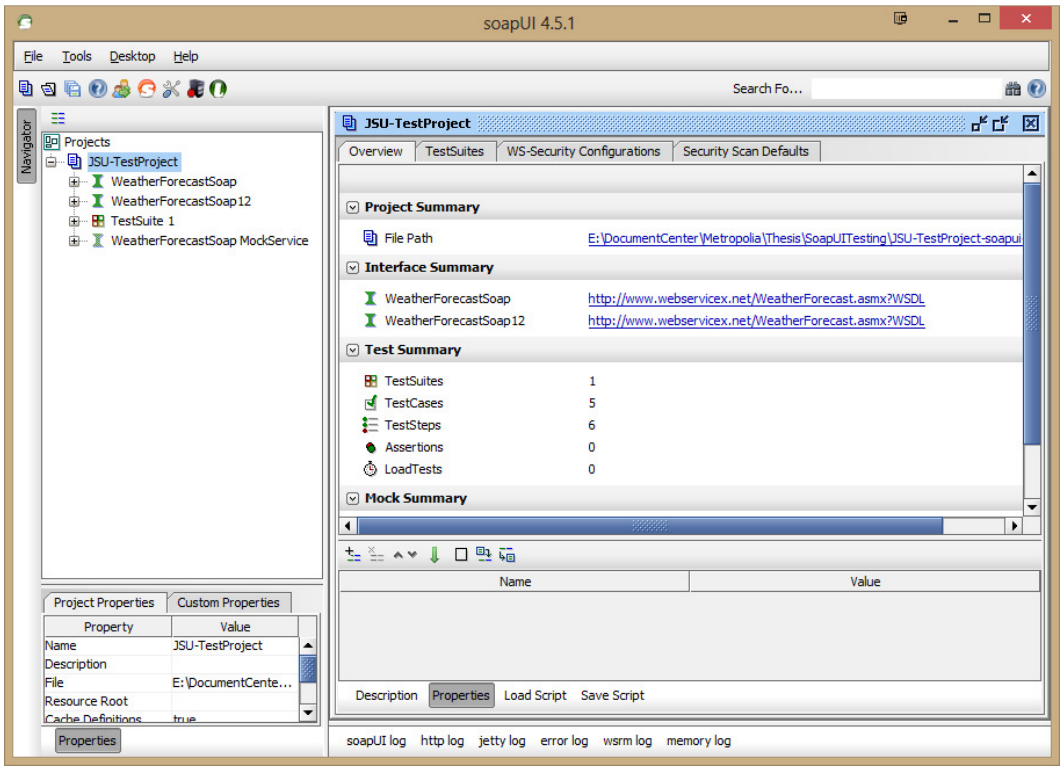
4.1.1 SoapUI

Kuten aiemmissa luvuissa on aihetta jo sivuttukin, SoapUI on graafinen työkalu www-sovelluspalveluiden eli niin kutsuttujen SOAP-palveluiden testaukseen. Sen käyttö on pienen opettelun jälkeen suhteellisen helppoa ja se mahdollistaa useiden testitapausten peräkkäisen suorittamisen. SoapUI:ssa on panostettu helppokäyttöisyyteen. Sillä pystytään helposti lisäämään testauslistalle kaikki www-sovelluspalveluun liittyvät liittymät (interfaces) ilmoittamalla pelkästään WSDL:n (Web Service Description Language) mukaisesti toteutetun www-sovelluspalvelun osoite sille tarkoitettuun kenttään. Tämän jälkeen SoapUI:n käyttöliittymästä voi luoda testitapauksen jokaiselle tarjolla

olevalle liittymälle. Näin ollen palvelun kaikkien ulkoisten liityntöjen testaus onnistuu suhteellisen pienellä vaivalla. (Working with SoapUI – Getting Started 2013.)

SoapUI toimii siten, että se lähettää www-sovelluspalveluun ennalta määritetyn pyynnön ja odottaa palvelusta takaisin vastauksena pyynnössä esitettyä informaatiota. Kun vastaus saapuu, sitä verrataan vertailutulokseen ja testiaskeleen lopputuloksen todetaan siten olevan joko hyväksytty, hylätty. Tuloksen voidaan katsoa olevan myös epävarma, mikäli vastausta ei saavu kohtuullisessa ajassa pyynnön lähettämisestä. Testien lähettämissä pyynnöissä voidaan käyttää myös tarkoituksellisesti vääränlaista tietoa ja odottaa, että www-sovelluspalvelu hylkäisi pyynnön. Siten voidaan varmistaa myös se, että www-sovelluspalvelu selviää mahdollisesta virhetilanteesta tai häiriöstä. (Working with SoapUI – Getting Started 2013.)

SoapUI:ta ei ole ensisijaisesti tarkoitettu suorituskykytestauksiin. Suorituskykytestausta varten SoapUI:n tekijä SmartBear tarjoaa erillistä LoadUI-työkalua, joka voidaan integroida osaksi SoapUI:ta. SmartBear tarjoaa SoapUI:sta sekä avoimen lähdekoodin kevytversiota, että maksullista pro-versiota. Pro-versioon tarjotaan mukaan paitsi tuotetukea, myös monia ammattimaista testausta ja sen raportointia helpottavia työkaluja. SoapUI:n ilmaisella versiollakin pääsee pitkälle mutta toisinaan monimutkaisempien toiminnallisuuksien toteuttamisessa voi joutua turvautumaan joko ohjelmointiin tai maksulliseen versioon. SoapUI:n graafinen käyttöliittymä on suhteellisen pelkistetty. Se on nähtävillä kuvassa 5. (About SoapUI – Features 2013.)



Kuva 5. SoapUI:n graafinen käyttöliittymä.

SoapUI:n projektitiedostot ovat xml-formaatissa. Niinpä projektitiedostoista on luettavissa millä tahansa tekstieditorilla, mitä ne sisältävät. Se mahdollistaa projektitiedostojen muokkauksen esimerkiksi ulkopuolisessa ohjelmassa. Lisäksi samoja projektitiedostoja voitaisiin ajaa myös muissa ohjelmissa, joihin on kehitetty tuki kyseisille tiedostoille. Tuen kehittämistä helpottaisi lisäksi se, että SOAP-protokollan www-sovelluspalveluiden viestintään käytettävien pakettien formaatti on standardoitu ja samaa formaattia käytetään siten myös SoapUI:n projektitiedostojen elementeissä.

SoapUI on luotu avoimen ohjelman filosofian mukaisesti. Sen avoimuutta tukee myös se, että SoapUI:hin on sisällytetty paitsi valmiita testausominaisuuksia, siinä voidaan myös tehdä kokonaan omia skriptejä, jotka mahdollistavat valmiita ratkaisuja huomattavasti monipuolisempien testien tekemisen. Skriptien kieli on Javalla tehty, yleisesti käytössä oleva Groovy Script. Lisäksi SoapUI:sta voidaan tehdä suoraan tietokantakyselyitä standardia SQL:ää (Structured Query Language) hyödyntäen. Se auttaa esimerkiksi tietokannan muuttuneiden tietojen verifioimisessa www-sovelluspalveluun lähetetyn pyynnön jälkeen.

4.1.2 SapSim

SapSim on testausohjelma, joka on kehitetty Enoron Norjan yksikössä. Se on aiemmin palvellut muita tarkoituksia, mutta sittemmin sitä on alettu hyödyntää SAP-MDUS-integraatioprojektin testien ajossa. SapSim:in suunnittelussa ei ole huomioitu miten-kään Autotester-yhteensopivuutta, koska sitä on alun perin kehitetty eräässä Enoroon fuusioituneessa yhtiössä, kun taas Autotesteriä on kehitetty toisessa fuusioituneessa yhtiössä. Tällä hetkellä SapSimin tulevaisuus on avoin, koska siitä ei ole tehty päätök-siä. Vaihtoehtoina ovat lopettaa kehitys kokonaan tai jatkaa ylläpitoa ja mahdollisesti jopa tehdä lisäkehitystä.

SapSim:in kehityksessä on sittemmin keskitytty siihen, että se vastaisi paremmin juuri SAP-MDUS-integraatioprojektin tarpeita. Nykyisen testauksen puitteissa se on ajastet-tu ajettavaksi päivittäin. SapSim testaa automaattisesti GENERISin MDUS-rajapinnan suorituskykyä sekä muita siihen liittyviä toiminnallisuuksia ja raportoi automaattisesti päivittäin testien tulokset suoraan testaajien sähköpostiin. Lisäksi SapSim hyödyntää SoapUI:ta ajamalla SoapUI:n kautta joitain testejä. SapSim kykenee tekemään myös suorituskykytestejä ilman SoapUI:ta.

Toisin kuin nimestä saattaisi päätellä, SapSim ei pelkästään matki SAP:n toimintaa. Se pystyy lisäksi matkimaan (mocking) myös GENERISin toimintaa SAP:in ja GENERISin välisessä kommunikoinnissa. Näin ollen GENERISin toimintojen testausta on voitu har-joittaa jo silloin, kun GENERISin kaikki kommunikointiin vaativat toiminnallisuudet eivät ole olleet vielä valmiina. SapSim sisältää oman SOAP-palvelun (www-sovelluspalvelun), johon se voi vastaanottaa pyyntöjä esimerkiksi GENERISiltä. SOAP-palvelu mahdollistaa SAP:in ja GENERISin välisessä kommunikoinnissa sellaisen tilan-teen testaamisen, jossa palvelupyyntö lähtee GENERISiltä. Lisäksi GENERISin tar-joamat vastaukset palvelupyyntöihin voidaan vastaanottaa tämän palvelun kautta. (SapSim Guide 2013.)

SapSim:in ehdottomiin vahvuuksiin kuuluu se, että siihen on integroitu oma kevytraken-teinen tietokanta. SapSim:in omaan tietokantaan kirjataan lähetetyt palvelupyynnöt ja vastaanotetut, pyyntöjä vastaavat vastaukset. Palvelupyynnöllä ja sitä vastaavalla vas-tauksella on SapSim:issä ja järjestelmien välisessä viestinnässä yhteinen tunnus, jolla ne voidaan yhdistää toisiinsa. Se mahdollistaa testitulosten selvittämisen, kun tiede-tään, mitkä pyynnöt ja vastaukset liittyvät toisiinsa. (SapSim Guide 2013.)

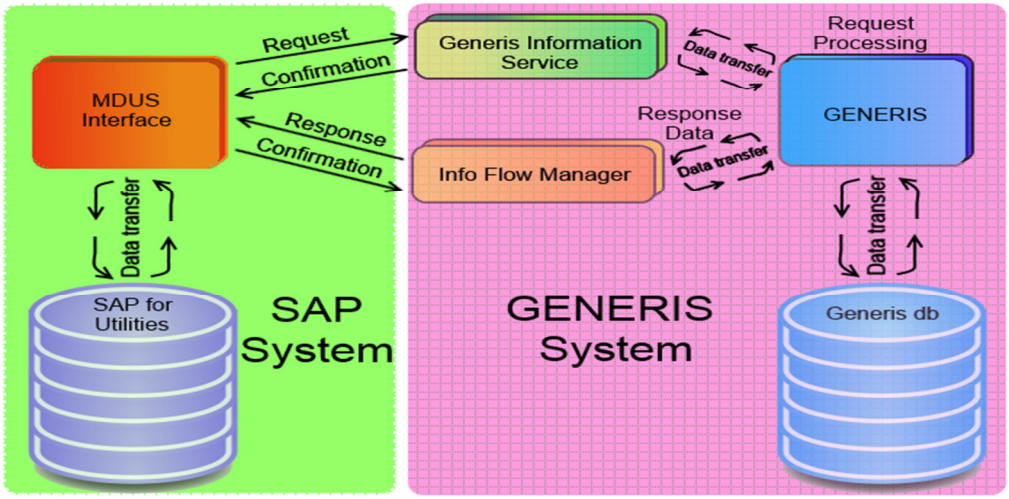
Kuten aiemmin jo mainittiinkin, SapSim ei nykyisellään integroidu Autotesteriin. Sen ajaminen Autotesteristä on mahdollista mutta ei optimaalista. SapSim voidaan käynnistää Autotesteristä komentorivikäynnistyksen kautta, mutta sen toimintaan ei voida mitenkään vaikuttaa Autotesteristä käsin. Tämä tarkoittaa käytännössä sitä, että varsinainen integraatio jää toteutumatta, kun ajettavaksi valittavia testejä ei voida määritellä Autotesteristä käsin, vaan ne vaativat SapSimin erillistä konfiguroimista. Kunnollisen integraation kehittäminen taas vaatii suhteellisen paljon töitä, eikä vielä ole tehty päätöstä, tullaanko sellainen toteuttamaan.

4.2 SAP-MDUS-integraatioprojekti

Insinöörityön yhdeksi tavoitteeksi on mainittu selvitystyön tekeminen siitä, miten SAP-MDUS-integraatioprojektiin liittyvä testaaminen voitaisiin liittää osaksi regressiotestejä tai tarkemmin kuvailtuna, miten nykyisin käytettävät testausmenetelmät ja -toiminnot saataisiin integroitua Autotesteriin. Jotta sellaisen selvitystyön tekeminen olisi mahdollista, tulee ensin ymmärtää kyseiseen projektiin liittyvät toiminnallisuudet ja niiden toiminta pääpiirteittäin. Toisin sanoen, pitää ymmärtää, mitä testataan ja miten.

Aiemmissa luvuissa on mainittu, että SAP:n MDUS-liittymän (MDUS interface) ja GENERISin vastaavan liittymän välinen kommunikaatio tapahtuu SOAP-protokollan mukaisesti www-sovelluspalveluiden sisältämien liittymien kautta. Kuvassa 6 on konkretisoitu kyseisten järjestelmien välinen kommunikaatio astetta tarkemmin kuvattuna.

GENERISissä MDUS-rajapinnan muodostavat Generis Information Service, joka toimii GENERISin ”kuuntelijana” ulkoa päin tuleville pyynnöille. Kun SAP:ltä tulee pyynnön muodossa kehotus suorittaa jokin toimenpide, sen suorittamisessa saattaa kestää jonkin verran aikaa. Siksi vastausta SAP:lle ei voida antaa välittömästi vaan takaisin lähetetään vain vahvistus siitä, että viesti on tullut perille. GENERISin komponentti nimeltä Info Flow Manager pystyy suorittamaan monipuolisesti erilaisia tehtäviä, jotka on voitu ajastaa ajettaviksi tai jotka käynnistyvät jonkin herätteen perusteella. Näin ollen MDUS-rajapinnan osalta Info Flow Manager on se komponentti, joka lähettää valmiin pyydetyn datan SAP-järjestelmälle, kun se on valmis lähetettäväksi. Tieto lähetetään SAP:n MDUS-rajapinnan oikeaan liittymään ja sieltä palautetaan vahvistus siitä, että tieto on tullut perille.



Kuva 6. SAP:n ja GENERISin kommunikaatioprosessi.

Kuvasta 6 on havaittavissa myös viitteitä siitä, miten MDUS-liittymän testaaminen tulisi suorittaa. Järjestelmien tiedonsiirto ei tapahdu yksittäisessä pyyntö-vastaus-ketjussa. Sen sijaan kommunikointiin voidaan tarvita kahta erillistä pyynnön ja vastauksen sisältävää tapahtumaa, mikäli lähettävä järjestelmä odottaa jotain informaatiota takaisin sen sijaan, että se antaisi vain yksittäisen komennon, jonka vastaanottokuittaus riittäisi vastineeksi. Näin ollen testauksen näkökulmasta pitäisi olla olemassa varmuus siitä, että kokonainen tiedonvaihtotapahtuma on käsitelty loppuun asti ennen kuin voidaan verifioida, onko testin lopputulos validi.

Tapahtuman riippuvuus toisesta tapahtumasta tuo mukanaan haasteen erityisesti testien automatisoinnin kannalta. Ihmistestaaja tunnistaisi testin epäonnistumisen mahdolliseksi syyksi heti sen, että jälkimmäisen tapahtuman tarkistus on tehty liian nopeasti, jolloin testaaja voisi ajaa testin vähän ajan kuluttua uudelleen tai tarkistaa järjestelmän todellisen tilanteen. Sen sijaan automatisoidussa testissä ilman synkronisoituja tapahtumia, ainoa tapa olisi arvata, koska prosessin pitäisi olla valmis. Arvaaminen taas ei ole kovin tehokas tapa automatisoida testejä, koska se tuo epävarmuutta ja lisää testiin kuluvaa aikaa, eikä se mukaudu tilanteeseen järjestelmän kehittyessä.

SAP-MDUS-integraatioprojektissa on jo käytössä automaattinen ratkaisu nykyisin suoritettaville testeille. Testaajat ovat käyttäneet SapSim:ia Windowsin tehtävien ajastuksella ajastettuna jokapäiväiseen testaukseen. Sen lisäksi testejä on käynnistetty SoapUI:lla manuaalisesti testaajien toimesta. SapSim on kehitetty käyttämään samoja pro-

jektitiedostoja, kuin SoapUI. Se mahdollistaa kerran SoapUI:lle tehtyjen testien viemisen osaksi SapSim:in automaattista testausprosessia.

SapSimin puute on kuitenkin se, että se on erillinen järjestelmä Autotesterin näkökulmasta. Syy, miksi SapSim:iä ei voida kovin helposti jättää pois MDUS-rajapinnan testauksesta, on se, että se kykenee tekemään testejä synkronoidusti, mihin taas SoapUI:lla ei pystytä. Käytännössä tämä tarkoittaa sitä, että SapSim on tietoinen siitä, milloin GENERIS on käsitellyt sille lähetetyn palvelupyynnön, koska sillä on sisäinen tietokanta, joka ylläpitää kyseisiä tapahtumia ja niihin linkattuja pyyntöjä. Mahdollisuus synkronoituun testaamiseen on ollut erityisen oleellinen toiminnallisuus sen jälkeen, kun GENERISin MDUS-rajapinnan toiminnot ovat monimutkaistuneet siten, että yhden palvelupyynnön käsittely saattaa kestää kohtalaisen pitkään ja vastaus kyselyyn pystytään lähettämään GENERISistä vasta huomattavasti pidemmän viiveen jälkeen.

SoapUI:n tapauksessa ainoa mahdollisuus suorittaa SapSimin ajamia testejä, jotka koostuvat usein kahdesta SoapUI-testitapauksesta, olisi harjoittaa niin kutsuttua kiertokyselyä (polling). Kiertokyselyssä SoapUI tarkistaisi jatkuvasti, onko ensimmäisen testin pyytämä toiminto jo käsitelty ja siihen liittyvä vastaus saapunut. Tämä tarkoittaisi käytännössä sitä, että testissä jäätäisiin odottamaan vastausdataa niin pitkäksi ajaksi, kunnes tieto olisi saapunut.

Kiertokyselytoiminnallisuuden kehittämien SoapUI:ssa käytettäväksi vaatisi tarkoitukseen sopivan skriptin kirjoittamisen sekä mahdollisten virhetilanteiden huomioimisen, jottei testiohjelma pysähtyisi kokonaan silloin, kun vastausta ei kuulu koskaan. Se ei myöskään ole kovin tehokasta, että yhtä testiä odotellaan pitkään, koska jo nykyisellään vastauksien saapuminen saattaa kestää pitkään. Tämän lisäksi SoapUI:sta ja Autotesteristä puuttuu oman SOAP-palvelun pystyttämisen mahdollisuus, jolloin vastauksen paluukanavaa ei ole toteutettu testaukseen sopivalla tavalla. Näin ollen nykytilanteessa SapSim:ia täytyy ylläpitää osana SAP-MDUS-integraatioprojektin testejä.

4.3 Muut projektit, jotka sisältävät www-sovelluspalvelukomponentteja

Insinöörityön keskeisimmät tavoitteet liittyvät www-sovelluspalveluiden testaukseen keskittyvien testien automatisoiminen ja liittäminen osaksi Autotesterillä suoritettavia testejä. Niihin liittyvää testaamista on tähän asti hoidettu SoapUI:n kautta siten, että

testaaja on käynnistänyt testit manuaalisesti. Nykyisellään testien liittäminen osaksi laajempia regressiotestejä ei onnistu – ei ainakaan millään järkevällä keinolla.

Jotta insinöörityön keskeisimmät tavoitteet saavutettaisiin, insinöörityöhön sisältyy sovelluskehitystä. Sovelluskehitys sisältää testiympäristön suunnitelmassa esitetyn kuvan mukaisen testijärjestelmän vaatimien toiminnallisuuksien kehittämisen. Samaa kehitettävää komponenttia tulee voida hyödyntää kaikkien niiden projektin testien automatisoinnissa ja liittämisessä osaksi regressiotestejä, jotka sisältävät www-sovelluspalveluiden testaamista tai jolle on jo laadittu testejä ajettavaksi SoapUI:ta hyödyntäen. Näin ollen insinöörityö palvelee oikeastaan useampia projekteja kuin insinöörityön yhteydessä on mainittu.

4.4 SoapUI:n käyttö tulevaisuudessa

SAP-MDUS-integraatioprojektin testien liittämiseksi osaksi laajempia regressiotestejä oli alkuperäisen selvityksen jälkeen kaavailtu myös insinöörityön yhteydessä kehitettävää SoapUI:n Autotester-integraation mahdollistavaa sovellusta. Tuolloin SoapUI:lla olisi voitu korvata SapSim:in käyttö osittain tai jopa kokonaan. Tavoite oli selkeä, koska SoapUI:n integroiminen Autotesteriin on suoraviivaisempaa. SoapUI:n testien ajoa pystytään paremmin hallitsemaan ulkoisesti. Syvällisempi selvitys kuitenkin paljasti ne pullonkaulat, jotka vaativat SapSim:in mukana oloa SAP-MDUS-integraatiotesteissä myös tulevien regressiotestien osalta, eikä pelkästään SAP:n rajapinnan osalta vaan myös Generis (MDUS) -rajapinnan osalta.

Toinen oleellinen asia on se, että SoapUI:lle tehtyjä testejä käytetään myös Enoron muiden asiakkaiden projekteissa ja siksi mahdollisuudelle ajaa SoapUI:n testejä Autotesteristä käsin on olemassa laajempikin tarve. Autotesterin lähiaikoina kehitettävien ominaisuuksien listalle oli siten toisten projektien puitteissa kirjattu SoapUI-tuen kehittäminen, joka selvisi toteutukseen liittyvän selvitystyön ohessa.

Edellisten seikkojen perusteella tämän insinöörityön keskeisimmäksi tavoitteeksi on tarkentunut Autotesterin SoapUI-projektien ajon tuen tekeminen integroituna toiminnallisuutena sen sijaan, että se toteutettaisiin erillisenä komentorivityökaluna. SoapUI:ta on siis tarkoitus hyödyntää jatkossakin osana Enoron testaustyökalupakkia. Tästä johtuen myös tehtävälle sovelluskomponentille, joka mahdollistaa SoapUI-testien ajami-

sen Autotesteristä käsin, on odotettavissa pitempi elinkaari kuin nykyisillään käynnissä olevilla projekteilla.

4.5 Selvitys SapSim:in käytöstä osana tulevaisuudessa suoritettavia testejä

Insinöörityön tavoitteiden mukaisesti on tehty pienimuotoinen selvitystyö siitä, mikä olisi kannattava ratkaisu SapSim:in käytön tulevaisuudesta. Se sisältää muun muassa analyysin, kannattaako SAP-MDUS-integraation regressiotestit tehdä myös jatkossa SapSim:illä vai voitaisiinko ne jatkossa toteuttaa suoraan Autotesterillä. Käytännössä selvitystyö sisältää arvion, minkälainen työ olisi integroida myös SapSim:in keskeiset toiminnot Autotesteriin sen sijaan, että sitä ylläpidetään erillisenä työkaluna.

Erillisesti ylläpidettynä SapSim olisi heti valmis ratkaisu eikä se vaatisi ylimääräistä lisäkehitystä. Sen nykyiset ongelmat jatkaisivat olemassa olevina, jolloin testien käyttö ja ylläpito olisivat testaajille työläämpiä. Hyvät puolet SapSimin integroinnissa olisivat pitkällä aikavälillä merkittävämpiä ja siksi selvitystyön tulos suosittelee integraatiota.

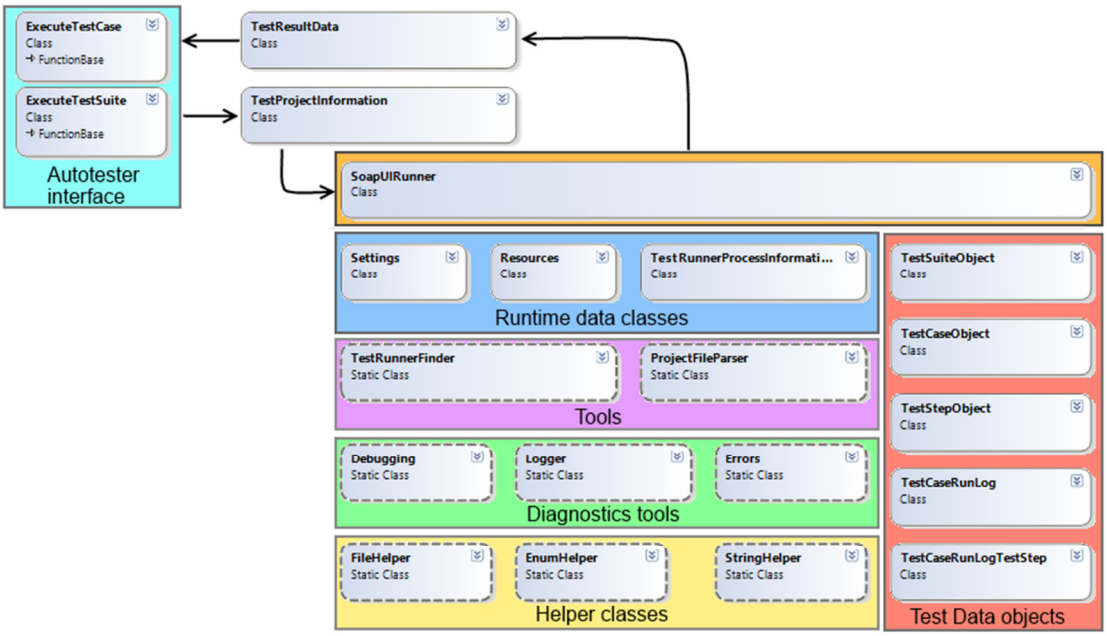
Selvitystyöstä laadittu raportti sisältää tietoa siitä, kuinka paljon kehitettyä ohjelmakoodia voitaisiin käyttää suoraan ja kuinka paljon sen osioita pitäisi muokata Autotesterille sopivaan formaattiin. Se sisältää myös arvion siitä, mitä toimintoja olisi kannattavaa tehdä kokonaan uudelleen, jotta ne soveltuisivat Autotesterin toimintamalliin. Lisäksi selvitystyön raportti sisältää aika-arvion kehitystyöhön kuluvasta ajasta listattuna integraation kannalta tarpeellisten sovelluskomponenttien osalta. Selvitystyön dokumentti on tehty vain Enoron sisäiseen käyttöön.

5 SoapUiRunnerin kehittäminen

SoapUiRunneriksi nimetty sovelluskomponentti suunnitteluineen ja toteutuksineen on tämän insinöörityön yhteydessä tehty käytännön toteutus. Sillä pyritään pääsemään johdanto-osiossa esitettyyn ensimmäiseen tavoitteeseen eli mahdollistaa www-sovelluspalveluiden testaamisen automatisointi ja liittäminen osaksi Autotesterillä ajettavia testejä. SoapUiRunner-sovelluskomponenttiin päädyttiin, kun alustavassa selvitystyössä osoittautui, että SoapUI:lla tehdyt testiprojektit ovat niitä testejä, joilla SAP-MDUS-integraation testausta hoidettiin. SoapUI:n testejä ajavalla komponentilla haluttiin ratkaista se keskeisin ongelma, joka projektin testien täydellisen automatisoinnin haasteena oli. Sen seurauksena alkoi komponentin suunnittelu ja käytännön toteutus. Komponentti päätettiin toteuttaa C#-ohjelmointikielellä, koska samaa ohjelmointikieltä on käytetty myös yrityksen muissa testausautomaation ratkaisuissa. Lisäksi SoapUiRunner on kohdennettu käyttämään Microsoftin .NET ajoympäristön versiota 3.5.

SoapUI:n projekteja ajavan komponentin olisi voinut toteuttaa hyvinkin yksinkertaisesti, mutta silloin lopputulos ei olisi täyttänyt lähellekään niitä laatuvaatimuksia, tähän insinöörityöhön liittyvälle toteutukselle oli suunnitteluvaiheessa haluttu asettaa. Tavoitteena oli tehdä vikasietoinen komponentti, joka pystyy diagnosoimaan mahdollisia vikatilanteita ja palvelemaan testaajia informatiivisilla raporteilla siitä, mitä testeissä todella tapahtui. Raportointiin haluttiin keskittyä etenkin silloin, kun testin suorittaminen epäonnistui. Komponentin tavoitteena on tarjota hyödyllinen, joustava ja käyttötarkoitusta palveleva toiminnallisuus, joka tuo käyttäjälleen lisäarvoa suhteessa SoapUI:n TestRunner-komentorivityökalun ajamiseen suoraan komentotulkin kautta.

SoapUiRunnerista syntyi kaiken suunnittelun, pohdinnan ja kehityksen jälkeen varsin monipuolinen työkalu. Kuvassa 7 näkyvät SoapUiRunnerin sisältämät luokat jaoteltuna niiden käyttötarkoituksen mukaisiin ryhmiin. Kokonaisuudessaan työkalu sisältää 21 eri käyttötarkoituksiin tehtyä luokkaa, jotka yhdessä muodostavat kokonaisuuden, jolla SoapUI-testien ajaminen Autotesteristä käsin on mahdollista. Kaikki monimutkaiset toiminnallisuudet on piilotettu sovelluskomponentin sisälle. Ulkoa päin katsottuna näkyvillä on selkeä käyttöliittymä, joka on täysin Autotesterin muun käyttöliittymän mukainen. Näin ollen SoapUiRunnerin käyttäminen on tehty Autotester-skriptejä luovalle testaajalle todella helpoksi.



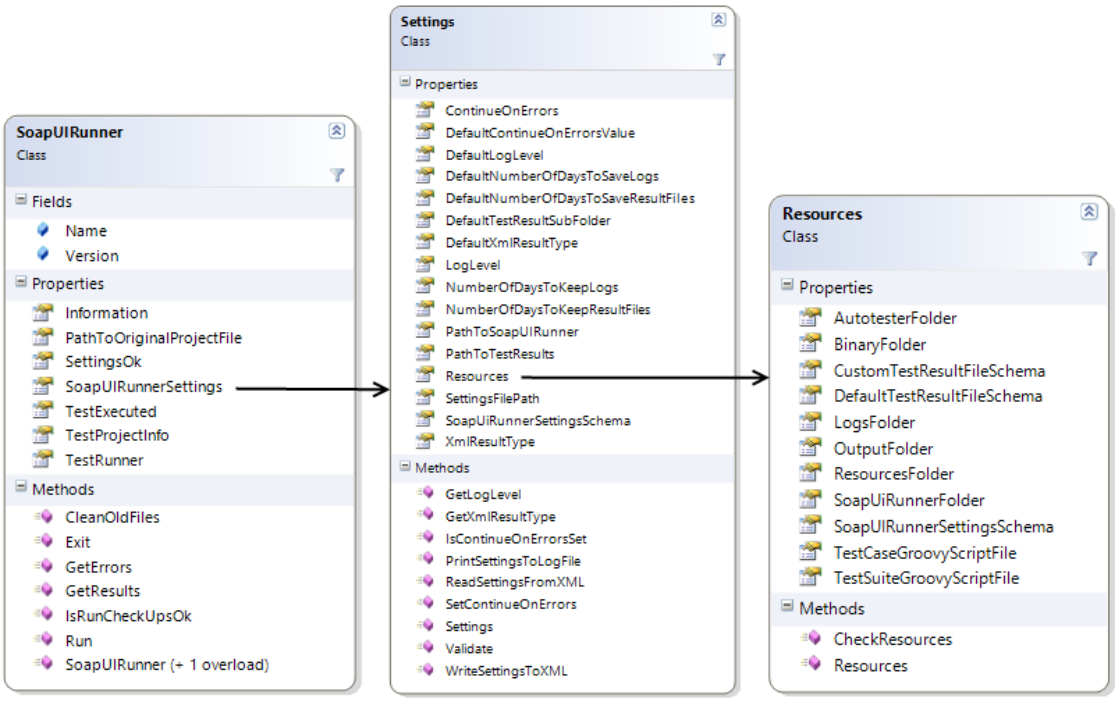
Kuva 7. SoapUiRunnerin luokkien keskinäinen suhde ja ryhmittely käyttötarkoituksen mukaiseen järjestykseen.

Hyvinkin yksinkertaisen käytön mahdollistavaan sovellukseen pystytään tekemään yksinkertaisuutta korostava ulkokuori, joka kuitenkin kätkee sisälleen hyvinkin monimutkaisia toiminnallisuuksia sekä paljon älykästä logiikkaa, joka parantaa käyttäjän käyttökokemusta huomattavasti. Ulkokuoren sisälle piilotettu toiminta ei näy ulkopuoliselle käyttäjälle mitenkään, lukuun ottamatta sen tuomaa käyttömukavuutta ja toimivaa lopputulosta, jonka sovellus pystyy tuottamaan lähes kaikissa tilanteissa. Tämä on se sovellusmalli, joita kohden nykytrendin mukaisessa sovelluskehityksessä näytettäisiin etenevän. Käyttäjällä ei ole halua tai kiinnostusta ryhtyä perehtymään sovelluksen toimintaan sen syvällisemmin. Sen sijaan sovelluksen halutaan vain yksinkertaisesti tekevän ne asiat, joita siltä odotetaan ilman, että se vaatisi monimutkaisten säätöjen tekemistä.

Kuvassa 7 on paitsi kaikki SoapUiRunnerissa käytetyt luokat, myös nuolin kuvatun etenemissilmukan siitä, miten sovellus toimii käytännössä. Autotesteristä käsin sovellusta ajetaan samalla tavalla, kuin mitä tahansa muutakin toimintoa, eli Autotesterin funktioiden kautta. Funktioiden mukana SoapUiRunnerille välitetään kaksi tietoa: mikä projekti halutaan suorittaa ja mikä testisarja tai -tapaus projektista suoritetaan. SoapUiRunner vastaanottaa tiedot siistinä pakettina, suorittaa saamansa informaation perusteella halutut testit, paketoi testitulokset niille tarkoitettuun pakettiin ja lähettää raportin testituloksista takaisin Autotesterin funktioille.

Pääasiallisen toiminnan lisäksi SoapUiRunnerilla on toimintaansa varten asetuksia ja resursseja. Lisäksi siihen kuuluu työkaluja, jotka mahdollistavat sen tietyt ominaisuudet sekä diagnostiikkaan liittyviä toiminnallisuuksia, joita hyödynnetään SoapUiRunnerin toiminnan seuraamiseen ja raportointiin. Kokonaisuuteen kuuluu mukaan myös jotain apuluokkia, jotka laajentavat kohdeajoympäristönä käytettävän .NET Framework 3.5:n toimintaa esimerkiksi sellaisten ominaisuuksien osalta, jotka on lisätty .NET-arkkitehtuuriin vasta version 4.0 myötä. Lisäksi SoapUiRunner-sovellukseen sisältyy useita säiliöitä testidatan säilömiseksi kahta vaihtoehtoista dataformaattia varten.

SoapUiRunnerin keskeinen toimintalogiikka sisältyy SoapUiRunner-luokkaan, joka on sovelluksen sisääntuloliittymä, sekä ajonaikaisia tietoja sisältäviin luokkiin. Kuvassa 8 näkyy edellä mainitut keskeisimmät luokat siltä osin, mitä ominaisuuksia niistä on merkitty julkisiksi. Sisäiset tai yksityiset metodit ja kentät ovat piilotettuina aivan, kuten ne ovat kyseistä luokkakirjastoa hyödyntävälle sovelluskehittäjällekin, joka haluaa käyttää näiden luokkien olioita. Kuvaan 8 on merkitty nuolin näiden luokkien riippuvuus toisistaan.



Kuva 8. SoapUiRunner-ohjelmakomponentin pääluokat ja niiden välinen suhde.

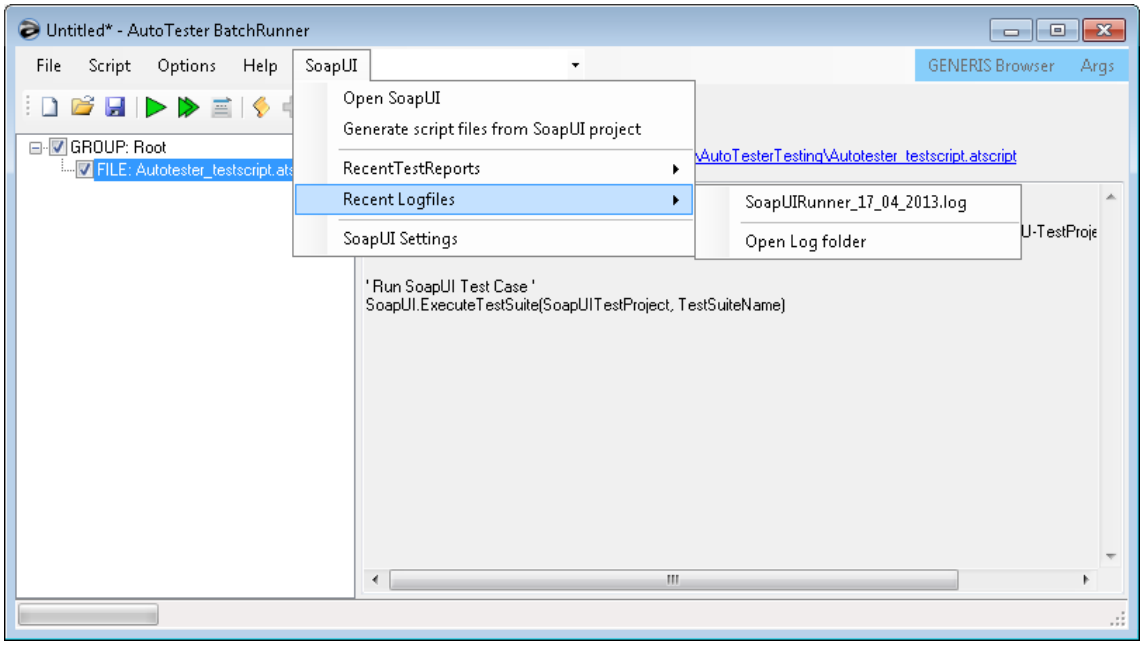
SoapUiRunner, joka on pääkomponentti SoapUI:n testitapausten ajossa, omaa Settings-luokan kentän, joka sisältää kaikki ajonaikaiset asetukset. Vastaavasti Settings-

luokan olio sisältää Resources-luokan olion, josta löytyy tiedot kaikista ulkoisista resursseista, kuten skriptitiedostoista ja xml-skeemoista, joita ohjelmakomponentti tarvitsee. Lisäksi resurssiolio sisältää tiedon ohjelman käyttämästä hakemistorakenteesta, joka on spesifioitu Autotesterin käyttämään formaattiin.

Seuraavissa luvuissa käsitellään SoapUiRunnerin toimintaa hieman pintaa syvemmältä. Ensin käsitellään SoapUiRunnerin yleinen toimintaprosessi, joka on jaettu selkeisiin vaiheisiin. Vaiheet suoritetaan tarkassa järjestyksessä, jotta tilaajasovelluksen käyttöön voidaan tuottaa testin tulokseen liittyvä data. Tämän jälkeen käsitellään vielä muita toiminnallisuuksia, jotka eivät ole tarpeellisia yksittäisten testitapausten ajossa, mutta jotka osaltaan varmistavat SoapUiRunnerin laadukkaamman ja vähemmän konfigurointia ja ylläpitoa vaativan toiminnan.

5.1 SoapUiRunnerin yleisimmän käyttötapauksen työnkulku vaiheittain

SoapUiRunnerin toiminnallisuuksia hyödynnetään Autotesterin käyttöliittymän kautta. Kuvassa 9 on nähtävillä Autotesterin käyttöliittymä. Siellä SoapUiRunnerille on oma valikko, josta löytyvät sen tarjoamat apuvälineet. SoapUI:n testien ajaminen tapahtuu suoritettavien Autotester-skriptien kautta.



Kuva 9. Autotesterin käyttöliittymä ja SoapUiRunnerin oma valikko.

Normaalissa tapauksessa, kun SoapUiRunner käynnistetään, sen asetukset ovat jo kunnossa aiempien käyttökertojen jäljiltä. Näin ollen SoapUiRunnerin toiminta etenee yleisen käyttötapauksen mukaisesti. Silloin SoapUiRunnerin toiminta etenee seuraavien vaiheiden mukaisessa järjestyksessä:

1) SoapUI-projektin käynnistäminen

SoapUI-projekti käynnistetään Autotesterin AT-Skriptistä, kutsumalla funktiota ”SoapUI.ExeuteTestSuite()” tai ”SoapUI.ExecuteTestCase()”. Funktioille annetaan kaksi parametria: Ajettavan projektin tiedostonimi polkuineen sekä funktiosta riippuen joko testitapauksen tai testisarjan nimi. Tämän jälkeen kutsuttu funktio hoitaa testidatan paketoimisen SoapUiRunnerille sopivaan muotoon.

Kun testidata on paketoitu tarkoitusta varten tehtyyn formaattiin, funktio luo uuden ilmentymän SoapUiRunner-olioluokasta, välittää sille testin tiedot ja käynnistää testin ajamisen. SoapUiRunner-tyyppisen olion luominen käynnistää automaattisesti myös diagnostiikan, joka tutkii, että olion asetukset ovat kaikin puolin kunnossa ja, että kaikki tarvittavat tiedot testin ajamiseksi ovat mukana tietopaketissa. Jos diagnostiikkaa ei suoriteta onnistuneesti läpi, saa suorittajafunktio tiedon virheestä jo ennen varinaisen testin käynnistämistä. Näin ollen testin suoritus voidaan hylätä jo hyvin aikaisessa vaiheessa. Samalla testaajalle palautetaan informaatiota siitä, mikä on mennyt suorituksessa vikaan ja kehotetaan tarkistamaan tarkempaa tietoa tapahtumista sovelluksen lokitiedostosta.

2) Testin ajaminen ja tulosten talteen ottaminen SoapUI:lta

Kun SoapUiRunner saa ajokäskyn Autotesterin funktiolta, se varmistaa vielä, että sen omat asetukset on kertaalleen validoitu. Näin ollen, mikäli kutsujafunktio yrittäisi jatkaa SoapUiRunnerin ajamista, vaikka aiemmassa asetusten validoinnissa on ilmennyt ongelmia, tehtäisiin siitä uusi virheilmoitus sen sijaan, että testiä yritettäisiin suorittaa. Sillä varmistetaan, että testi voidaan todellakin suorittaa ja että SoapUiRunneria käyttävä funktio on laadittu oikein.

Kun on saatu vahvistus siitä, että asetusten validointi on suoritettu onnistuneesti, SoapUiRunner valmistautuu ajamaan TestRunneria. Ensin TestRunnerin ajoa varten koostetaan ajoon liittyvät parametrit, jotka perustuvat sekä SoapUiRunnerin asetuksiin, että

sille syötettyihin testitapauksen tietoihin. Seuraavaksi TestRunner käynnistetään ja SoapUiRunner jää odottamaan sen valmistumista.

3) Vastaanotetun testi-informaation käsittely ja jalostaminen

TestRunnerin ajon päättyessä, se palauttaa lopetuskoodin sekä raportin suoritetuista toiminnoista. TestRunnerin lopetuskoodi kertoo, onko ajon aikana tapahtunut virheitä vai onko suoritus mennyt läpi ongelmitta. TestRunnerilta saatu lopetuskoodi kirjataan lokiin, jotta siitä jää myös tieto nähtäväksi tarvittaessa. Samalla TestRunner tulostaa informaatioraportin, jota ei kuitenkaan käytetä sellaisenaan mihinkään muuhun kuin saadun tiedon lokitiedostoon taltioimiseen. Varsinaiset testin tulokset luetaan ja tulkitaan sen sijaan TestRunnerin lopetuksen yhteydessä kirjoitetusta xml-muotoisesta tiedostosta, jonka formaatti riippuu tietystä SoapUiRunnerille määritellystä asetuksesta.

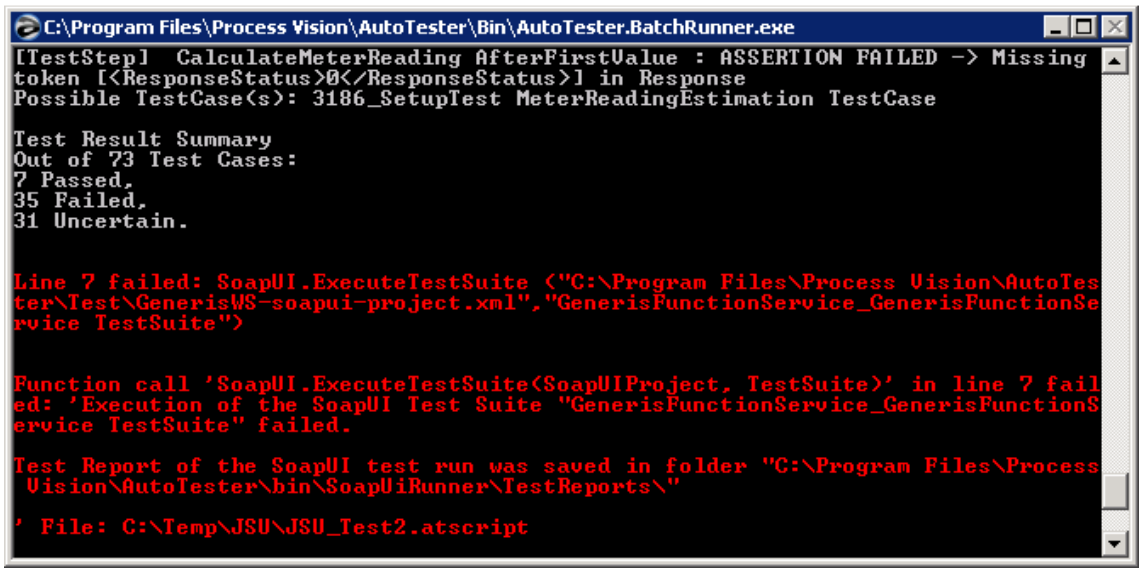
Kun TestRunnerin suoritus on päättynyt, etsii SoapUiRunner tuloshakemistosta määritettyjen kriteerien mukaan nimettyä tiedostoa. Kun tiedosto löytyy, varmistetaan siitä vielä luonti-ikä, jotta voidaan olla varmoja, että uusi tulostiedosto on luotu ajon yhteydessä. Jos esimerkiksi Windowsin tiukoista suojausasetuksista johtuen tiedostoa ei olekaan voitu luoda, pystytään vika jäljittämään tässä tarkistuksessa ja raportoimaan virheen todennäköinen alkuperä testaajalle.

4) Tulosten raportoiminen

Kun testin yhteydessä luotu tulostiedosto on läpäissyt sille tehtävät tarkistukset sekä xml-tiedoston skeeman mukaisen validoinnin, voidaan tulokset koota ja raportoida edelleen Autotesterille sen ymmärtämässä muodossa. Raportin jokainen testitapaus ja testiaskel tarkistetaan ensin onnistumisen osalta ja jos yksikin testiaskel on epäonnistunut, merkitään koko testitapaus epäonnistuneeksi. Tällöin epäonnistuneet testit ja testiaskeleet raportoidaan yhdessä niiden testistatusten kanssa.

Jos kaikki testit on läpäisty onnistuneesti, raportoidaan onnistunut testi, eikä silloin raportoida mitään ylimääräistä lisäinformaatiota. Onnistuneen tai epäonnistuneen testin sisältämä raportti palautetaan Autotesterille raportointiin suunnitellussa tietorakenteessa, josta Autotesterin funktiot avaavat tiedot ja kirjoittavat niistä xml-tiedoston. Vastaavalla tavalla etenee kaikkien testitapausten ajo ja vastausten raportointi riippumatta siitä, kumpaa Autotesterin funktiota käytetään.

Jos käyttäjä on kytkenyt näkyviin Autotesterin komentorivi-ikkunan, hän näkee siitä SoapUiRunnerin raportoimia asioita. Raportoiduista asioista yhteenveto on oleellisin tieto, ja se välitetään käyttäjälle yksittäisten tapausten tietojen jälkeen. Kuvassa 10 on nähtävillä osa tietovirrasta, jota SoapUiRunner välittää Autotesterille raportoitavaksi.



Kuva 10. Otos SoapUiRunnerin Autotesterillä lähettämästä informaatiosta.

5.2 Toimintaa tukemaan kehitettyjä toiminnallisuuksia

SoapUiRunnerin kehittämisessä on ollut aikaa keskittyä myös toimintaa tukevien ominaisuuksien luomiseen. Kehitystä on tehty myös sen osalta että, sovellus huomioisi paremmin poikkeuksellisetkin tilanteet ja toimisi luotettavasti myös niissä. Lisäksi tässä yhteydessä on kehitetty säätöjä, joilla voidaan käyttää harvinaisemmin tarvittuja, monipuolisempia toimintoja.

Ylimääräistä kehitystä on voitu tehdä sen osalta, että sovelluskomponentti toimisi mahdollisimman automaattisesti, eikä se vaatisi välttämättä alkusäätöjä ollakseen käyttövalmis. Yleensä harvemmin käytettävien ominaisuuksien kehittämiseen, joita käytetään esimerkiksi vain ensimmäisellä ajokerralla, ei haluta käyttää kovinkaan paljon aikaa. Harvemmin käytettävien ominaisuuksien hyöty kustannuksiin nähden ei ole paras mahdollinen.

Monipuolisemmat säädöt ja säätämisen tarpeettomuus eivät synny aivan itsestään. Siksi näiden toiminnallisuuksien kehittämiseen on kulunut huomattavasti enemmän aikaa, kuin normaalissa ohjelmistokehitysprojektissa olisi haluttu tähän käyttää. Seuraavissa luvuissa käsitellään tärkeimmät toiminnallisuudet, jotka eivät olisi olleet välttämättömiä komponentin ydintoiminnallisuuden kannalta mutta jotka parantavat sen toimintaa ja osaltaan varmistavat testien onnistumisen.

5.2.1 Säädettävät asetukset

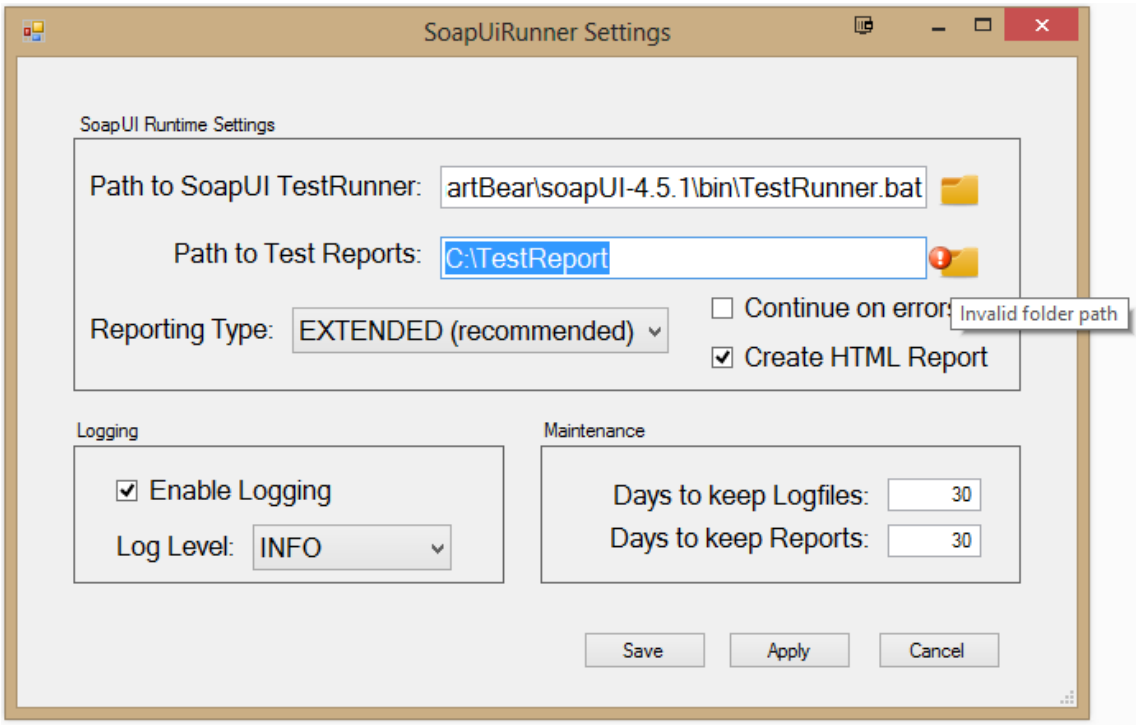
SoapUiRunner-komponenttiin kuuluu asetustiedosto, josta sovellus lukee erilaisia asetuksia. Asetustiedosto noudattaa omaa xml-formaattia, joka perustuu sille laadittuun xml-skeemaan. Jos asetustiedostoa ei löydy, luodaan uusi asetustiedosto, joka sisältää sovellukselle määritetyt oletusasetukset.

Asetustiedoston rakenne verifioidaan kehitettyä xml-skeemaa vasten ja asetusten arvot verifioidaan ohjelman lukiessa asetuksia. Asetuksia muuttamalla voidaan muuttaa SoapUiRunnerin toimintaa. Jos asetustiedostossa on virheitä, ne pyritään korjaamaan ajon aikana ja tallentamaan korjattuina asetustiedostoon. Yleensä asetuksen virheellinen arvo muutetaan sen oletusarvoon. Onnistunut korjausyritys tiedotetaan lokitiedostossa ja ohjelman toiminta jatkuu normaalisti. Sen sijaan ohjelman käytön kannalta oleellinen virhe ilmoitetaan käyttäjälle heti ja sen lisäksi ohjelman suoritus pysäytetään.

Asetuksista löytyy tieto siitä, missä SoapUI:n komentorivityökalu, TestRunner, sijaitsee sekä hakemistopolku, minne testien tulostiedostot tallennetaan. Lisäksi sieltä löytyvät asetukset sille, kirjoitetaanko lokitiedostoa, ja jos kirjoitetaan, millä tarkkuudella merkintöjä tehdään. Asetuksilla voidaan myös määritellä käytettävän tulostiedoston xml-formaatti, miten toimitaan virheen sattuessa, luodaanko tuloksesta html-muotoinen raportti sekä kuinka monta vuorokautta loki- ja tulostiedostoja säilytetään. Nämä mahdollistavat ohjelmiston joustavan käytön ja informaation hallinnan sekä ylläpidon valintojen tekemisen. Mahdollisista asetuksien arvoista kerrotaan ohjelman mukana toimitettavassa readme.txt-dokumentissa. Esimerkki asetustiedostosta ja siihen liittyvästä skeemasta on nähtävillä liitteessä 1.

SoapUiRunnerin asetuksia voidaan säätää Autotesterin graafisesta käyttöliittymästä. Autotesterin valikkoon on lisätty ”SoapUiRunner settings” -valinta. SoapUiRunnerin asetuksien avaaminen avaa kuvan 11 mukaisen ikkunan, josta asetuksia pääsee muut-

tamaan. Ikkuna on toteutettu Windows Forms -kehitystyökaluilla. Asetusikkunan näyttämät asetukset luetaan ja tallennetaan käyttäen sovelluksen omaa asetustiedostoa. Asetusikkuna validoi kaikki sille annettavat arvot, eikä mihinkään kenttään pysty tallentamaan vääränlaista tietoa. Numerokentät estävät muiden kuin numeroiden syöttämisen ja muuttavat tyhjät kentät nolla-arvoiksi. Polkukentät tarkistavat sisällön oikeellisuuden sekä kohteiden olemassaolon levyjärjestelmässä. Virheelliset arvot estävät tallennuksen ja virheen alkuperä ilmaistaan kentän punaisella virhemerkillä sekä sen kohdalle hiirellä osoittaessa myös virhetekstillä.



Kuva 11. SoapUiRunnerin asetusikkuna, jossa on virheellinen polkumääritelmä.

5.2.2 Extended-formaatti tuloksille ja työkalu sen käyttöönotolle

SoapUI:n avoimen lähdekoodin versio on kohtalaisen rajoittunut verrattuna ohjelman maksulliseen pro-versioon. Sillä ajettujen testien tulosformaatti on epäkäytännöllinen siinä tapauksessa, että testisarja (test suite) sisältää useita testitapauksia (test case). Lisäksi sen käyttö lisää SoapUiRunnerin toimintariskiä. XML-formaatti voi esimerkiksi muuttua SoapUI:n kehittyessä, jolloin SoapUiRunner lakkaisi todennäköisesti toimimasta. Hyvä puoli asiassa on se, SoapUI tarjoaa mahdollisuuden tehdä omia ”tear-down-skriptejä”, jotka ajetaan automaattisesti testitapauksen tai testisarjan päätteeksi riippuen siitä, mihin kohtaan projektissa skripti on asetettu. Skriptit ovat ”Groovy Script”

-kielellä kirjoitettavia ohjelmapätkiä, jolla voidaan suhteellisen helposti luoda lisää toimintalogiikkaa testien ajoon (Groovy – a dynamic language for the Java platform 2013).

Suoritetuista testeistä olisi hyvä saada informaatiota jokaisesta testitapauksesta, niin keston kuin myös onnistumisen osalta. Sen vuoksi oli järkevää kehittää tarkoitukseen sopiva xml-formaatti, johon tiedot kerättäisiin kaikista suoritetuista testeistä erillisen skriptin avulla. Tarkoitukseen sopivan skriptin kehittäminen onnistuu Groovy Scriptillä, jota SoapUI tukee.

SoapUI tarjoaa testien päätteeksi ajettavalle teardown-skriptille suhteellisen laajasti yksityiskohtaisia tietoja ajetuista testitapauksista. Siten tiedon kerääminen ja jäsentely omaan formaattiin onnistuu kohtuullisen helposti. Suurempi ongelma oman formaatin käyttöönoton takana on se, että jos kaikkiin testiprojekteihin jouduttaisiin aina erikseen liittämään sopiva tulostusskripti riippuen suoritetaanko yksi testitapaus tai kokonainen testisarja, nakertaisi se osin automatisoinnin mukanaan tuomaa hyötyä tai vähintäänkin nostaisi vaadittavan työn määrää, kun testejä haluttaisiin ottaa osaksi Autotesterin automatisoituja testejä.

SoapUI:n käyttämät projektitiedostot ovat itsessäänkin xml-muotoisia tiedostoja. Siksi on mahdollista toteuttaa kohtuullisella työmäärällä sellainen ratkaisu, joka automatisoi sopivan skriptin lisäämisen osaksi testiprojektia, ennen kuin se suoritettaisiin. Kehitetyssä toiminnallisuudessa oikean tulostusskriptin lisäys perustuu siihen, halutaanko SoapUiRunneria käyttää yhden testitapauksen vai kokonaisen testisarjan ajoon. Se, mihin kohtaan testiprojektissa skripti tulee lisätä, riippuu siitä, ajetaanko testiprojektin kaikki testitapaukset yhdellä kertaa vai ajetaanko sieltä vain tietty yksittäinen testitapaus. Siksi projektitiedoston käsittelyä varten ollut järkevää toteuttaa oma, kyseisen logiikan sisältävä luokka, ”ProjectFileParser”, joka hoitaa SoapUI:n projektitiedostojen muokkauksen ja niiden sisältämän tiedon hakemisen. Kyseinen työkalu on vain yksi osa SoapUiRunner-komponentin tekemään automaatiota, joka mahdollistaa extended-formaatin käytön.

Kokonaisuutena extended-formaatin käytön mahdollistava automaatio toimii seuraavasti. Ajon aikana SoapUiRunner-komponentti:

– tekee alkuperäisestä testiprojektitiedostosta väliaikaisen kopion
– uudelleennimeää kopioidun testiprojektin uniikilla nimellä, jotta suoritettavia tiedostoja voisi olla tarvittaessa rinnakkain useita
– lisää resurssitiedostosta sopivan tulostusskriptin oikeaan paikkaan väliaikaistiedoston testiprojektissa
– suorittaa testin käyttäen väliaikaistiedostoa, johon on lisätty teardown-skripti ja
– testin suorittamisen jälkeen poistaa luodun väliaikaistiedoston.

Näin ollen testaajille ei koidu mitään lisätyötä siitä, että extended-formaattia käytetään. Extended-formaatin tuoma lisähyöty näkyy erityisesti silloin, kun testi epäonnistuu. Tuolloin testistä pystytään kertomaan ne testitapaukset ja -askeleet, jotka ovat epäonnistuneet. Yhtä kattavaa informaatiota ei ole saatavilla SoapUI:n tarjoamassa oletusformaatissa. Lisäksi extended-formaatti mahdollistaa tiedon jalostamisen, jos tulevaisuudessa halutaan esimerkiksi seurata testeihin ja niiden osiin kuluvaa aikaa tai kerätä muuta statistiikkaa. Liitteestä 2 löytyy esimerkki extended-formaatin mukaisesta tiedostosta.

Extended-formaatin ohessa sen luomaa xml:ää varten kehitettiin xml-transformaatiotiedosto, jonka avulla xml:stä voidaan muodostaa lennossa selkeä html-muotoinen testiraportti. HTML-testiraportti on selkeä apu testitapausten ja -askelien onnistumisen tarkasteluun. Extended-formaattia käytettäessä SoapUiRunner koostaa extended-formaatin xml-tiedostosta lennossa html-muotoisen raportin. Esimerkkiraportti on liitteessä 5.

5.2.3 Tiedostojen sisällön validointi

Aiemmat kokemukset ovat osoittaneet, etteivät kaikki sovellukset ole aina luotettavia siltä osin, että tiedostoihin kirjoittaminen tapahtuisi sataprosenttisella varmuudella loppuun asti. Virheitä sattuu erityisesti silloin, kun järjestelmä on kovassa kuormituksessa. Lisäksi, kun ohjelmakomponentin toiminta on riippuvainen kolmannen osapuolen ohjelmistosta, jonka toiminta saattaa muuttua myöhempien versioiden myötä, on hyvä pyrkiä jotenkin validoimaan saatua tietoa eli varmistamaan sen oikeellisuus. Tiedon oikeellisuuteen liittyvät ongelmat tulisi huomioida jotenkin ohjelman toiminnassa ja niihin tulisi varautua tavalla tai toisella.

XML-muotoisen tiedon validoimiseksi yksi suhteellisen vähätöinen ratkaisu on tehdä xml-formaatille skeema, jota xml-tiedoston rakenteen tulee noudattaa. SoapUiRunnerissa on käytössä kolmentyyppisiä xml-tiedostoja, joille kaikille on omat skeemat. Skeemoissa määritellään, mitä tietoja tiedostossa täytyy esiintyä ja siellä lisäksi yksilöidään, mitä tietoa saa esiintyä tiedostossa korkeintaan yhden kerran, jotta tiedosto olisi validi. Aina kun SoapUiRunner lukee xml-tiedoston sisällön, se tarkistaa, että tiedosto noudattaa xml-skeemassa asetettuja normeja. Jos virheitä esiintyy, niistä annetaan ilmoitus testin raportoinnin yhteydessä ja tarkempi tieto viasta kirjoitetaan lokitiedostoon.

Ohjelman xml-muotoisessa tiedostossa olevat asetukset ovat erittäin oleellisia ja niiden täytyy olla kunnossa, jotta ohjelma toimisi vakaasti. Siksi niiden osalta tiedot tarkistetaan xml:n validoinnin lisäksi vielä asetettujen arvojen osalta, kun asetuksia ollaan ottamassa käyttöön. Virheelliset asetukset korvataan oletusasetuksilla ja ne myös tallennetaan asetustiedostoon, jotta seuraavalla suorituskerralla samat virheet eivät toistuisi.

5.2.4 Tapahtumien kirjaus lokiin

Ohjelman edistymisen ja toiminnan seuraamisen kannalta lokien pitäminen on järkevää. Lokiin voidaan kirjata tarkkoja tietoja tapahtumista esimerkiksi silloin, kun ohjelman käytössä tapahtuu jotain poikkeavaa. Lisäksi lokista voi ajon aikana seurata ohjelman etenemistä esimerkiksi niissä tilanteissa, kun ohjelman suoritus kestää pitkään ja ulkoisesti saattaisi näyttää siltä, ettei mitään tapahdu. Loki on myös hyvä paikka säilöä muistiin, mitä testejä on ajettu ja millä asetuksilla varustettuna.

SoapUiRunneriin on myös toteutettu lokitiedostojen kirjoitus, koska siitä on hyötyä ohjelman ylläpidon kannalta. Lokien kirjoitus voidaan kytkeä asetuksista päälle tai pois. Oletuksena lokeja kirjoitetaan. Jotta lokitiedoston koko ei kasvaisi mahdottomaksi ajan saatossa, joka päivä luodaan uusi lokitiedosto kyseisellä päivämäärällä leimattuna. Päivän lokitiedostoon talletetaan tiedot kyseisen päivän suorituksista.

Jokaiselle lokimerkinnälle on asetettu tapahtuman vakavuusarvo (log level), jonka perusteella lokimerkintöjä kirjoitetaan. Kirjattavien lokien vakavuusarvo määritellään asetukista, jonka mukaiset tai sitä vakavammat tapahtumat lokitiedostoon kirjataan. Esimerkiksi jos asetuksissa on asetettu loglevel:iksi vakavuusaste ”Error”, vain virheet ja sitä suuremman vakavuuden lokitiedot tallennetaan lokiin.

Lokien erilaisia vakavuusasteita suurimmasta pienimpään ovat ”system”, ”error”, ”warning”, ”info” ja ”debug”. Ensimmäinen näistä kattaa vain ohjelman käynnistymiseen ja lopettamiseen liittyvät tiedot ja viimeisin kattaa hyvinkin tarkasti ohjelman suoritukseen liittyvää informaatiota, jolla on helppoa paikallistaa mahdollisia ohjelmavirheitä. Esimerkki lokitiedostosta on nähtävissä liitteessä 3.

5.2.5 TestRunnerin etsintyökalu

SoapUI:n komentorivityökalu, joka ajaa SoapUI:n projektien mukaisia testejä SoapUI:lla annettujen parametrien perusteella, on nimeltään TestRunner. SoapUI:n ja TestRunnerin olemassaolo ovat välttämättömiä SoapUiRunner-komponentin toiminnan kannalta. Siksi TestRunnerin sijainti tallennetaan SoapUiRunnerin asetustiedostoon. Alkuperäinen ajatus oli, että TestRunnerin polku tulisi tallentaa manuaalisesti asetustiedostoon, kun SoapUiRunner otetaan ensimmäisen kerran käyttöön. Ajatus, että jotain pitäisi manuaalisesti säätää paikoilleen, tuntui ristiriitaiselta tämän projektin taustalla olevaa ideaa ajatellen. Ideahan oli tiivistettynä jonkin manuaalisen prosessin helpottamista automatisoinnin avulla. Siksi on luontaista ajatella, että myöskään TestRunnerin sijaintia ei tulisi joutua asettamaan manuaalisesti paikoilleen vaan se pyrittäisiin etsimään sovelluksen toimesta automaattisesti.

TestRunneria etsivän työkalun suunnittelu ei ole ollut kuitenkaan aivan mutkatonta. Alun perin ajatus siitä, että TestRunnerFinder-työkalu kävisi läpi koko levyjärjestelmän, kuulosti erittäin hitaalta operaatiolta. Siksi työkaluun on luotu parametri, jolla valitaan vaihtoehtojen: nopean, normaalin tai kokonaisvaltaisen haun käytöstä.

Nopea haku etsii yleisimmät sijainnit, kuten Windowsin ohjelmatiedostot, kaikilta asemilta ja etsii nimenomaan SoapUI:n kehittäjän, SmartBearin nimellä varustettuja hakemistoja, joihin SoapUI oletuksena asennettaisiin. Normaali haku laajentaa hakua muihinkin ohjelmatiedostoihin sekä Autotesterin juurihakemistoon ja käy läpi kaikkien isäntähakemistojen alta löytyvät polut, kunnes se on edennyt levyn juureen asti. Täyshaku puolestaan etsii SoapUI:ta kaikista hakemistoista kaikilta levyjärjestelmiltä lukuun ottamatta niitä hakemistoja, joiden käyttö on estetty järjestelmän toimesta. Jokaisella hakuvaihtoehdolla on priorisoitu haettavaksi kevyemmän hakuvaihtoehdon hakukohteet ensin, jolloin täyshaku löytää kohteen yhtä nopeasti kuin nopea hakukin, mikäli nopea haku yleensäkin löytäisi kohteen.

TestRunnerFinderin toimintaa testattiin vaihtamalla SoapUI:n sijaintia levyillä eri paikkoihin ja seuraten testiohjelmassa aikaa siitä, kuinka kauan kukin hakutapa kesti. Testien perusteella nopea haku kesti aina alle sekunnin, normaali haku yleensä alle kymmenen sekuntia ja täyshakukin maksimissaan 70 sekuntia. Toki hakuaika pitenisi, jos SoapUI:n pyrkisi piilottamaan vielä syvemmälle tiedostorakenteisiin tai jos levyjä olisi huomattavasti enemmän kuin testijärjestelmässä, jossa oli kolme eri levyä. Lisäksi testit osoittivat, ettei TestRunneria löytynyt silloin, kun se piilotettiin järjestelmän suojattuihin hakemistoihin, joiden selaukseen ei ole normaalisovelluksilla oikeutta. Tällöin ohjelma huomauttaa, että TestRunnerin sijainti pitää lisätä manuaalisesti.

Testit osoittivat sen, että haku voidaan huoletta säätää pysyvästi täyshakua hyödyntäväksi. Haku suoritettaisiin vain kerran silloin, kun Autotester asennetaan uuteen järjestelmään, jonka jälkeen sijainti on tallessa asetuksissa. Tässä tapauksessa muutamankin minuutin haku ei aiheuta liian suurta haittaa toiminnalle suhteessa hyötyyn. Useimmiten haun ei kuitenkaan pitäisi kestää yli kymmentä sekuntia, ellei SoapUI:ta ole sijoitettu poikkeukselliseen sijaintiin. Hakutyypin parametri mahdollistaa kuitenkin tarvittaessa sen, että haku voidaan helposti säätää nopeammalle, mikäli täyshaun kanssa havaitaan ilmenevän jotain ongelmia.

5.2.6 Tiedostojen aikaleimaus ja vanhojen tiedostojen siivous

SoapUiRunnerin suunnittelussa on pyritty huomioimaan sen käyttöä myös pitkällä tähtäimellä. Nykyisin on hyvin yleistä kehittää ohjelmia ajettavaksi rinnastetusti useilla suoritinytimillä, joka vaatii joidenkin asioiden huomioimista ohjelmistokehityksessä ja ohjelman toiminnan suunnittelussa. Tähän liittyen SoapUiRunnerin komponentit on tehty säieturvallisiksi, jonka lisäksi tulostiedostot aikaleimataan sekunnin tarkkuudella, jotta niiden tiedostonimet olisivat mahdollisimman yksilöllisiä.

Myös SoapUiRunnerin luomissa väliaikaistiedostojen nimissä on huomioitu yksilöllisyysvaatimus. Ajettavasta projektista tehdylle väliaikaistiedostolle annetaan tietokoneen laskemaan aikaan perustuva numerosarja osaksi nimeä, jolloin rinnakkaisesti ajettavat prosessit eivät voi vahingossa korvata samaa tiedostoa. Näin ollen jokainen suoritettava säie omaisi yksilöllisen kontekstitunnuksen.

Kun SoapUiRunneria käytetään pitkään ja ahkerasti, alkaa ajan myötä lokihakemistoon ja tuloshakemistoon kertyä huomattavia määriä vanhoja tiedostoja. Siksi SoapUiRun-

nerin asetuksissa on asetus molemmille tiedostotyypeille, jolla voidaan määritellä, kuinka monen päivän jälkeen kyseinen tiedosto poistetaan SoapUiRunnerin ajon yhteydessä. Näin ollen voidaan esimerkiksi määrittää, että kaikki kuukautta vanhemmat tulostiedostot poistetaan automaattisesti, jolloin levytila ei täyty pitkällä aikavälillä. Sekin on mahdollista, ettei tiedostoja säilytetä ollenkaan, jolloin ne poistetaan, kun sovelluksen suoritus päättyy.

5.2.7 Työkalu automaattisten Autotester-skriptien luomiseksi

Kun SoapUiRunner julkaistaan virallisesti Autotesterin versiopäivityksen yhteydessä, on ensimmäinen asia SoapUI:n testien automatisoimiseksi se, että niille täytyy tehdä Autotester-skriptit. Autotester-skriptejä täytyy tehdä yksi per testisarja tai joissain tapauksissa ehkä myös yksi per testitapaus, mikäli jokainen testitapaus halutaan erotella omaan skriptiinsä. Testejä on olemassa paljon ja niihin liittyvien skriptien kirjoittaminen on liukuhihnamaista työtä, koska skripteissä vaihtelevat lähinnä testiprojektien ja -tapausten nimet.

Jotta SoapUI:n testien automatisoiminen olisi entistä helpompaa ja nopeampaa, on myös testiskriptien laatiminen hyvä pyrkiä automatisoimaan. Testaajan aika on arvokasta, ja ohjelmallisesti toteutettuna työtä säästyy huomattavasti, kun testiskriptit luodaan automatisointia hyödyntäen. Testiskriptien luontityökalun olisi voinut tehdä myöhemmässäkin vaiheessa, mutta siitä saadaan paras hyöty irti, jos se julkaistaan yhdessä SoapUI-toiminnallisuuden yhteydessä, koska silloin tarve skriptien luomiselle on erityisen suuri.

Testiskriptien luomistoiminnallisuus löytyy Autotesteriin tehdystä SoapUI-valikosta. Se avaa graafisen ikkunan, jossa on viisiportainen työnkulku testiskriptien luomiseksi. Työnkulussa valitaan ensin SoapUI-projektitiedosto ja hakemisto, jonne testiskriptit luodaan. Seuraavassa vaiheessa aukeavat valintalistat, joista voidaan valita joko kokonaisia testisarjoja tai yksittäisiä testitapauksia. Testisarjoja valittaessa voidaan lisäksi valita, halutaanko jokaiselle testitapaukselle tehdä oma skriptitiedosto sen sijaan, että testisarjalle tehtäisiin yksi yhteinen tiedosto. Testien valitsemisen jälkeen voidaan vielä lisätä omia skriptikoodirivejä sekä ennen että jälkeen testin suorittamisen. Tämän jälkeen sovellus generoi esimerkkiluonnoksen yhdestä tehtävästä testiskriptistä sekä yhteenvedon, jossa on listattuna kaikki luotavat testitapaukset. Työnkulku päättyy painik-

keeseen, jossa luodaan testiskriptitiedostot ja suljetaan ikkuna. Liittestä 6 on nähtävillä ”GenerateTestScripts” -työnkulun käyttöliittymä kuvina.

5.3 Kehityksen aikaiset haasteet ja ennalta huomioidut seikat

SoapUiRunnerin kehityksen aikana on tullut vastaan monia haasteita ja ongelmia. Osaan niistä on löytynyt täysin toimiva ratkaisu ja osan kanssa on jouduttu tekemään kompromisseja. Jälkimmäisissä tapauksissa on kuitenkin pyritty minimoimaan riskiä siitä, että ohjelma lakkaisi toimimasta kokonaan tai ettei se toimisi oikein sen kaikissa mahdollisissa käyttötarkoituksissa. Yleensäkin tulevilta ongelmilta on pyritty välttymään ennalta ehkäisemällä mahdollisia ongelmakohtia jo sovelluksen toiminnan suunnitteluvaiheessa.

SoapUiRunnerin suunnittelussa ja toteutuksessa on pyritty myös huomioimaan asioita, joiden huomioiminen ei välttämättä ole edes tarpeellisia silloin, kun prosessia ajetaan kerrallaan vain yhdestä Autotester-skriptistä käsin. Nämä asiat liittyvät esimerkiksi sovelluksen rinnakkaiseen ajamiseen usealla suoritinytimellä. Tässä yhteydessä puhutaan sovelluksen komponenttien osalta käsitteistä vapaakäyntinen (reentrancy) ja säieturvallisuus (thread safety).

SoapUiRunnerin kehitystä ei suunniteltu kaikilta osin ennen varsinaisen käytännön toteutuksen alkua. Tämä johtui osittain siitä, ettei alkuperäinen selvitystyö ollut riittävän kattavaa, jotta siitä olisi ollut täysin selkeä kuva, mitä lopullisen tuotoksen tulisi sisältää. Lisäksi insinöörityön sisältö on muovautunut jonkin verran prosessin aikana ja siksi lopullinen kokonaisuus ei ollut vielä tiedossa siinä vaiheessa, kun sovellus oli suunnitteilla. Kolmantena syynä sille, että suunnittelutyötä on jouduttu tekemään osittain kehityksen rinnalla, on ollut se, että osa toiminnallisuuksista on kehitetty sen perusteella, että aikataulu on sallinut niiden kehityksen. Tärkeimmäksi arvioitujen ominaisuuksien kehitys on näin ollen ollut etusijalla.

5.3.1 Riippuvuus kolmannen osapuolen ohjelmistosta

SoapUiRunner nimensä mukaisesti ajaa SoapUI:n projekteja. Se ei kuitenkaan aja niitä itsenäisesti, vaan se ohjeistaa SoapUI:n komentorivityökalun, TestRunnerin ajamaan

tietyn testin ja tuottamaan siitä tietynlaista informaatiota. Näin ollen SoapUiRunnerin toiminta on hyvin riippuvaista siitä, miten SoapUI ja TestRunner toimivat.

SoapUiRunnerin riippuvuus TestRunnerista johtaa muutamaan seikkaan, jotka on täytynyt huomioida SoapUiRunnerin suunnittelussa. Ensiksikin SoapUiRunner pystyy ohjaamaan testien ajoa vain niiltä osin, mitä TestRunnerin käyttöliittymä mahdollistaa. Näin ollen kaikkia mahdollisia ominaisuuksia, joita haluttaisiin toteuttaa, ei välttämättä ole mahdollista toteuttaa – ei ainakaan ilman, että TestRunnerista tehtäisiin kokonaan oma versio. Se taas lisäisi tarvittavan työn määrää merkittävästi.

SoapUiRunnerin toteutuksen yhteydessä kävi myös ilmi ohjelmavirhe, joka TestRunnerin komentoriviargumenttien käsittelyssä on. Kyseinen virhe täytyi ensin todentaa, ja siihen oli luotava ratkaisu, jotta sovellukset saatiin toimimaan yhteen. Nykyisellään SoapUiRunnerin toiminta on täysin riippuvainen TestRunnerin toiminnasta. Se on tietoinen riski, joka on otettu silloin, kun sovellus on päätetty kehittää hyödyntämään SoapUI:ta. Hyvä puoli SoapUI:n ja TestRunnerin käytössä on se, että suuri osa toiminnallisuudesta on tehty valmiiksi, eikä kaikkea ole tarvinnut kehittää tyhjästä, jotta www-sovelluspalveluja testaavien testien automaattinen ajo olisi mahdollista.

TestRunnerin olemassa oleminen ja sen sijainnin paikallistaminen on välttämätöntä SoapUiRunnerin toiminnalle. Sen vuoksi SoapUiRunner on täytynyt toteuttaa siten, että TestRunnerin sijainti, joka ei välttämättä ole aina sama jokaisessa järjestelmässä, voidaan välittää SoapUiRunnerin tietoon. Näin ollen SoapUiRunneriin on täytynyt kehittää mahdollisuus tallentaa tieto TestRunnerin sijainnista. Toisaalta, TestRunnerin sijainti tuskin muuttuu samassa järjestelmässä kovinkaan usein. Siksi ei olisi mitään järkeä, että TestRunnerin sijainti jouduttaisiin välittämään SoapUiRunnerille jokaisella käynnistyskerralla. Näin ollen järkevin toteutus on ollut se, että TestRunnerin sijainti on tallennettu SoapUiRunnerin asetustiedostoon ja siten on pystytty vähentämään TestRunnerin sijaintiin liittyviä toimintariskejä.

5.3.2 Rinnakkainen käyttö

Rinnakkaisella käytöllä tarkoitetaan tässä yhteydessä sitä, että samoja ohjelma- eli binääritiedostoja pystytään ajamaan samanaikaisesti yhden tai useiden ohjelmien toimesta. Myös SoapUiRunnerin suunnittelussa on pyritty huomioimaan mahdollisuus ajaa sovellusta rinnakkaisina ajoina. Suunnittelussa on erityisesti huomioitu kaikkein

kriittisimpien komponenttien osalta säieturvallisuus ja vapaakäyntisyys. Säieturvallisuus tarkoittaa metodien osalta sitä, että niitä voidaan ajaa eri säikeissä yhtä aikaa ilman, että on pelkoa esimerkiksi siitä, että yhteisessä käytössä olevat resurssit sekoaisivat tai muuttuisivat kriittisellä hetkellä.

Vapaakäyntisyys on ohjelmoinnissa käytettävä termi sille, että jotain funktiota tai metodia voidaan ajaa samanaikaisesti useampaan kertaan, jolloin vaatimukset ovat jossain määrin samat, kuin säieturvallisuudessakin. Lisäksi vapaakäyntistä metodia tarvitaan siinä tilanteessa, kun metodia halutaan suorittaa rekursiivisesti. Rekursiivisella tarkoitetaan tässä yhteydessä ohjelmasilmukka, jossa sama metodi kutsuu itseään, jolloin samaa metodia suoritetaan sisäkkäin useammalla instanssilla. SoapUiRunnerissa tiettyjä hakemistopolkuja tutkivia funktioita kutsutaan rekursiivisesti, koska jokaisessa alihakemistossa tehtävä toimenpide on täysin sama, kuin alihakemiston isäntähakemistossakin. Näin ollen rekursiivinen suorittaminen on ollut kaikkien järkevin tapa toteuttaa kyseinen toiminnallisuus ja metodin on täytynyt toteuttaa vapaakäyntisyyden vaatimukset.

Yleensä rinnakkain suoritettavissa sovelluksissa ongelmia tuottavat staattiset luokat. Staattiset luokat ovat sellaisia luokkia, joista luodaan ohjelman käynnistyessä vain yksi ilmentymä ja se on yhteinen kaikille sitä käyttäville komponenteille. Staattisten luokkien staattiset metodit eivät ole niinkään ongelmallisia mutta sen sijaan staattiset kentät tuottavat vaikeuksia, koska niihin tallennettuja arvoja voidaan parhaimmillaan muuttaa useasta sijainnista yhtä aikaa.

SoapUiRunnerissa on käytetty staattisia olioluokkia silloin, kun se on ollut järkevin tapa toteuttaa jokin toiminnallisuus. Staattisten luokkien yhteiset kentät on kuitenkin merkitty C#:n tarjoamalla ”[Thread Static]”-tunnuksella, joka tarkoittaa sitä, että resurssi on yhteinen vain samassa säikeessä. Näin ollen näitä komponentteja voidaan käyttää huoletta useammasta säikeestä käsin ilman ongelmia. Nykyisen arvion mukaan ainoa tunnettu rinnakkaiselle suoritukselle pullonkaulaksi muodostuva resurssi voi olla lokitiedosto, johon kaikista säikeistä yritettäisiin kirjoittaa todennäköisesti osittain samaan aikaan. Se aiheuttaisi sen, että välillä säikeet joutuisivat odottamaan vuoroaan lokikirjoituksen kanssa ja näin ollen toiminta hidastuisi hieman. Kokonaisuutena lokiin kirjoittamiseen kuluva aika on murto-osa testien suoritukseen kuluvasta ajasta, jolloin kyseinen pullonkaula tuskin tuottaa merkittävää ongelmaa.

TestRunnerin rinnakkaisuustuesta ei ole tarjolla informaatiota, eikä sitä testattu tässä projektissa. Sen sijaan TestRunnerista ja SoapUI:sta löytyy tuki sille, että useita testitapauksia ajettaisiin samasta projektista rinnakkain. Rinnakkaisuuden tuen kehittäminen vaatisi käytännössä sopivan GroovyScriptin kirjoittamista ja liittämistä lennosta ajettaviin testitapauksiin. Tuki tämäntyyppiselle rinnakkaisuudelle on mahdollista kehittää sovelluskomponenttiin myöhemmässä vaiheessa, jos sellaiselle nähdään tarvetta.

5.3.3 Hylätyt ohjelmakomponentit

Kaikkia tämän projektin yhteydessä kehitettyjä ohjelmakomponentteja ei tulla ottamaan käyttöön lopullisessa julkaisussa. Nämä pois jätettävät toiminnallisuudet liittyvät kaikki komentorivityökaluun, joka oli tarkoitus käynnistää Autotesteristä, kun SoapUI:n projekteja halutaan ajaa. Projektin myöhemmässä vaiheessa SoapUiRunnerista päätettiin tehdä tiiviimpi osa Autotesteriä, jota voitaisiin hyödyntää myös muihin projekteihin liittyvien testien automatisoimiseen. Näin ollen luotu ratkaisu palvelee huomattavasti laajempaa käyttötarkoitusta ja siitä tullee sitä kautta myös paljon pitkäikäisempi ratkaisu.

Kun SoapUiRunner päätettiin laittaa Autotesterin osaksi sen sijaan, että se tulisi ulkoiseksi komponentiksi, täytyi projektin keskeiset toiminnallisuudet erotella omaan kokonaisuuteensa. Lisäksi komentorivityökaluun liittyvä osa tuli erotella erilliseksi käyttöliittymäksi, joka käyttää samaa ohjelmakoodia, mitä myös Autotesteristä käsin suoritetaan. Kehitetystä komentorivityökalusta ei tullut tässä yhteydessä täysin turha, sillä sitä pystytään hyödyntämään SoapUiRunnerin ominaisuuksien testauksessa, koska SoapUiRunnerilla itsellään ei varsinaisesti ole ulkoista käyttöliittymää. Komentorivityökalun avulla SoapUiRunneria on voitu testata ennen kuin se on integroitu Autotesteriin. Näin ollen SoapUiRunneriin liittyvät yksikkötestit ja pienemmän mittakaavan integraatiotestit on voitu suorittaa ensin ja siten varmistaa toiminnallisuuksien toimiminen erillisenä järjestelmänä ennen kuin varmistetaan niiden yhteensopivuus Autotesterin muiden toiminnallisuuksien kanssa.

Sovelluskehityksessä on varsin yleistä, etteivät kaikki kehitetyt ominaisuudet päädy lopulliseen julkaisuun. Tämä johtuu siitä, ettei alussa tiedetä vielä kaikkia tarpeita, joita sovellukseen kohdistuu. Lisäksi asetetut tarpeet saattavat muuttua sovelluskehityksen aikana. Niin kävi myös SoapUiRunnerin tapauksessa. Tässä tapauksessa alkuperäinen suunnittelutyö ja olemassa olevat luokat tukivat komentorivityökalun irrottamista muusta toiminnallisuudesta ja siksi erotteluun liittyvä työ ei ollut kovinkaan suuri. Lisäksi ko-

mentorivityökaluun liittyvät toiminnallisuudet, kuten parametrien lisäämis- ja tarkistustoiminnallisuudet, on luotu sen verran yleiskäyttöisiksi, että niitä voitaisiin hyödyntää myös muissa komentorivipohjaisissa työkaluissa. Niinpä pois jätettävistä toiminnallisuuksista voi olla vielä hyötyä myös tulevaisuudessa.

5.3.4 Sovelluksen testaaminen

Kuten ohjelmistotestauksen teoriaosioissa on käynyt ilmi, liittyy sovelluksen kehitykseen myös paljon testausta. Näin ollen myös tämän projektin toteutuksen yhteydessä on pohdittu ja harjoitettu testausta, jolla pyritään varmistamaan ohjelmiston oikeanlainen toiminta. Lisäksi testaamalla on tutkittu mahdollisia virhetilanteita silloin, kun käyttäjä syöttää virheellistä informaatiota sovellukselle.

Kun käyttäjällä on vain yksi liityntäkohta ohjelmistoon, on siihen liittyvät syötteet testattu suhteellisen nopeasti. Testauksessa täytyy kuitenkin huomioida myös järjestelmästä peräisin olevat poikkeustilanteet sekä se, että eri järjestelmien toiminta poikkeaa toisiinsa nähden. Näitä kaikkia asioita tulisi pyrkiä tutkimaan ja testaamaan mahdollisimman paljon, jotta sovelluksen toimintakyky erilaisissa ympäristöissä voitaisiin varmistaa. Lisäksi tulisi laatia lista niistä edellytyksistä, joiden täyttymisellä sovelluksen voitaisiin katsoa toimivan oikein. Testauksen yhteydessä usein saattaa käydä ilmi niitä poikkeusolosuhteita, joissa sovellus ei vain yksinkertaisesti toimi, eikä sitä ole kannattavaa korjata toimimaan sellaisissa poikkeusolosuhteissa.

Insinöörityön yhteydessä luotiin sovellus, joka tehostaa ja helpottaa sovelluskehityksen yhteydessä tapahtuvaa testausta ja sitä myöten tähtää parempilaatuisiin sovellustuotteisiin. Toteutuksen yhteydessä oli tarvetta perehtyä varsin syvällisesti sovellustestauksen teoriaan. Aiheeseen liittyvän tiedon hankkiminen on myös vahvistanut ymmärrystä siitä, kuinka tärkeässä osassa testauksen tulisi olla kehitettävien sovellusten elinkaaren alkupäässä. Aivan testivetoisesta kehityksestä (Test-Driven Development) tässä yhteydessä ei voida puhua, mutta komponenttien yksikkötestausta ja uusien toiminnallisuuksien testausta yhdessä muun kokoonpanon kanssa toteutettiin kehityksen jokaisessa välissä.

Kehitysvaiheessa SoapUiRunner-projektin oheen luotiin toinen projekti, johon kehitettiin testausluokkia. Testausluokat testaavat jotain kehitettyä toiminnallisuutta. Samalla varmistettiin, että kehitetty luokka tuottaa halutun lopputuloksen, eikä aiheuta mitään

sivuvaikutuksia muulle toiminnalle. Tässä yhteydessä voidaan puhua yksikkötestauksesta (unit testing). Lisäksi sovelluskomponentilla ajettiin manuaalisia testejä useita erilaisia asetuksia ja parametreja käyttäen aina, kun jokin uusi komponentti valmistui. Se kattaa osittain integraatio- ja järjestelmätestaukset.

Testeissä paljastui useimmiten jotain puutteita sovelluksen toiminnassa. Niitä ei olisi ollut yhtä helppoa paikallistaa, ellei samoja asioita olisi aiemmin jo testattu toimiviksi ilman uutta kehitettyä toiminnallisuutta. Kun virheitä on löytynyt, ne on pyritty korjaamaan välittömästi ja sen jälkeen varmistamaan korjauksen onnistuminen suorittamalla testi uudelleen. Lisäksi eräälle komponentille, jonka suoritukseen kuluva aika oli potentiaalinen riskitekijä, toteutettiin tarkempi testi, jossa kellotettiin komponentin suoritukseen kuluvaa aikaa. Kyseinen testaus vaati myös manuaalisia järjestelyitä, koska testattava komponentti etsi levyjärjestelmästä TestRunnerin sijaintia ja siihen kuluvaa aikaa pystyttiin tutkimaan hyödyllisesti vain, jos sijainti muuttui testien välillä.

Kun SoapUiRunner-komponentin kehitys oli edennyt siihen vaiheeseen, että sitä voitiin jo käyttää Autotesterin osana SoapUI-testejä ajettaessa, oli aika aloittaa varsinainen järjestelmätestaus. Järjestelmätestausta suoritettiin palvelimella, josta käsin ajetaan oikeaan IT-projektiin liittyviä testitapauksia SoapUI-ohjelmaa käyttäen. Testeissä ajettiin yksittäisiä onnistumiseen ja epäonnistumiseen päätyviä testitapauksia sekä kokonainen testisarja, joka sisälsi 73 testitapausta. Testiprojektia kokeiltiin ajaa sekä lokaalilta levyltä että verkkosijainnista.

Järjestelmätestauksen aikana sovelluksesta löytyi lukuisia puutteita sekä ohjelmavirheitä, joita ei aiemmissa testeissä ollut havaittu. Virheitä korjattiin sitä mukaa kuin niitä löydettiin, jonka jälkeen samoja testejä ajettiin uudelleen. Samassa yhteydessä havaittiin myös, että tuloksille luotuihin xml-skeemoihin tarvitsi tehdä muutoksia.

Testeissä esiintyi myös sellaisia tapauksia, joita ei xml-tuloksen skeemassa ollut aiemmin huomioitu, kuten esimerkiksi testitapauksia, joissa ei ole yhtään testiaskelta. Lisäksi tässä yhteydessä paljastui, ettei palvelimella asennetun SoapUI:n aiempi versio (4.0.1) mahdollistanut SoapUI:n oman xml-formaatin tuottamista, jolloin siinä pystyttiin käyttämään ainoastaan kehitettyä Extended-formaattia. Koska SoapUI:n versio 4.0.1 on Enorolla yleisesti käytössä ja siirtyminen tuoreempaan versioon vaatisi olemassa olevien testien toiminnallisuuden varmistamista, olisi tämä voinut olla mahdollisesti suurempikin ongelma. Pelastava tekijä oli se, että monipuolisempi extended-formaatti

oli jo kehitetty ja sen toiminta ei ollut yhtä riippuvaista SoapUI:n tarjoamista toiminnallisuuksista. Lopulta järjestelmätestaamisen avulla saavutettiin se, että SoapUiRunner suoriutui ongelmitta testien ajosta ja se voitiin ottaa käyttöön.

6 Kehitettyjen ratkaisujen analysointi ja johtopäätökset

Kehittämällä SoapUiRunner mahdollistettiin SoapUI:n testitapausten ajo Autotesterin skriptistä käsin. Tämä ei tarkoita sitä, että SoapUI:sta päästäisiin kokonaan eroon. Kaikki tulevat testitapaukset joudutaan edelleenkin tekemään SoapUI:sta käsin. Se ei ole välttämättä huono asia, sillä SoapUI:n käyttöliittymä on helppo ja monipuolinen. Se, mitä kehitetyllä sovelluskomponentilla saavutettiin, oli kyky automatisoida SoapUI:lle tehtyjen projektien ajo.

Liitteessä 4 on nähtävillä esimerkki ATScriptistä, jolla ajetaan yksi SoapUI:n testiprojekti. Mikään ei periaatteessa estä, ettei samaan testiin voisi laittaa ajoon vaikka kymmentä vastaavaa testitapausta sisältävää skriptiä. Niistä saataisiin joka tapauksessa talteen kaikkiin epäonnistuneisiin testitapauksiin liittyvät tiedot, joilla virheet voidaan yksilöidä. Käytännössä tämä tarkoittaa, että ennen jokainen testiprojekti olisi jouduttu yksitellen käynnistämään mutta nyt ne kaikki voidaan suorittaa yhdessä ajamalla Autotesterillä yhden testisarjan. Tämä tarkoittaa, että testauksessa on otettu askel eteenpäin. SoapUiRunner toteuttaa ensimmäisen tavoitteen täyttymisen, joka insinöörityölle asetettiin.

Pitkällä aikavälillä voidaan arvioida, että SoapUiRunner auttaa parantamaan kehitettävien sovellusten laatua. Arvio perustuu siihen, että regressiotestejä voidaan lisätä saavutetun automatisoinnin ansiosta. Toimitusprojektit voivat nopeutua tulevaisuudessa, kun osa aiemmin toimitukseen asti päässeistä virheistä voidaan havaita jo aiemmin sen seurauksena, että automaattisesti ajettavia testejä tehdään enemmän. Siten insinöörityön yhteydessä kehitetyllä sovelluksella saattaa olla positiivisia vaikutuksia sovelluksien koko kehitys- ja toimitusketjun pituuteen.

Toiseen insinöörityössä asetettuun tavoitteeseen liittyen insinöörityön yhteydessä tehtiin selvitystyö. Selvitystyössä arvioidaan, että järkevin ratkaisu on pyrkiä integroimaan myös selvitystyön kohteena oleva SapSim osaksi Autotesteriä. Tämänkin tavoitteen osalta päästiin ratkaisuehdotukseen mutta sen tuomasta edusta ei ole tässä vaiheessa vielä konkreettista näyttöä. Tämän insinöörityön aikana kyseiseen projektiin liittyviä testejä ei ole vielä liitetty osaksi regressiotestejä.

Opinnäytetyn saavutuksiin liittyvää jatkokehitystä voidaan tehdä myös tämän projektin päätyttyä. SoapUiRunner-komponentin toiminnallisuuksia on mahdollista jatkossa kehittää entistäkin monipuolisemmaksi. Siihen voitaisiin lisätä esimerkiksi mahdollisuus

määritellä suoritettavalle projektille www-sovelluspalvelun yhteysosoite, jolla ohitettaisiin projektiin määritetty osoite. Näin ollen jos sama www-sovelluspalvelu olisi asennettuna useaan testijärjestelmään esimerkiksi eri versioilla varustettuna, niitä voitaisiin testata helposti ilman, että varsinaista testiprojektia tarvitsisi muokata. Lisäksi testien ajoa voitaisiin nopeuttaa lisäämällä SoapUiRunneriin tuki SoapUI-projektiin kuuluvien testitapauksien rinnakkaiselle suorittamiselle. Siten useita erilaisia testitapauksia sisältäviä projekteja voitaisiin suorittaa nopeammalla tahdilla.

7 Yhteenveto

Insinöörityön konkreettinen tulos on sovelluskomponentti, joka helpottaa testaajien arkea. Sovelluskomponentti mahdollistaa www-sovelluspalveluiden testaukseen keskittyvien testien automatisoinnin ja lisäämisen osaksi yrityksen automatisoitujen regressiotestien portfoliota. Lisäksi insinöörityön tuloksena tehtiin selvitys siitä, miten sellaisia www-sovelluspalveluiden testejä, joita joudutaan ajamaan synkronisesti toisten testien kanssa, voidaan tulevaisuudessa myös automatisoida.

SoapUiRunner on liitetty osaksi Autotester-nimistä testiskriptien ajo-ohjelmaa. Se liitettiin Autotesteriin dll-kirjastotiedostona ja muutamana mukana kulkevana resurssitiedostona, jotka asentuvat jatkossa yhdessä Autotesterin kanssa. SoapUiRunner sisältää 4500 riviä ohjelmakoodia, jota jouduttiin uudelleenjärjestelemään ja muokkaamaan moneen otteeseen prosessin aikana. Tämän lisäksi sovellukseen kuuluu käyttöliittymäosia, jotka koostuvat myös tuhansista koodiriveistä. Nyt Autotesteristä käsin voidaan ajaa skriptejä, joissa ajetaan SoapUI:lla tehtyjä testiprojekteja tai niissä olevia yksittäisiä testitapauksia. Lisäksi testaajille on tarjolla graafinen työkalu, jolla SoapUI:n projektiin sisältyvistä testeistä saadaan automaattisesti generoitua valmiit testiskriptit. Sen avulla säästetään tuntikausia testaajien aikaa.

Sovelluskomponentti mahdollistaa nyt lukuisten testien liittämisen osaksi jatkuvasti ajettavia regressiotestejä, joita voidaan ajaa automaattisesti aina tarpeen mukaan. Testaajat ovat ottaneet asian mahdollistavan ominaisuuden hyvin vastaan ja ovat siihen tyytyväisiä. Sovelluskomponenttia ruvetaan tämän jälkeen ylläpitämään ja siihen voidaan kehittää jatkossa myös uusia ominaisuuksia, mikäli testaustarpeet lisääntyvät.

8 Loppusanat ja työn laadun arviointi

Insinöörityön valmistuminen vaati paljon ponnisteluja ja uskoa siihen, että työ valmistuisi jossain vaiheessa. Insinöörityön tekeminen oli prosessi, joka opetti tekijäänsä paljon paitsi itse aiheesta myös työskentelytavoista. Työn alkuvaiheessa koko aihealue oli täysin uusi ja vieras. Teoriaa ja informaatiota testaukseen liittyen on tarjolla paljon.

Uudelle aiheelle tyypilliseen tapaan aloituskohdan löytäminen tuntui erittäin haastavalta. Aloituskohdan löytämistä vaikeutti myös se, ettei alussa ollut selvillä tarkkaa tietoa siitä, mitä käytännön toteutuksen pitäisi oikeastaan tehdä. Lopulta punainen lanka työlle löytyi: ensin oli tutustuttava aiheen teoriaan, jotta siihen liittyvät käsitteet ja tavoitteet olisivat selkeitä. Sitten oli hyvä perehtyä projektiin liittyvään dokumentaatioon ja lopulta vielä haastatella uudelleen toteutukseen sidoksissa olevia henkilöitä. Näiltä henkilöiltä sai tarkemman tiedon siitä, mitä työltä oikeastaan haluttiin. Sitä varten tarvitsi vain ensin osata kysyä ne oikeat kysymykset. Tämän jälkeen prosessi alkoi kulkea oikeaan suuntaan ja tavoite alkoi hahmottua selkeästi.

Insinöörityö on uskoakseni varsin laadukas. Työ on selkeästi rajattu, jonka lisäksi siinä on esitetty selkeät tavoitteet, jotka työllä haluttiin saavuttaa. Työn rakenne etenee luontevasti ja varsinainen teoriaosuus on riittävä, jotta vähemmän aiheeseen perehtynyt henkilö pystyy hankkimaan käytännön työhön liittyvän tiedon. Lisäksi esitetyt asiat pohjautuvat pääosin sellaisiin luotettavaksi arvioituihin lähteisiin, jotka ovat julkisesti saatavilla. Työ sisältää myös selkeän pulman ja siihen esitetyn käytännön ratkaisun.

Insinöörityön yhdeksi mahdolliseksi heikkoudeksi arvioin sen, että tavoitteet jakoivat aihetta hieman kahteen suuntaan. Tavoitteet ovat kuitenkin linjassa toisiinsa, kun ensimmäiseen tavoitteeseen pääseminen paljastui olevan edellytyksenä myös toisen tavoitteen saavuttamiselle. Siksi näiden aiheiden yhteen saattaminen onnistui mielestäni kohtalaisen hyvin.

Kokonaisuutena olen insinöörityöhöni erittäin tyytyväinen. Saavutin sillä jotain, mikä ratkaisee jonkin olemassa olevan ongelman. Lisäksi työn tuloksella saavutetaan taloudellista hyötyä, koska tehdyllä sovelluskomponentilla säästetään huomattavasti testaajien aikaa.

Lähteet

About SoapUI – Features. 2013. Verkkosivu. SmartBear. < http://www.soapui.org/About-SoapUI/features.html>. Luettu 26.3.2013.

Automated Testing – TestComplete. 2013. Verkkosivu. SmartBear corporation. < http://smartbear.com/products/qa-tools/automated-testing-tools>. Luettu 20.2.2013.

Dustin, E., Garret, T. & Gauf, B. 2009. Implementing Automated Software Testing. Boston: Pearson Education Inc.

Functional testing. 2013. Verkkosivu. Opensourcetesting.org. <http://www.opensourcetesting.org/functional.php> . Luettu 20.2.2013.

Groovy – a dynamic language for the Java Platform 2013. Verkkosivu. Springsource. < http://groovy.codehaus.org/>. Luettu 26.3.2013.

Gross, Hans-Gerhard. 2005. Component-Based Software Testing with UML. Berlin: Springer-Verlag.

Kaner C., Bach J. & Pettichord B. 2002. Lessons learned in software testing. New York: John Wiley & Sons, Inc.

Patton, Tony. 2008. Easily test Web services with soapUI. Verkkodokumentti. <http://www.techrepublic.com/blog/programming-and-development/easily-test-web-services-with-soapui/699>. Päivitetty 3.7.2008. Luettu 3.3.2013.

Pressman 2001. Software Engineering – a practitioner's approach. 5. painos. New York: McGraw-Hill.

Ritscher, April. 2010. Incorporating user scenarious in test Design. Conference Paper Excerpt from the conference proceedings. 28th annual software quality conference.

SapSim Guide. 2013. Yrityksen sisäinen dokumentti. Enoro Oy.

Sprott, David & Wilkes Lawrence. 2004. Verkkodokumentti. Understanding Service-Oriented Architecture. Microsoft Developer Network. <http://msdn.microsoft.com/en-us/library/aa480021.aspx>. Luettu 1.4.2013.

Standard glossary of terms used in Software Testing. 2012. Verkkodokumentti. International Software Testing Qualifications Board (ISTQB). <http://www.istqb.org/downloads/finish/20/101.html>. Luettu 10.2.2013.

Tatham, Simon. 1999. How To Report Bugs Effectively. Verkkodokumentti. < http://www.chiark.greenend.org.uk/~sgtatham/bugs.html> . Luettu 24.3.2013.

Tietotekniikan termitalkoot. 2012. Verkkosivu. Sanastokeskus TSK ry. <http://www.tsk.fi/tsk/termitalkoot/fi/hakemistot-267.html?page=get_id&id=ID0176&vocabulary_code=TSKTT>. Päivitetty 10.1.2012. Luettu 29.3.20913.

Tour Visual Studio Test Professional 2010. 2013. Verkkosivu. Microsoft corporation. <https://www.microsoft.com/visualstudio/en-us/try/test-professional-2010-tour/get-started>. Luettu 20.2.2013.

Whitmill, Kelly. 2010. Writing Effective Defect Reports. Verkkodokumentti. FYICenter. <http://sqa.fyicenter.com/art/Writing_Effective_Defect_Reports.html >. Luettu 24.3.2013.

Working with SoapUI – Getting Started. 2013. Verkkosivu. SmartBear. <http://www.soapui.org/Working-with-soapUI/getting-started.html>. Luettu 26.3.2013.

SoapUiRunner-komponentin asetustiedoston formaatti, sisältö ja skeema

Asetustiedoston sisältö näyttää alla olevalta. Se toteuttaa hyvin laaditun xml:n rakenteen eli siinä on yksi juurielementti ja jokainen sisäelementti päätetään xml:n sääntöjen mukaan. Lisäksi sisällön oikeellisuus validoidaan alempana olevaa xml-skeemaa vasten. Validointi tarkistaa, että kaikki pakolliset kentät ovat mukana ja niissä on odotetun tyyppistä informaatiota.

Esimerkki xml-muotoisesta asetustiedostosta, jota SoapUI Runner-komponentti käyttää:

”Autotester.Library.SoapUI.config”:

```
<?xml version="1.0" encoding="utf-8"?>
<Settings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <PathToSoapUITestRunner>C:\soapUI4.5.1\bin\TestRunner.bat
    </PathToSoapUITestRunner>
    <PathToTestReports>D:\AutoTester\Output\</PathToTestReports>
    <Logging>Enabled</Logging>
    <LogLevel>INFO</LogLevel>
    <ReportingType>EXTENDED</ReportingType>
    <HtmlReport>Enabled</HtmlReport>
    <ContinueOnErrors>Disabled</ContinueOnErrors>
    <NumberOfDaysToKeepLogs>30</NumberOfDaysToKeepLogs>
    <NumberOfDaysToKeepResultFiles>30</NumberOfDaysToKeepResultFiles>
</Settings>
```

Laadittu xml-skeema, jolla määritellään, minkä tyyppinen asetustiedoston tulee olla:

”SoapUIRunnerSettingsSchema.xsd”:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-
in-
stance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xs="http://www.w3.
org/2001/XMLSchema" attributeFormDefault="unqualified" elementFormDefault="qu
alified">
  <xsd:element name="Settings">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="PathToSoapUITestRunner" type="xsd:string" />
        <xsd:element name="PathToTestReports" type="xsd:string" />
        <xsd:element name="Logging" type="xsd:string" />
        <xsd:element name="LogLevel" type="xsd:string" />
        <xsd:element name="ReportingType" type="xsd:string" />
        <xsd:element name="ContinueOnErrors" type="xsd:string" />
        <xsd:element name="HtmlReport" type="xsd:string" />
        <xsd:element name="NumberOfDaysToKeepLogs" type="xsd:int" />
         <xsd:element name="NumberOfDaysToKeepResultFiles" type="xsd:int" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xs:schema>
```

Esimerkki extended-formaatin mukaisesta xml-tulostiedostosta

”TestSuite 1_ExtendedFormat_23_03_2013_23_11_56.xml”:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<TestSuiteObject
    Name='TestSuite 1'
    ExecutionDate='23.03.2013 23:11:56.724'
    NumberOfTestCases='5'
    TotalTimeTaken='2935'
    Pro-
jectPath='E:\DocumentCenter\Metropolia\Thesis\SoapUITesting\JSU-
TestProject-soapui-project.xml'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>
    <TestCases>
        <TestCaseObject Name='GetWeatherByZipCode TestCase'
            Status='FINISHED' Duration='201' NumberOfTestSteps='1'>
            <TestSteps>
                <TestStepObject Name='GetWeatherByZipCode'
                    Status='UNKNOWN' Duration='201'/>
            </TestSteps>
        </TestCaseObject>
        <TestCaseObject Name='TestCase 1' Status='FINISHED'
            Duration='1069' NumberOfTestSteps='2'>
            <TestSteps>
                <TestStepObject Name='Delay'
                    Status='OK' Duration='1000'/>
                <TestStepObject Name='Properties'
                    Status='OK' Duration='69'/>
            </TestSteps>
        </TestCaseObject>
        <TestCaseObject Name='GetWeatherByPlaceName TestCase'
            Status='FINISHED' Duration='202' NumberOfTestSteps='1'>
            <TestSteps>
                <TestStepObject Name='GetWeatherByPlaceName'
                    Status='UNKNOWN' Duration='202'/>
            </TestSteps>
        </TestCaseObject>
        <TestCaseObject Name='GetWeatherByZipCode TestCase'
            Status='FINISHED' Duration='199' NumberOfTestSteps='1'>
            <TestSteps>
                <TestStepObject Name='GetWeatherByZipCode'
                    Status='UNKNOWN' Duration='199'/>
            </TestSteps>
        </TestCaseObject>
        <TestCaseObject Name='GetWeatherByPlaceName TestCase'
            Status='FAILED' Duration='21338' NumberOfTestSteps='1'>
            <TestSteps>
                <TestStepObject Name='GetWeatherByPlaceName'
                    Status='FAILED' Duration='21338'/>
            </TestSteps>
        </TestCaseObject>
    </TestCases>
</TestSuiteObject>
```

Esimerkki SoapUiRunnerin lokitiedoston sisällöstä

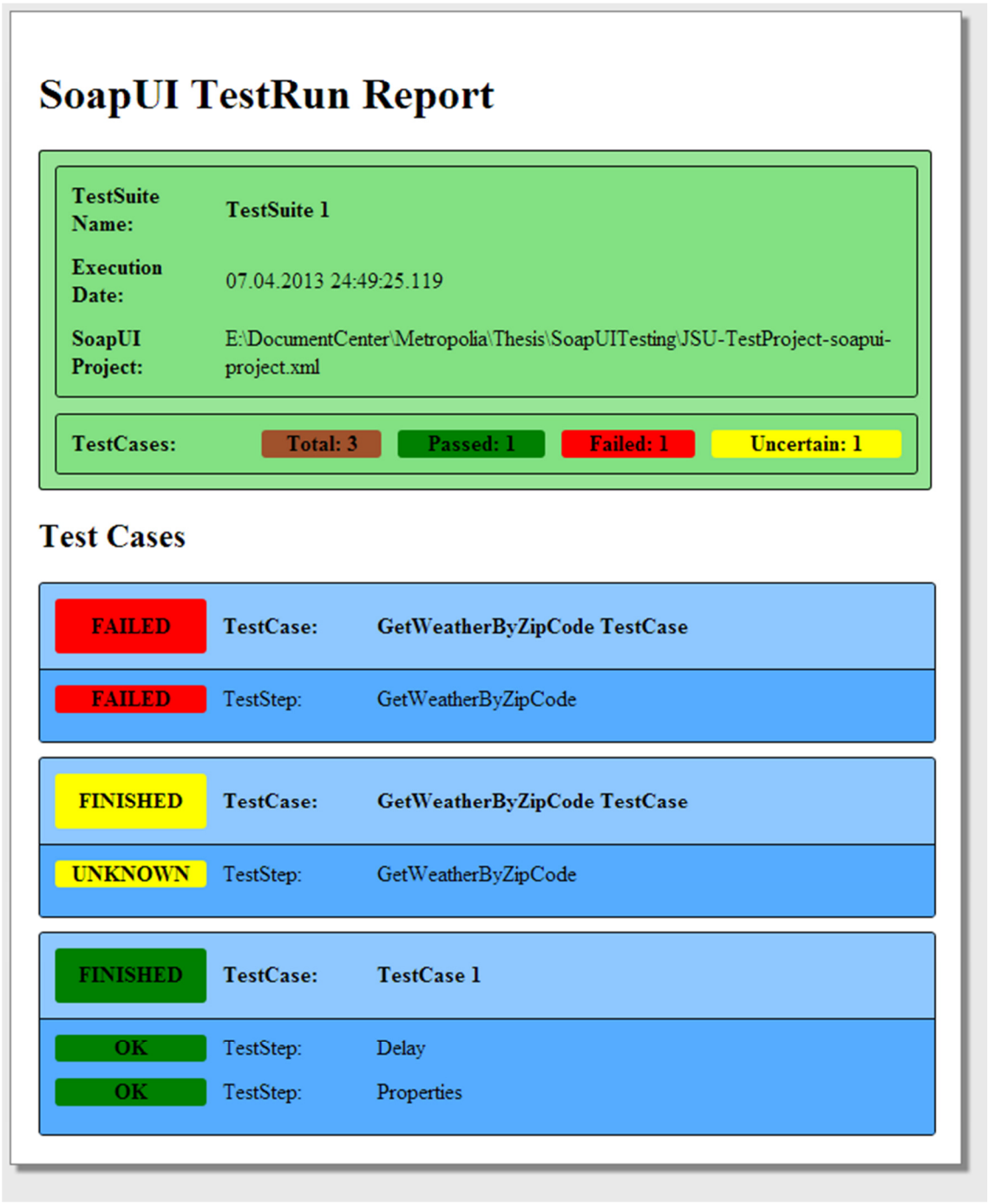
”SoapUIRunner_26_03_2013.log”:

```
26/03/2013 15:13:08 INFO    [Validation] TestRunner location confirmed
26/03/2013 15:13:08 INFO    [Validation] Runtime settings validation PASSED
26/03/2013 15:13:08 INFO    [Validation] Resource files' & folders' validation PASSED
26/03/2013 15:13:08 INFO    [Output Script] Adding Test Suite GroovyScript to the project file as a TearI
26/03/2013 15:13:08 INFO    [AddTearDownScrip] Updated Project File "C:\Users\janne.suomalainen\Documents
26/03/2013 15:13:08 INFO    [AddTearDownScrip] Added new teardown script for Test Project from file"E:\Au
26/03/2013 15:13:08 SYSTEM  ------------------------------------------------------------
26/03/2013 15:13:08 SYSTEM  [SoapUIRunner] Starting to ExecuteSoapUI TestRunner
26/03/2013 15:13:08 SYSTEM  ------------------------------------------------------------
26/03/2013 15:13:08 INFO    [Output] Output folder for TestRunner set to "E:\AutoTester\Output\"
26/03/2013 15:13:08 INFO    [Parameters] Parameters for process "TestRunner.bat":
26/03/2013 15:13:08 INFO    [Parameters] Working directory: C:\Program Files (x86)\SmartBear\soapUI-4.5.1
26/03/2013 15:13:08 INFO    [Parameters] Argument 1:  -s"TestSuite 1"
26/03/2013 15:13:08 INFO    [Parameters] Argument 2:  -fE:\AutoTester\Output\
26/03/2013 15:13:08 INFO    [Parameters] Argument 3:  E:\AutoTester\SoapUi\SoapUiProjectInProgress_634999
26/03/2013 15:13:08 SYSTEM  ------------------------------------------------------------
26/03/2013 15:13:08 SYSTEM  [SoapUIRunner] SoapUI TestRunner Execution Started
26/03/2013 15:13:08 SYSTEM  ------------------------------------------------------------
26/03/2013 15:13:08 INFO    [SoapUI TestRunner] Process start time: 26/03/2013 15:13:08
26/03/2013 15:13:15 INFO    soapUI 4.5.1 TestCase Runner
Configuring log4j from [C:\Program Files (x86)\SmartBear\soapUI-4.5.1\bin\soapui-log4j.xml]
15:13:10,444 INFO  [DefaultSoapUICore] initialized soapui-settings from [C:\Users\janne.suomalainen\soapu
15:13:11,030 INFO  [WsdlProject] Loaded project from [file:/E:/AutoTester/SoapUi/SoapUiProjectInProgress_
15:13:11,280 INFO  [SoapUITestCaseRunner] Running soapUI tests in project [JSU-TestProject]
15:13:11,280 INFO  [SoapUITestCaseRunner] Running TestSuite [TestSuite 1], runType = SEQUENTIAL
15:13:11,294 INFO  [SoapUITestCaseRunner] Running soapUI testcase [TestCase 1]
15:13:11,294 INFO  [SoapUITestCaseRunner] running step [Delay]
15:13:12,301 INFO  [SoapUITestCaseRunner] running step [Properties]
15:13:12,853 INFO  [SoapUITestCaseRunner] Finished running soapUI testcase [TestCase 1], time taken: 1077
15:13:12,853 INFO  [SoapUITestCaseRunner] Running soapUI testcase [GetWeatherByPlaceName TestCase]
15:13:12,859 INFO  [SoapUITestCaseRunner] running step [GetWeatherByPlaceName]
15:13:13,772 INFO  [SoapUITestCaseRunner] Finished running soapUI testcase [GetWeatherByPlaceName TestCas
15:13:13,773 INFO  [SoapUITestCaseRunner] Running soapUI testcase [GetWeatherByZipCode TestCase]
15:13:13,773 INFO  [SoapUITestCaseRunner] running step [GetWeatherByZipCode]
15:13:14,141 INFO  [SoapUITestCaseRunner] Finished running soapUI testcase [GetWeatherByZipCode TestCase]
15:13:14,142 INFO  [SoapUITestCaseRunner] Running soapUI testcase [GetWeatherByPlaceName TestCase]
15:13:14,142 INFO  [SoapUITestCaseRunner] running step [GetWeatherByPlaceName]
15:13:14,512 INFO  [SoapUITestCaseRunner] Finished running soapUI testcase [GetWeatherByPlaceName TestCas
15:13:14,513 INFO  [SoapUITestCaseRunner] Running soapUI testcase [GetWeatherByZipCode TestCase]
15:13:14,513 INFO  [SoapUITestCaseRunner] running step [GetWeatherByZipCode]
15:13:14,881 INFO  [SoapUITestCaseRunner] Finished running soapUI testcase [GetWeatherByZipCode TestCase]
15:13:15,159 INFO  [SoapUITestCaseRunner] TestSuite [TestSuite 1] finished with status [FINISHED] in 3598

26/03/2013 15:13:15 INFO    [SoapUI TestRunner] Process end time:   26/03/2013 15:13:15
26/03/2013 15:13:15 INFO    [SoapUI TestRunner] Process Exit code:  0
26/03/2013 15:13:15 SYSTEM  ------------------------------------------------------------
26/03/2013 15:13:15 SYSTEM  [SoapUIRunner] SoapUI TestRunnerExecution Stopped
26/03/2013 15:13:15 SYSTEM  ------------------------------------------------------------
```

Esimerkki SoapUI-projektin ajavasta Autotester-skriptistä

```
#const TestSuiteName = "TestSuite 1"
#const SoapUiTestProject =
"C:\Users\janne.suomalainen\Documents\Thesis\SoapUITesting\JSU-
TestProject-soapui-project.xml"


' Run SoapUI Project :
SoapUI.ExecuteTestSuite(SoapUiTestProject, TestSuiteName)
```

Esimerkki SoapUiRunnerin tuottamasta html-raportista



SoapUI TestRun Report

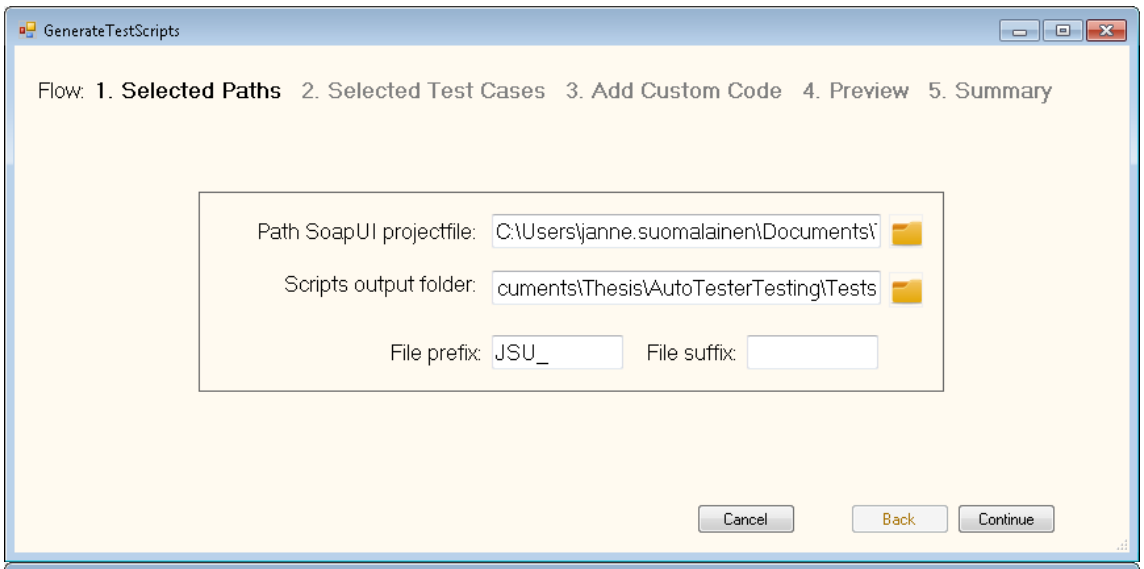
TestSuite Name:	**TestSuite 1**
Execution Date:	07.04.2013 24:49:25.119
SoapUI Project:	E:\DocumentCenter\Metropolia\Thesis\SoapUITesting\JSU-TestProject-soapui-project.xml

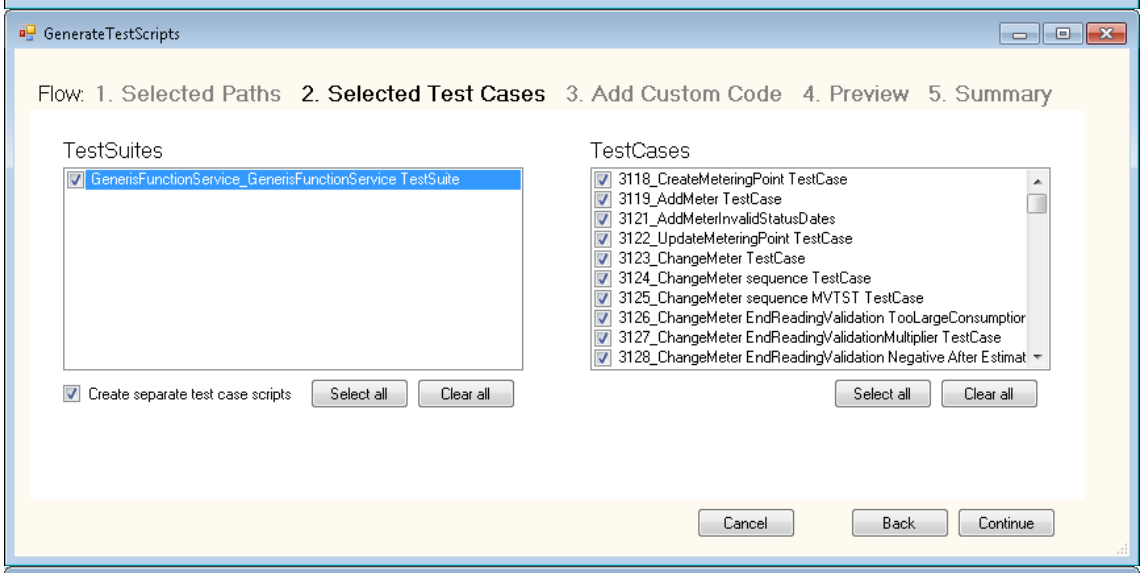
TestCases: **Total: 3** **Passed: 1** **Failed: 1** **Uncertain: 1**

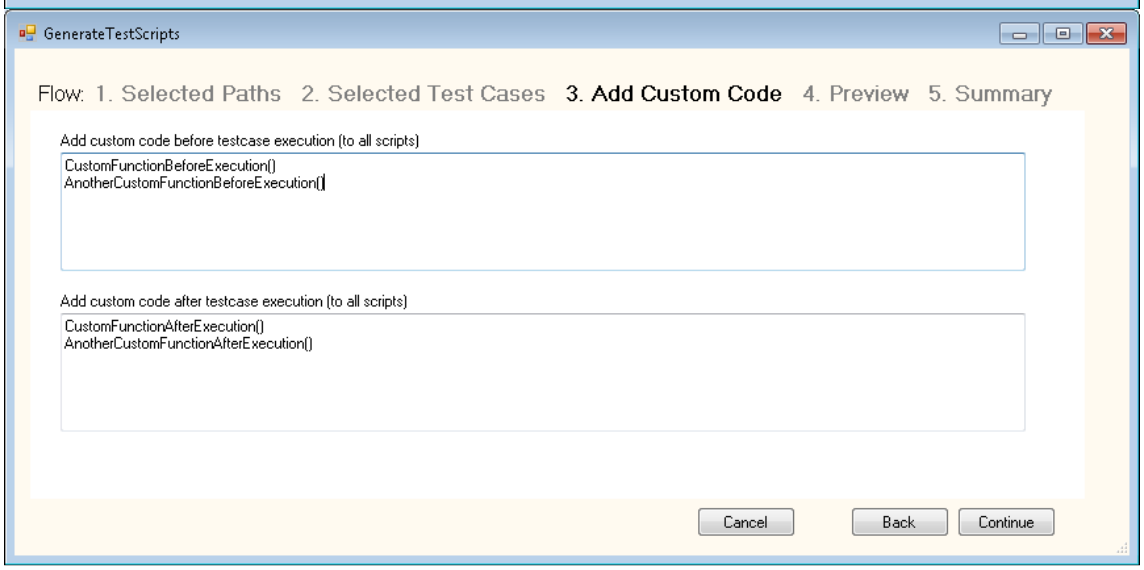
Test Cases

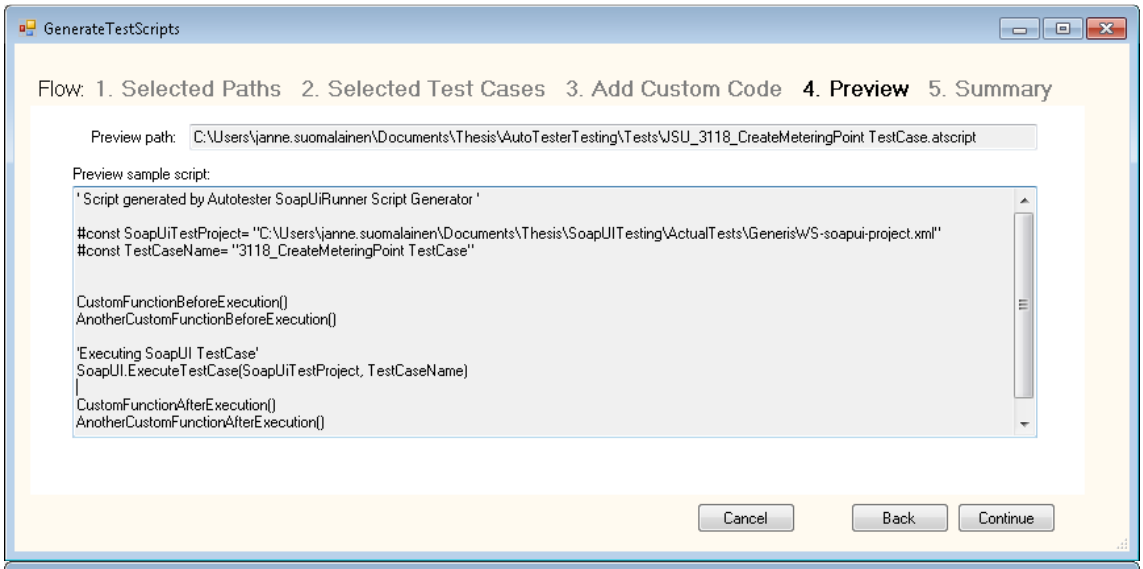
FAILED	**TestCase:**	**GetWeatherByZipCode TestCase**
FAILED	TestStep:	GetWeatherByZipCode
FINISHED	**TestCase:**	**GetWeatherByZipCode TestCase**
UNKNOWN	TestStep:	GetWeatherByZipCode
FINISHED	**TestCase:**	**TestCase 1**
OK	TestStep:	Delay
OK	TestStep:	Properties

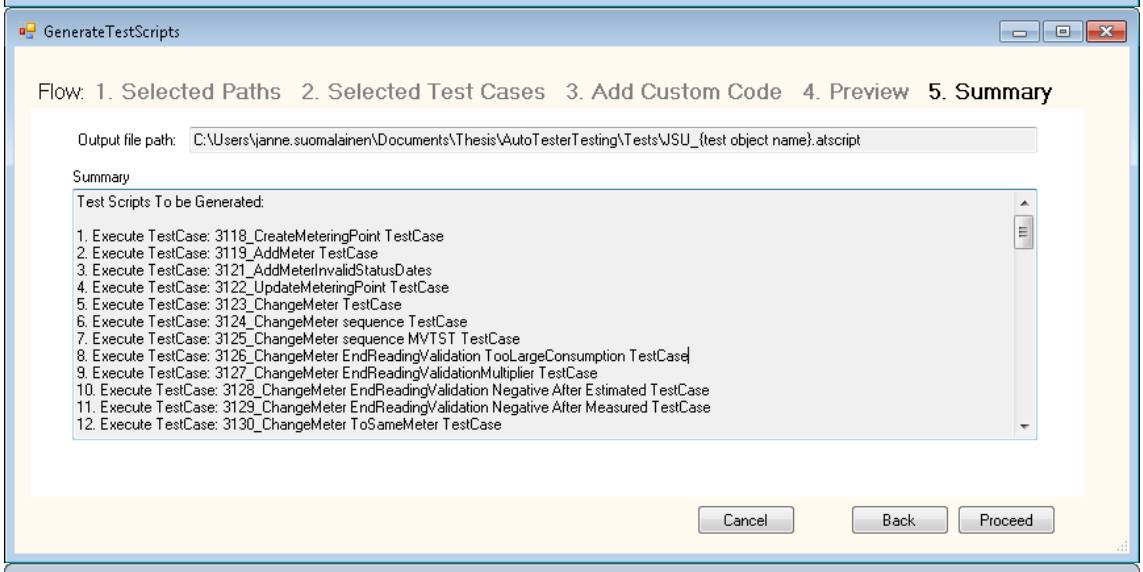
SoapUI:n Autotesterskriptien luontityökalun käyttöliittymä.

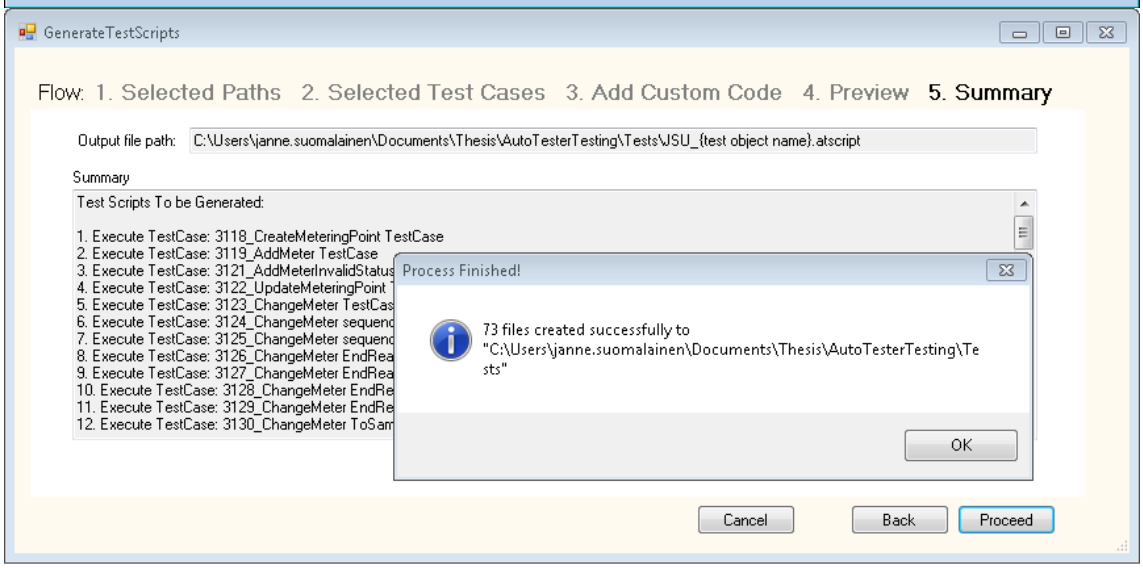







GenerateTestScripts
Flow: 1. Selected Paths 2. Selected Test Cases 3. Add Custom Code 4. Preview 5. Summary
Preview path: C:\Users\janne.suomalainen\Documents\Thesis\AutoTesterTesting\Tests\JSU_3118_CreateMeteringPoint TestCase.atscript
Preview sample script:
' Script generated by Autotester SoapUiRunner Script Generator '
#const SoapUiTestProject= "C:\Users\janne.suomalainen\Documents\Thesis\SoapUITesting\ActualTests\GenerisWS-soapui-project.xml"
#const TestCaseName= "3118_CreateMeteringPoint TestCase"
CustomFunctionBeforeExecution()
AnotherCustomFunctionBeforeExecution()
'Executing SoapUI TestCase'
SoapUI.ExecuteTestCase(SoapUiTestProject, TestCaseName)
CustomFunctionAfterExecution()
AnotherCustomFunctionAfterExecution()
Cancel
Back
Continue


GenerateTestScripts
Flow: 1. Selected Paths 2. Selected Test Cases 3. Add Custom Code 4. Preview 5. Summary
Output file path: C:\Users\janne.suomalainen\Documents\Thesis\AutoTesterTesting\Tests\JSU_{test object name}.atscript
Summary
Test Scripts To be Generated:
1. Execute TestCase: 3118_CreateMeteringPoint TestCase
2. Execute TestCase: 3119_AddMeter TestCase
3. Execute TestCase: 3121_AddMeterInvalidStatusDates
4. Execute TestCase: 3122_UpdateMeteringPoint TestCase
5. Execute TestCase: 3123_ChangeMeter TestCase
6. Execute TestCase: 3124_ChangeMeter sequence TestCase
7. Execute TestCase: 3125_ChangeMeter sequence MVTST TestCase
8. Execute TestCase: 3126_ChangeMeter EndReadingValidation TooLargeConsumption TestCase
9. Execute TestCase: 3127_ChangeMeter EndReadingValidationMultiplier TestCase
10. Execute TestCase: 3128_ChangeMeter EndReadingValidation Negative After Estimated TestCase
11. Execute TestCase: 3129_ChangeMeter EndReadingValidation Negative After Measured TestCase
12. Execute TestCase: 3130_ChangeMeter ToSameMeter TestCase
Cancel
Back
Proceed


GenerateTestScripts
Flow: 1. Selected Paths 2. Selected Test Cases 3. Add Custom Code 4. Preview 5. Summary
Output file path: C:\Users\janne.suomalainen\Documents\Thesis\AutoTesterTesting\Tests\JSU_{test object name}.atscript
Summary
Test Scripts To be Generated:
1. Execute TestCase: 3118_CreateMeteringPoint TestCase
2. Execute TestCase: 3119_AddMeter TestCase
Process Finished!
73 files created successfully to "C:\Users\janne.suomalainen\Documents\Thesis\AutoTesterTesting\Tests"
OK
Cancel
Back
Proceed