NIKO KONTIO

RADIOVERKKO-OHJAIMEN VERTAILUTESTAUKSEN TESTIAUTOMAATIOMITTAUKSIEN TOTEUTUS

Tietotekniikan koulutusohjelma

2014

RADIOVERKKO-OHJAIMEN VERTAILUTESTAUKSEN TESTIAUTOMAATIO-MITTAUKSIEN TOTEUTUS

Kontio, Niko

Satakunnan ammattikorkeakoulu

Tietotekniikan koulutusohjelma

Maaliskuu 2014

Ohjaaja: Haataja, Rauli

Sivumäärä: 32

Liitteitä: 5

Asiasanat: Ohjelmistotestaus, Automaatio, Jenkins, Robot Framework.

Opinnäytetyössä etsittiin uusia käyttökohteita testiautomaatiokehys Robot Frameworkin ja jatkuvan integraatiosovellus Jenkinsin käyttöön radioverkko-ohjaimen perustoiminnallisuuksien hyväksyntätestauksessa. Opinnäytetyössä Robot Framework ja Jenkins asennettiin testausympäristöön.

Opinnäytetyössä kartoitettiin testitapauksien vertailutestaustoteutusta. Robot Frameworkista selvitettiin myös, mitä tapoja se tarjoaa testien muotoiluun ja mitä asioita voidaan uudelleen käyttää myöhemmin. Jenkinsin rooli oli toimia testiympäristön ajastimena ja pitää huoli testausdatan organisoinnista.

Jenkinsille tehtiin ensin manuaalista testausta, josta siirryttiin ajastettuihin automatisoituihin vertailutestauksiin. Käytännön toteutuksesta saatujen kokemuksien perusteella voidaan todeta, että Jenkinsillä ja Robot Frameworkilla voidaan luoda joustavia ja tehokkaita testausympäristöjä.

RADIO NETWORK CONTROLLER COMPARISON TESTAUTOMATIONS MEASUREMENT IMPLEMENTATION

Kontio, Niko

Satakunta University of Applied Sciences

Degree Programme in Information Technology

March 2014

Supervisor: Haataja, Rauli

Number of pages: 32

Appendices: 5

Keywords: Software testing, Automation, Jenkins, Robot Framework

---

The study looked for new uses for the test frame Robot Framework and continuous integration program Jenkins' use of the radio network controller basic functionalities acceptance testing. In the thesis project Robot Framework and Jenkins were installed into an testing environment.

The study surveyed comparison test procedure for test cases. Robot Framework was surveyed for what objects can be repurposed for later use. Jenkins' role was to serve in a test environment as a timer and to take care of the organization of test data.

Jenkins was first used for manual testing and later for automated timed comparison testing. According to the information that was gathered from practical experience, the Jenkins and Robot Framework can be used to make flexible and effective testing environments.

SISÄLLYSLUETTELLO

KÄSITTEET JA LYHENTEET

Jatkuva integraatio	Continuous integration, ohjelmistojen kehitysversioden sulautuskäytäntö.
mcRNC	multicontrol Radio Nework Controller, tuotenimi radioverkko-ohjaimelle.
Perustoiminallisuus	Sisältää järjestelmäkäynnistystä, yksikkö-käynnistyksiä, redundanttisuutta, jne.
Python	Tulkattava ohjelmointikieli. Pythonia käytetään usein sen yksinkertaisen syntaksin ja korkean tason tietorakenteiden takia.
Radioverkko-ohjain	Network Controller, radioverkkoliikennettä ohjaileva laite.
SSH	Secure Shell, protokolla, jolla lähetetään tietoa salattuna tietokoneverkkojen sisällä.
Suorituskykytestaus	Performance test, suorituskykyä seuraava testaus.
Testiautomaatiokehys	Test automation frame, rajapinta missä testitapaukset ajetaan.
Testipeti	Ohjelmistorunko jonka avulla testejä on helppo suorittaa.
Vertailutestaus	Comparison testing, arvoihin perustuva testausmenetelmä.

1. JOHDANTO

Kohdealue oli radioverkko-ohjaimen testauksen suunnitteleminen ja vertailutestauksen toteutus automaattisiin ympäristöihin. Raportin tarkoitus oli nostaa esiin asioita, joita pitää käsitellä ja ratkaista, kun testausta automatisoidaan. Työ toteutettiin Nokia Solutions and Networksin ympäristöissä.

Työn lähtökohtana oli tarkoitus löytää mahdollisia käyttökohteita automaatiotestaukselle useasti toistuville testauskohteille, joita esiintyy perustoiminallisuuksiin kohdistuviin suorituskykytestauksiin. Opinnäytetyötä lähdettiin viemään läpi projektina, joka alkoi kesällä 2013. Tavoitteena oli saada toimiva testiympäristö vertailutestausdemonstraatioita varten, joka olisi osana hyväksyntätestausta (Kuvio.1 s.9). Ajatuksena oli, että ympäristön testausmallia sovellettaisiin muihinkin kohteisiin, mikäli se koettaisiin tarpeelliseksi.

Isona osana opinnäytetyötä oli selvittää, miten testiautomaatiokehys Robot Framework soveltuisi osana radioverkkoympäristön suorituskykytestausta ja kuinka tämä saataisiin automatisoitua jatkuvan integraatio sovellus Jenkinsillä. Nokia Solutions and Networksin olemassa olevat testausmenetelmät ja testausympäristön toimivuus olivat suurena osana tutkimusta.

2. TYÖN LÄHTÖKOHDAT

2.1. Toimeksiantaja

Nokia Solutions and Networks eli NSN sai alkunsa huhtikuun 1. päivä vuonna 2007 kahden yhtiön yhteenliittymästä. Nämä yhtiöt olivat Nokia Networks ja Siemens Communications. Yrityksen nimeksi muotoutui silloin Nokia Siemens Networks. Nokia osti Siemensin osuuden Nokia Siemens Networksistä helmikuussa 2013. (YLE Uutiset www-sivut 2013.)

Nokia Solutions and Networks on kansainvälinen yritys, joka toimii 150 maassa. Nokialla on pitkä historia televerkkoratkaisuissa ja tuotekehityksessä. Yritys on tunnettu laadustaan, tehokkuudestaan ja luotettavuudestaan. (NSN www-sivut 2014.)

2.2. Opinnäytetyön tavoitteet

Tavoitteisiin kuului selvittää Robot Frameworksin ja Jenkkinsin käyttömahdollisuudet hyväksyntätestauksessa. Testausalueeseen kuului radioverkko-ohjaimen perustoiminallisuuden suorituskyvyntestausta. Työssä keskitytään vertailutestausmenetelmiin.

Testiautomaatiokehyksenä toimi Robot Framework. Tässä ajettiin testitapauksia, joilla testattiin radioverkko-ohjaimen ohjelmistoa. Työn aloitushetkellä tehtiin radioverkko-ohjaimeen käyttöliittymän asennukset ja testaukset alustavasti manuaalisesti. Tämä tarkoitti järjestelmän seuraamista, jotta voitiin olla varmoja, että laite on asentunut ja käynnistynyt oikein. Nämä olivat ensimmäisiä asioita, jotka oli tarkoitus korvata automaattisilla testaustyökaluilla.

Automaatiotyökaluksi valittiin jatkuvaan integraatioon käytetty Jenkins sovellus. Tämän käyttö aloitettiin, kun testiympäristö oli rakennettu ja ensimmäiset Robot Framework testausohjelmat oli ohjelmoitu ja todettu toimiviksi.

Opinnäytetyötä toteutettiin projektina, joka oli vain pieni osa kokonaiskuvaa. Työtä kehitettiin niin, että sitä voitaisiin jatkaa myöhemmin, kun opinnäytetyön osuus loppuu, mikäli tarve sitä vaatii.

3. YLEISTÄ TESTAUKSESTA

Testaus on virheidenetsintää ohjelmistosta ja niiden korjaamista. Tällä päästään siihen, että kehitettävän ohjelmiston laatu nousee sekä toimintavarmuus paranee. Virhe on tila tai tapahtuma ohjelmassa, joka poikkeaa sen oletetusta toiminnasta. Nämä esiintyvät yleensä testaajalle virheellisinä lukuarvoina tai virheviesteinä, joita on ohjelmoitu ohjelmistokoodiin.

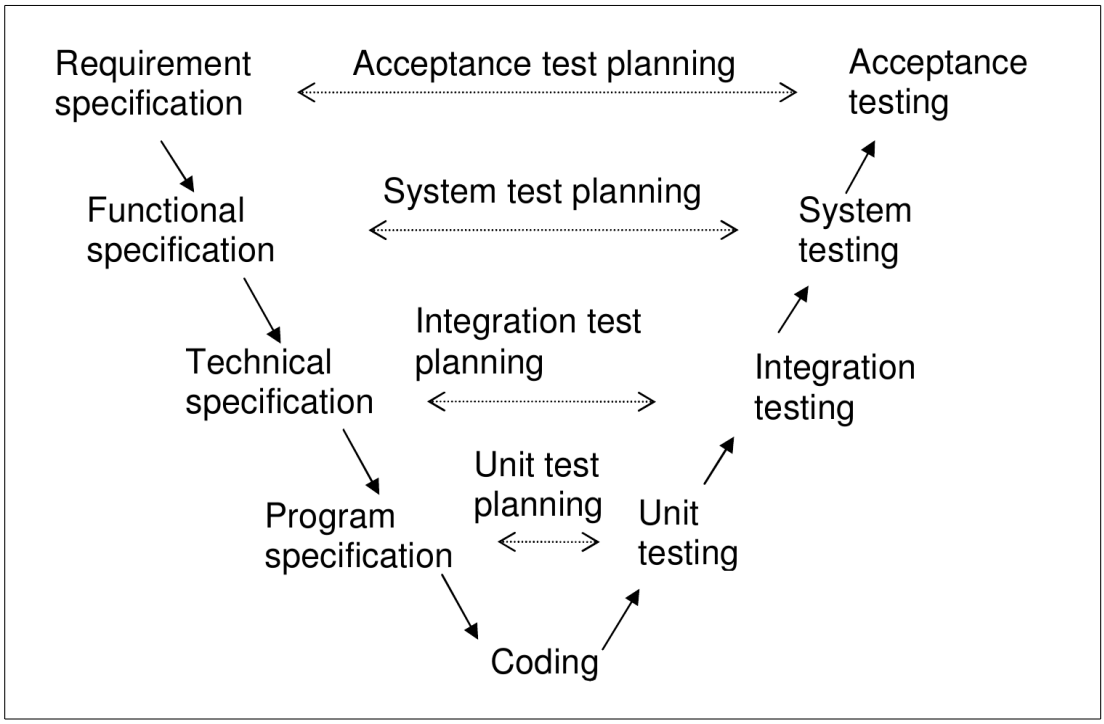
Testauksen voi jakaa neljä vaiheeseen: suunnittelu, testiympäristön luonti, testin suorittaminen ja tulosten tarkastelu. Suunnittelussa arvioidaan testauksen laajuus ja kuinka niitä testataan. Testiympäristön luonnissa luodaan testiympäristö, joka voi olla fyysisesti olemassa tai virtuaalisesti identtinen järjestelmä, jossa testaukset pystytään suorittamaan. Testiympäristössä suoritetaan testaus ja lopuksi tarkastellaan lopputulosta (Haikala & Märijärvi 2006, 289-290.)

Glenford Myers toteaa testauksesta seuraavaa: ”*Testing is the process of executing a program with the intent of finding errors. Testing is the process of establishing confidence that a program does what it supposed to do*”. Ajatuksena on siis yrittää kaikin tavoin löytää tapoja rikkoa ohjelma sekä luoda luottamus, että ohjelma tekee mitä sen kuuluu tehdä. (Myers 2004.)

3.1. Testauksen V-malli ja tasot

V-mallia eli vesiputousmalli on tänä päivänä yksi testausmallien peruskivistä. V-mallin tasot toimivat pareina, jolloin ohjelmiston suunnitteluvaiheessa määritellään myös vaiheeseen tarvittavat testaukset. Tämä tarkoittaa sitä, että vaatimusanalyysin suunnittelussa otetaan myös huomioon hyväksyntätestaus, toiminnallisen määrittelyn suunnittelussa järjestelmätestaus,

teknisenmäärittelyn suunnittelussa integrointitestaus ja moduulisuunnittelun suunnittelussa moduulitestaus. Kuviossa 1 näkyvät kaikki vesiputousmallin tasot. (Watkins 2001, 39-41.)



Kuvio 1. V-malli ohjelmistosuunnitteluun (Watkins. 2001, 40).

3.2. Staattinen ja dynaaminen analyysi

Staattisella analyysillä tarkoitetaan vikojen etsintää ohjelmasta ilman sen suorittamista. Suorittaminen voi tapahtua joko ohjelmallisesti tai perinteisesti ohjelmakoodia selaamalla. Analyysin avulla voidaan löytää virheitä esimerkiksi muuttujien tai osoittimien väärinkäytöstä. (Gerndt 2002, 8.)

Dynaaminen analyysi suoritetaan silloin, kun ohjelma on käynnissä ja ohjelman palauttamia arvoja tarkkaillaan. Dynaamisen analyysin suorittamiseen tarvitaan toimiva ohjelma ja ympäristö, missä nämä testit pystytään ajamaan. Tässä vaiheessa suoritetaan sellaisia ohjelman ominaisuuksia, joiden testaamiseen staattinen analyysi ei sovellu tai olisi työlästä tai jopa mahdotonta. (Gerndt 2002, 11.)

4. AUTOMAATIOTESTAUS

Ohjelmistot ovat nykyisin niin laajoja kokonaisuuksia, että kaiken kattava täydellinen testaus on käytännössä mahdotonta saavuttaa. Ohjelmistoprojektin budjetista 25 - 50 prosenttia kuluu testaukseen ja siihen sisältyy yleensä sekä testaustyökalujen kehittäjät, että manuaalista ja automaattista testausta tekevät testaajat. (Lin ja Wun 2004, 2 – 9.)

Testauksen kattavuuden kasvattamiseksi manuaalitestausta on automatisoitu, jolla voidaan testata suurempia ja pidempiä aikoja, jolloin ihmisten ei tarvitse olla paikalla. Testausta ei ole kuitenkaan täysin mahdollista suorittaa automaattisesti. Testauksen suunnittelu, automatisoinnin ohjelmointi sekä testaustuloksien analysointi on aina pakko tehdä ihmisvoimin.

Testauksen automaatiolla parannetaan tarkkuutta. Tunnollisinkin testaaja voi joskus tehdä virheitä toistuvassa manuaalisessa rutiinitestauksessa. Automatisoimalla vältytään tältä. Automatisointi suorittaa joka kerta testit poikkeuksetta samalla tavalla. Jos testit tulisi toistaa joka kerta kun järjestelmään tulee muutoksia, testien manuaalisen rutiinitestauksen sijaan automatisoinnilla voidaan säästää aikaa ja rahaa.

Automaattitestaus ei kuitenkaan ole täysin aukotonta vaan siinäkin on omat haittapuolensa. Seuraavat asiat on hyvä ottaa huomioon automaatiotestausta suunnitellessa.

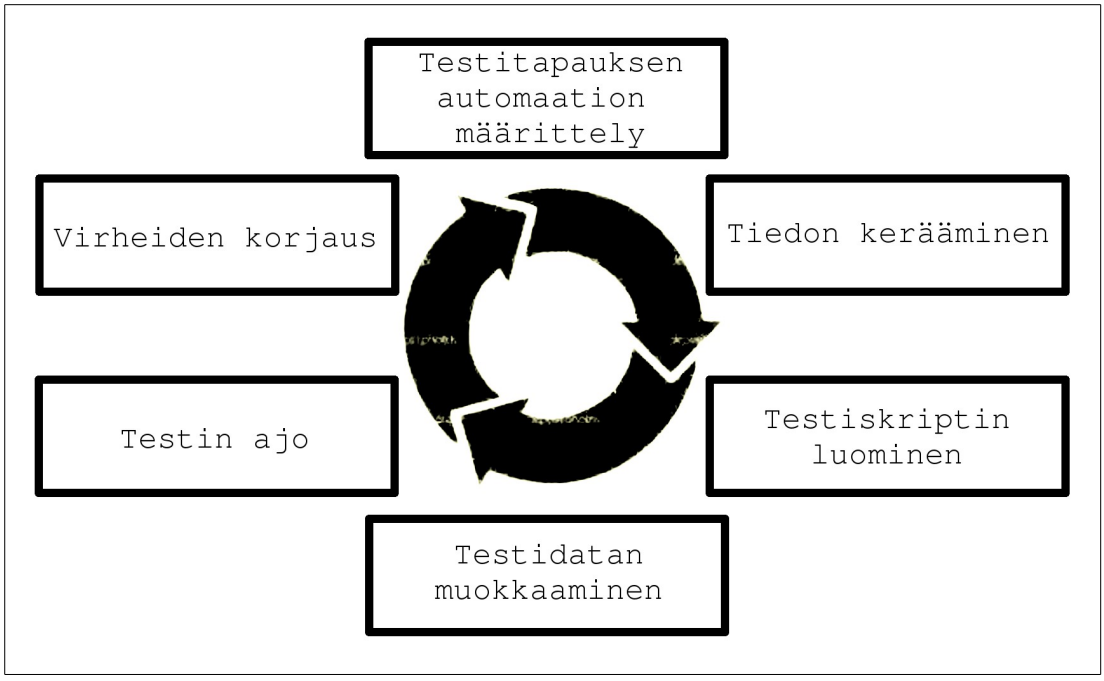
- Testausskriptit vaativat usein uudelleen tarkkailua, sillä pienetkin ohjelmistojen ominaisuuksien muutokset saattavat aiheuttaa ongelmia automatisoitujen testitapausten ajossa.
- Mikään automatisointityökalu ei voi kokonaan suorittaa itsenäisesti testausta, vaan testaajaan on pakko tehdä osa työstä manuaalisesti.
- Automatisointityökalut eivät saata olla yhteensopivia testattavan kohteen kanssa.

- Takaisinmallinnusprosessi on toteutettu erillään testiskriptien ohjelmoinnista, jolloin virhetilanteiden syyt saattavat jäädä selvittämättä.

4.1. Testitapauksien automatisointi

Testaus tulisi aloittaa määrittelemällä testitapausryhmä, jota on tarkoitus suorittaa kokonaan testiajon aikana. Testitapauksella tarkoitetaan yhtä tiettyä tilannetta tai ehtoa testattavassa järjestelmässä. Testitapauksen on tarkoitus kuvata, mitä testataan ja kuinka testataan. Lisäksi testitapaus ohjaa testaajaa tai testausvälinettä testauksen aikana. Testitapauksen ohjaus sisältää ohjeita testin alustamiseen ja suorittamiseen, sekä mitä tarkkailla ja kuinka arvioida lopputulos.

Testitapausryhmän automatisoinnin aikana testaaja käy kaikki ryhmän testitapaukset läpi yksitellen seuraten automatisoidun testauksen kuutta askelta, jotka näkyvät kuviossa 2.



Kuvio 2. Automatisoidun testauksen askeleet Li ja Wu (2004).

4.2. Käytettävyyden arviointi

Käytettävyydellä tarkoitetaan, kuinka miellyttävää, helppoa ja luotettavaa on käyttää ohjelmistoa. Käytettävyydellä haetaan usein osatekijöitä, jotka ovat, opittavuus tehokkuus, muistettavuus, virheettömyys sekä tyydyttävyys. (Nielsen, 1993, 26—27.)

Opinnäytetyössä tutkittiin Robot Frameworkin ja Jenkinsin käytettävyyttä automatisoinnin työkaluna. Automatisoinnin tulisi olla laajennettavissa ja ennen kaikkea joustava käyttäjän tehtäviin. Automatisoinnilla tulisi saada etuja rutiinitestauksiin, joita suoritetaan useasti. Tämän kaltaiset testaukset ovat usein testejä, joiden tarkoitus on selvittää, ovatko kaikki vanhat asiat vielä toiminnassa ajettujen muutosten jälkeen. Nämä eivät aina ole ensisijaisia testikohteita mikäli muutosten yhteydessä ei tapahdu mitään suuria muutoksia. On kuitenkin tärkeää ajaa laajoja järjestelmätestauksia säännöllisesti, koska pienistäkin vioista voi huomaamatta tulla isoja ongelmia.

Testausprojektiin liittyy kiinteästi testin suunnittelu ja testiin liittyvä materiaalivirtojen ohjailu eli testiaineiston hallinta. Testiaineiston hallinta hoidettiin Jenkins sovelluksella. Testauksen automatisoinnin kannalta on kriittistä että kaikki tieto tallentuu mahdollisia ongelmatilanteita varten. Automaattisen testauksen rakentamiseen kuuluu tietokannan hallintaa. Kun automatisoitua testiä rakennetaan, tietojen pitää olla nopeasti saatavilla. Testin ajoympäristö ja testin käyttöympäristö on erotettava toisistaan. Näin voidaan olla varmoja, että testin tiedot tallentuvat varmaan paikkaan, johon on pääsy myöhemmin. Testattavassa laitteessa voi esiintyä ongelmia, jolloin testauksesta saatu informaatio voi kadota laitteen mukana.

4.3. Uudelleenkäytettävyys

Testien uudelleen soveltuvuutta selvitettiin mahdollisiin uusiin testiympäristöihin. Uudelleenkäytössä vanhaa olemassa olevaa ohjelmistoympäristöä käytettiin pohjana tuleville testauksille. Uusille testeille pyrittiin löytämään mahdollisia yhtäläisyyksiä ja käytettiin muokattuja olemassa olevia testitapauksia.

Uudelleenkäytöllä saadaan pienennettyä testien toteuttamiseen tarvittavaa aikaa. (Eronen 2010, 7.)

4.4. Hyväksyntätestaus

Hyväksyntätestaukseen sijoittuvassa suorituskykytestauksen liittyvissä järjestelmätestauksessa ja integraatiotestauksessa yritettiin ajaa testitapaukset loppukäyttäjää vastaavassa ympäristössä. Tällöin voitiin olla varmoja, että pullonkauloja ei ollut sivullisista järjestelmistä, jotka rajoittaisivat suorituskykyä. Tällöin saatiin esiin vikatilanteet, jotka muuten ilmenisivät vasta tuotteen ollessa loppukäyttäjällä. (Black. 2009. 38.)

Isojen testitapaus määrien eli testijoukkojen hallintaan käytettiin tietokonetta johon oli asennettu Jenkins ajamaan Robot Frameworkin ajettavia testitapauksia. Kun tuotetta kehitetään ennen uutta julkaisua, ajetaan ohjelmointiympäristössä kaikki vanhat yksikkötestit. Tärkeintä on, että testit voidaan ajaa ajalla, jolloin se ei haittaa muuta työskentelyä. Testitapauksien luonti aloitettiin tekemällä joukko aloitustestejä (Intake Test). Kun aloitustestit on ajettu onnistuneesti, ajetaan savutestijoukko (Smoke Test). Aloitustestien tehtävänä on päätellä, onko komponentti tai järjestelmä valmis tarkempiin tuleviin testeihin. Savustustesti on komponentin tai järjestelmän päätoiminnallisuuden kattava testaus. Päivittäinen testaus kuuluu teollisuus alan käytäntöihin. (Watkins 2001, 82.)

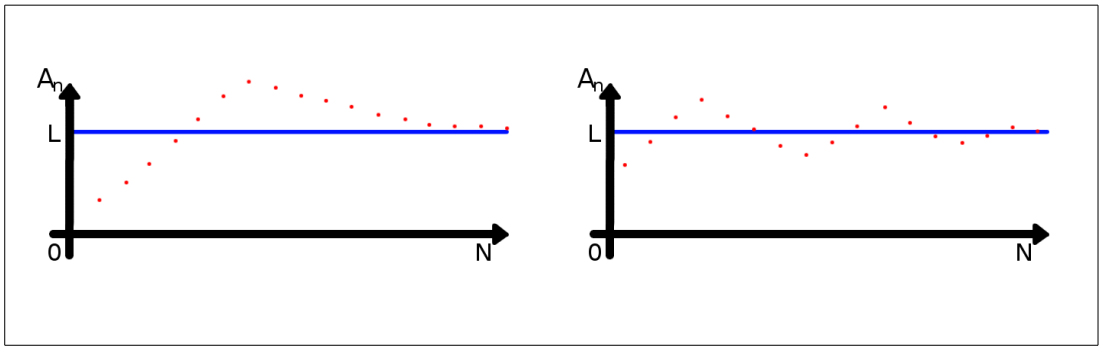
5. VERTAILUTESTAUS

Vertailutesti tai vertailuperiaate on tapa tutkia testisarja-arvojen muutoksia. Vertailutestaus tutkii saatua tulosta määriteltyihin raja-arvoihin tai raja-arvo alueisiin. Käytännössä tämä tarkoittaa sitä, että testitulosta vertaillaan määriteltyihin arvoihin ja tehdään johtopäätökset, onko tulos muuttunut edellisiin tuloksiin verrattuna. Tuloksen muutokseen reagoiminen on tapauskohtaista ja määrittelyt löytyvät testitapauksista.

5.1. Matemaattinen sovellus

Jono on matematiikan kaikkein perustavimpia käsitteitä ja sen avulla kohdataan tilanteet, jossa äärettömyys on osana ongelmaa. Lukualueista muodostetaan luonnollisten lukujen joukko N={1, 2, 3, …}, mikä kuvastaa testikertojen määrää ja sen lukumäärä voi olla ääretön. Jonot ovat myös osa iterointimenetelmiä, joissa ongelmaa ratkaistaan toistuvasti, kuten testauksessa on tapana. Ideaalissa testauksessa iterointien lukumäärän kasvaessa, jonon arvot lähestyvät haettua lopputulosta. (Silvennoinen. 2005.)

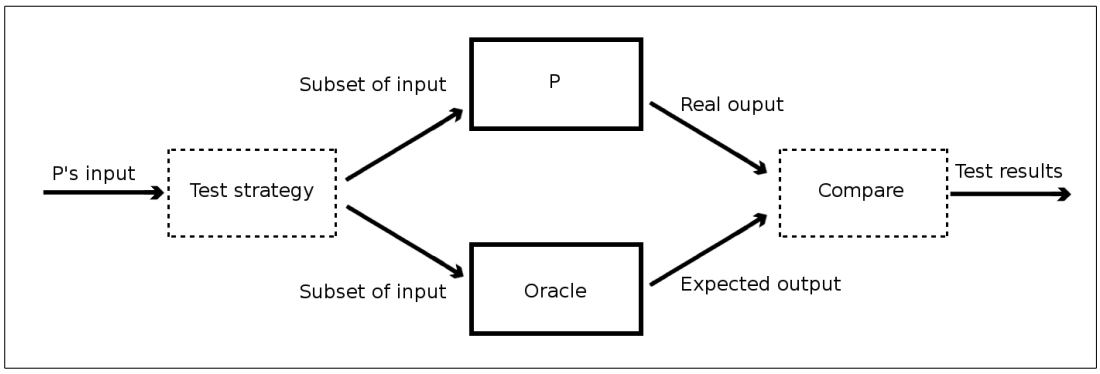
Kuviossa 3 on kaksi kuvitteellisten testitulosten lukujonoa (a_n), jotka etenevät kohti tavoiteltua raja-arvoa L. Vaaka-akseli kuvaa testauskertoja, joissa lukuarvot N kulkevat 1, 2, 3, ... ja pystyakselilla kuvaa ovat vastaavat jonon testiarvot a_n.



Kuvio 3. Kuvitteellinen graafi ideaaliseen tulokseen pääsemisestä.

5.2. Suorituskykyvertailutestaus

Suorituskykytestaukseen kuuluvat suorituskyky, muistinhallinta, luotettavuus, yhteensopivuus, palautuvuus, tietojen konversio, konfiguraatio testaus yms. Testauksessa tämä tarkoittaa komponenttipohjaisten järjestelmien testausta ja vertailutestausmenetelmillä tuotettavien tulosten vertailua.



Kuvio 4. Yksinkertaistettu testausprosessi (Manolache, Kourie, 2002)

Testausprosessin kulku yksinkertaistettuna:

- Testin suunnittelu
- Testin luonti
- Testin ajo
- Testitulosten vertailu
- Testin alkutilaan palautus, myös liittyvät järjestelmät

Tärkein vaihe on odotetun tuloksen vertailu (compare). Tässä vaiheessa käsitellään tieto, jota testi on suorittanut. Arvoja vertailemalla selviää, onko lopputulos kuinka lähellä haluttuja arvoa. Jos arvot heittävät merkittävästi edellisistä tuloksista, on pääteltävissä, että edellisen testin jälkeen tehdyt muutokset ovat aiheuttanut ongelmia. Testaajan tehtävänä on etsiä ongelma ja käyttää vertailutestauksesta saatuja tuloksia apuna.

6. TESTAUKSEN AUTOMATISOINNIN TYÖKALUT

Projektia varten rakennettuun testiympäristöön tehtiin käytettävien työkalujen valintoja. Valinnoissa painotettiin Nokia Solutions and Networkin aiemmin tutuiksi tulleita ohjelmistoja sekä palveluita.

Testauksen automatisointiin ja hallintaan on olemassa monenlaisia ratkaisuja. Automatisointiin käytettäviä ratkaisuja ovat muun muassa testipetigeneraattorit, vertailijat ja testikattavuustyökalut. Robot Framework on itsessään yhdistelmä testipetigeneraattoria ja vertailijaa. Tarvittiin kuitenkin Robot Frameworkin rinnalle hallinnointityökalu valvomaan ja ajastamaan ajettavia testauksia. Tähän työhön valjastimme Jenkins sovelluksen.

Robot Framework:lla luotiin testattavalle radio-ohjaimelle testipedit. Testiä varten kuvattiin eri vaiheet sekä haluttu tulosalue, jolloin tarkastelu oli automatisoitavissa. Järjestelmätestauksessa testitapaukset voitiin nauhoittaa ja käyttää automatisoinnissa.

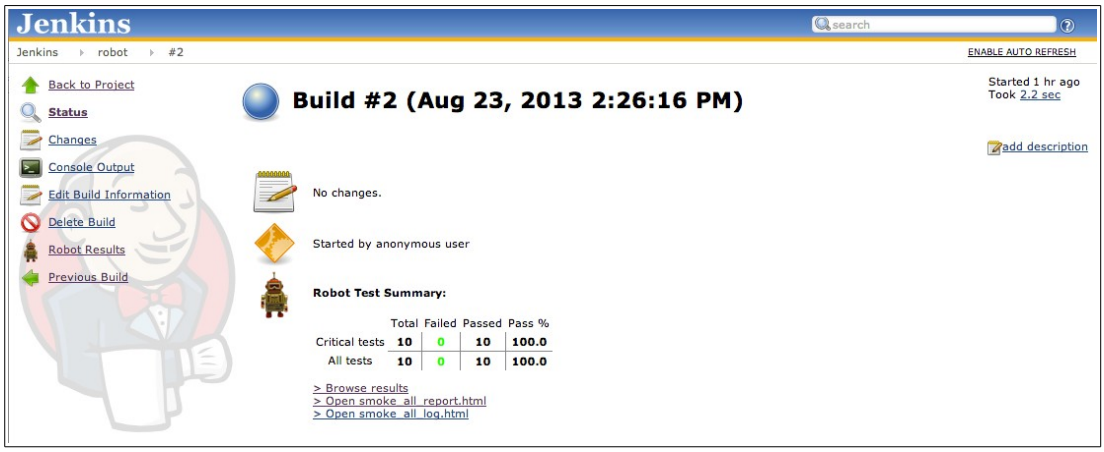
6.1. Robot framework

Robot framework on sovelluskehys, joka on suunniteltu sellaiseksi, että sitä käyttävien henkilöiden on helppo tutkia sillä ajettavia testejä.

Vielä 2000-luvun alussa testaussovelluskehykset piti rakentaa itse, mutta nykyään on olemassa valmiita hyviä testaussovelluskehyksiä. Nokia Solutions and Networksin käytössä on Robot Framework, jota on kehitetty yrityksen toimesta vuodesta 2008. (Robot Framework www-sivut. 2014.)

6.2. Jenkins

Jenkins on sovellus, jota käytetään jatkuvaan integraatioon ohjelmistokehityksessä. Jenkinssillä on tuki moniin ohjelmistojen versiohallintasovelluksiin ja siihen on saatavilla erilaisia laajennuksia. Jenkins tukee myös Robot Frameworkia. Nämä laajennukset mahdollistavat automaattisten testauskokonaisuuksien visuaalisen seuraamisen yhdestä paikasta. Laajennukset tuottavat Jenkins ympäristöön graafeja suoritetuista testeistä ja antavat varoituksia, mikäli testauksessa on tapahtunut ongelmia. Jokaisen testauksen alta löytyy lokitiedostot ja testitulokset. (Jenkins Plug-in www-sivut. 2014.)



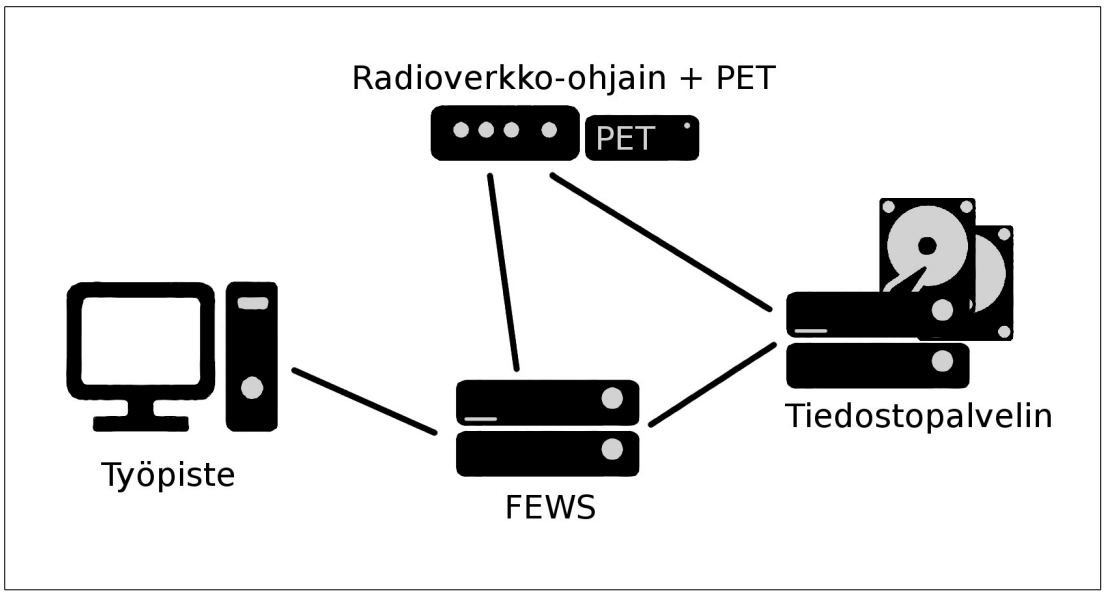
Kuvio 5. Jenkins sovellus Robot framework laajennuksella

6.3. RIDE

RIDE on editori, joka on suunniteltu Robot Frameworkin testien suunnitteluun. Testitapauksien muokkaaminen ja tekeminen onnistuu millä vain tekstieditorilla, mutta RIDE mahdollistaa paremman visuaalisen ympäristön. Testitapauksissa on usein linkitettyjä dokumentointeja tai kirjastoja jotka ovat ymmärrettävästi esillä RIDE editorissa. (RIDE www-sivut. 2014.)

7. TESTAUSYMPÄRISTÖ

Testausympäristö koostui testattavasta mcRNC radioverkko-ohjaimesta, PET:stä (PErformance Tester), FEWS:stä (Field Engineer Work Station) eli järjestelmäasennuksiin tarkoitetusta tietokoneesta, Jenkins tietokoneesta ja tiedostopalvelimesta. Työpistettä lukuun ottamatta, ympäristössä olevat laitteistot olivat olemassa Nokia Solutions and Networkin konesalissa.



Kuvio 6: Kuvaus testausympäristöstä.

7.1. Radioverkko-ohjain

Radioverkko-ohjain toimi testattavana kohteena ja sen päätehtävänä on välittää tietoa tukiaseman sekä runkoverkon välillä. Laite mahdollistaa radioresurssien ohjauksen eri televerkkoihin ja on myös yhteydessä muihin radioverkko-ohjaimiin. Tuotenimi radioverkko-ohjaimelle oli mcRNC.

7.2. PET (PErformance Tester)

Suorituskykytestaukseen käytettiin PET-emulaatioita. PET-testeri emuloi asetuksia ja verkko-operaatioita koko radioverkko-ohjaimen ympärillä. Tämä laite mahdollisti radio-ohjaimen tukiasema-alijärjestelmien ohjaamisen erilaisissa televerkko rasitustesteissä. (Tanskanen. 2010.)

7.3. FEWS (Field Engineer Work Station)

FEWS oli asennuksiin käytettävä tietokone, jolla ajettiin ohjatut testitapaukset radioverkko-ohjaimelle. FEWS:llä oli oikeudet toimia radioverkko-ohjaimen ja tiedostopalvelimen kanssa.

Tähän koneeseen oli asennettu Jenkins, joka hoiti Robot Frameworkin testitapauksien ajojen välityksen ja testausrutiinien ajoituksen. Robot Frameworkin käyttämät testitapaukset sijaitsivat joko paikallisesti tai tiedostopalvelimella. Tämä yksikkö välitti käskyjä SSH-tunnelin välityksellä tiedostopalvelimelle ja radioverkko-ohjaimelle. Robot Framework ajoi annetut testitapaukset ja palautti testitulokset takaisin Jenkinsin käsiteltäväksi. Jenkins arkistoi ajetut testitapaukset ja niiden tulokset.

7.4. Tiedostopalvelin

Tiedostopalvelimella säilytettiin asennuspakettien eri versioita, joita käytettiin testauksessa. Jenkins seurasi tiedostopalvelimelta uusien asennuspakettien saapumista ja aloitti verkko-ohjaimen päivityksen, mikäli muita prosesseja ei ollut käynnissä Jenkinsin toimesta.

7.5. Työpiste

Työpistetietokoneelta seurattiin testituloksia Jenkins sovelluksella. Jenkins seuraustyökalu avattiin Internet-selaimessa, jossa työntekijällä oli mahdollisuus seurata tai muokata ajastettuja testitapauskokonaisuuksia. Työpisteitä voi olla useita ja jokaisella käyttäjällä on mahdollista olla omat tunnukset joihin on määritelty erilaisia toiminnallisuuksia (SaLiux Tech Blog www-sivut 2013). Tätä ei kuitenkaan nähty tarpeelliseksi projektin pienen käyttäjäkunnan vuoksi.

8. TESTITAPAUKSIEN TOTEUTUS

Testiautomaatiota toteuttaessa kannattaa käyttää avointa ohjelmointia (open coding), jossa toistettavaa ohjelmakoodia ei siirretä ohjelmointikirjastoon vaan pidetään paikallaan, kuten normaalissa ohjelmointiprosessissa. Näin ollen sovelluskehyksen testitapauksien koodi on tärkeä pitää yksinkertaisena. (Kaner, Bach, Pettichord, 2002. 109-124.)

Testauskirjastot ohjelmoitiin Python kielellä. Testitapaukset tehtiin Robot Framework omalla taulukkotestitaparakenteella. (Robot Framework www-sivut 2014.)

8.1. Testitapauksien muotoilu

Robot Framework testitapauksien luonnin tavoitteena oli luoda helposti luettavia ja muokattavia testitapauksia, joita myös testausryhmän ulkopuoliset ihmiset pystyvät tulkitsemaan. Robot Frameworkissa testitapaukset voidaan muotoilla muutamilla eri tavoilla, jotka eivät vaikuta testitapauksien ajamiseen millään tavalla. Muotoilu tosin tekee testitapauksista helppolukuisemmaksi aina siinä suhteessa, kuka tiedostoa käsittelee. Havainnollistamisesimerkeissä oli käytetty Robot Framework kehitystiimin luomaa demoa. (Robot Framework demo www-sivut. 2014.)

Yksittäinen testitapaus suoritetaan kutsumalla Robot Framework Pybot ohjelmalla. Testitapaukseen oli kirjoitettu kuvaus mitä ja millä testataan. Esimerkissä testattavana kohteena toimii Pythonilla ohjelmoitu laskin Calculator.py (Liite1). Testauskirjastoon CalculatorLibrary.py (Liite2) oli Pythonilla ohjelmoitu funktioita, jotka käytti laskinta. Testitapausesimerkkejä oli kolme ja jokainen niistä oli toteutettu erilaisella muotoilutavalla.

8.1.1. Avainsanoihin perustuva testitapausmuotoilu

Keyword-driven eli avainsanoihin perustuvissa testitapauksissa (Liite 3) muotoilu oli toteutettu siten, että avainsanat löytyivät esitellyistä testauskirjastoista. Tätä testitapaus on yleisesti käytetty tavallisissa testauksissa.

8.1.2. Arvoon perustuva testitapausmuotoilu

Data-driven eli arvoon perustuvassa testitapauksessa (Liite 4) muotoilu toimii samankaltaisesti kuin avainsanoihin perustuvissa testitapauksissa. Tässäkin testitapauksessa käytetään testauskirjastoa. Erona kuitenkin on, että testitapaukseen on luotu sisäisiä avainsanoja testitapauksen omaan käyttöön. Tämän kaltainen menettely toimii hyvin, jos testauksessa on alueita, joita pitää toistaa useita kertoja.

8.1.3. Gherkin testitapausmuotoilu

Gherkin testitapaus (Liite 5) muistuttaa paljon avainsanoihin perustuvaa testitapausta. Erona on, että käytettävien avainsanojen toiminta on kuvailtu testitapauksen sisälle paljon tarkemmin ja arvot on esitelty testitapauksen nimissä. Tätä testitapausmuotoilua käytetään, jos testauksen pitää olla ymmärrettävässä muodossa myös testaustiimin ulkopuolella.

8.2. Vertailutestauksen luonti

Vertailutestaus on usein iso osa savustustestausta. Arvoja vertailemalla saadaan helposti selville, onko testattava ohjelmisto pysynyt annetuissa normeissa. (Turnquist 2011. 276.)

Testitapaukset luodaan seuraamaan ajon vievää aikaa, suorituskykyä tai vaikka laitteen lämpötiloja. Projektissa testattiin asennuksessa vievää aikaa. Testitapauksen ajon jälkeen testattiin laitteen uudelleenkäynnistysaikoja. Testitapauksien muotoiluna käytettiin avainsanoihin perustuvaa muotoilua.

8.3. Projektin kulku

Testiympäristö rakennettiin vastaamaan suunniteltua testiympäristöä. FEWS oli valmiiksi asennettu Jenkinsiä ja Robot Frameworkia lukuun ottamatta. Jenkins ja Robot Framwork asenettiin FEWS:iin ensimmäisenä. Tämän jälkeen tehtiin testitapaus, joka seurasi verkko-ohjaimen ohjelmiston asennusta, jota alustavasti ajettiin manuaalisesti.

Robot Frameworkin ja Jenkinsin toimivuutta kokeiltiin ensin ohjelmistojen valmiiksi luoduilla demoilla, jotta saatiin parempi käsitys siitä, kuinka testitapauksien ajaminen tapahtui. Tämä auttoi myös testitapauksien ohjelmoinnissa.

Koetestitapauksien toimivuuden tarkastelun jälkeen luotiin testitapauksia seuraamaan laitteen toimintaa. Testitapaukset ajettiin ennen asennusta, asennuksen suorituksen jälkeen ja laitteen uudelleen käynnistyksen jälkeen. Samalla luotiin vertailutestitapauksia testaamaan edellä mainittujen suoritusten ajan käyttöä.

Kun testejä oli ajettu manuaalisesti ja testitapauksien toimivuudesta oltiin varmoja, asetettiin testitapaukset ajettavaksi Jenkins sovelluksessa kerran päivässä. Ajon jälkeen Jenkinsin käyttöä jatkettiin ja testitapauksia laajennettiin. Uusia testitapauksia luotiin aina tarvittaessa.

9. YHTEENVETO

Testaus on tärkeä osa tuotekehitystä isoissa organisaatiossa. Nokia Solutions and Networksin tavoitteena oli kehittää tarpeiden ja vaatimusten mukainen, virheetön ja toimiva tuote. Tuotteessa ajettavan ohjelman oikean toiminnan varmistaminen vaatii testausta. Testausprosessin kehittäminen on myös tärkeä osa testausta, jota tämä opinnäytetyö käsitteli.

Opinnäytetyön päätavoitteena oli etsiä uusia käyttökohteita Robot Frameworksiin ja Jenkinsiin. Käytännössä tämä tarkoitti sitä, että nämä sovellukset liitettiin osaksi hyväksyntätestausta, jonka jälkeen tarkasteltiin niiden käytettävyyttä automatisoidussa vertailutestauksessa.

Olemassa oleva testauslaitteisto asenettiin vastaamaan suunniteltua testausympäristöä. Jenkinsin ja Robot Frameworkin käyttöönotto testauksen automatisoinnin työkaluna sujui vaivattomasti. Ympäristöt testien kehittämiselle ja testien suorittamiselle olivat valmiina yllättävän nopeasti.

Uudelleenkäytön tutkinta jäi pintapuoliseksi. Luotuja testitapauksia pystyttiin käyttämään eri testien yhteydessä. Testikirjastoilla pystyttiin luomaan suurempia kokonaisuuksia, joita voitiin käyttää mahdollisesti tulevissa testiprojekteissa.

Loppuen lopuksi työn suoritukseen sisältyi kaksi täysin uutta testauskirjastoa ja kolme niitä käyttävää Robot Framework testitapausta. Työn loppuessa testiympäristön kehitystä jatkettiin testitapauksia luomalla Nokia Solutions and Networkin omasta toimesta.

LÄHTEET

Black, R. 2009. Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional. Indianapolis, Indiana: Wiley Publishing, Inc.

Eronen, H. 2010. Uudelleenkäytettävyys eräässä ohjelmisto projektissa. Viitattu: 1.2.2014. http://www.doria.fi/bitstream/handle/10024/69179/nbnfi-fe201103181363.pdf?sequence=3

Gerndt, M. 2002. Performance-Oriented Application Development for Distributed Architectures. Amsterdam, NLD: IOS Press.

Haikala, I. & Märijärvi, J. 2006. Ohjelmistotuotanto. Helsinki: Talentum.

Jenkins Plug-in www-sivut. 2014. Viitattu: 28.1.2014. https://wiki.jenkins-ci.org/display/JENKINS/Robot+Framework+Plugin

Kaner, C., Bach, J. & Pettichord, B. 2002. Lessons Learned in Software Testing. Hoboken (NJ) : John Wiley & Sons.

Laitila, E. 2011. Päättelytekniikka ohjelmisto-ongelmien selvittämiseen. Jyväskylä: Uusi IT

Li, K. & Wu, M. 2004. Effective Software Test Automation: Developing an Automated Software Testing Tool. Alameda: SYBEX Inc.

Manolache, L. I. & Kourie, D. G. 2001. Software testing using model programs Software - Practice and experience. Viitattu: 12.2.2014. http://www.ecs.csun.edu/~rlingard/COMP595VAV/ManolachePaper.pdf

Myers, G., Sandler, C., Badgett, T. & Thomas, T. 2004. The Art of Software Testing. Second edition. Hoboken (NJ) : John Wiley

Nielsen, Jakob, 1993. Usability Engineering, San Francisco (CA), Academic Press.

Nokia Solutions and Networks www-sivut. 2014.
Viitattu: 12.1.2014. http://nsn.com/about-us/company

RIDE editor www-sivut. 2014.
Viitattu: 2.2.2014. https://github.com/robotframework/RIDE/

Robot Framework www-sivut. 2014.
Viitattu: 23.2.2014. http://robotframework.org/

Robot Framework demo www-sivut. 2014.
Viitattu: 22.2.2014. https://bitbucket.org/robotframework/robotdemo/wiki/

SaLiux Tech Blog www-sivut. 2013. Viitattu:1.2.2014.
https://saliux.wordpress.com/2013/06/26/manage-jenkins-global-security/

Silvennoinen, R. 2005. Laaja matematiikka 2. Viitattu: 20.2.2014.
http://matriisi.ee.tut.fi/courses/7303045/1_Lukusarjat.pdf

Tanskanen, J. 2010. Regression Testing Improvements. Viitattu: 9.2.2014.
https://noppa.aalto.fi/noppa/kurssi/s-38.3310/harjoitustyot/S-38_3310_jukka_tanskanen.pdf

Turnquist, G. L. Python Testing Cookbook. Olton Birmingham: Packt Publishing Ltd.

Watkins, J. 2001 Testing IT : An Off-the-Shelf Software Testing Handbook. Port Chester, NY, USA: Cambridge University Press

YLE Uutiset www-sivut. 2013. Viitattu: 13.1.2014.
http://yle.fi/uutiset/nokia_ostaa_siemensin_osuuden_nokia_siemens_networksista/6711881

LIITTEET

LIITE 1

Calculator.py

```
class Calculator(object):
    BUTTONS = '1234567890+-*/C='
    def __init__(self):
        self._expression = ''

    def push(self, button):
        if button not in self.BUTTONS:
            raise CalculationError("Invalid button '%s'." % button)
        if button == '=':
            self._expression = self._calculate(self._expression)
        elif button == 'C':
            self._expression = ''
        else:
            self._expression += button
        return self._expression

    def _calculate(self, expression):
        try:
            return str(eval(expression))
        except SyntaxError:
            raise CalculationError('Invalid expression.')
        except ZeroDivisionError:
            raise CalculationError('Division by zero.')


class CalculationError(Exception):
  pass
```

LIITE 2 (1/2)

CalculatorLibrary.py

```
from calculator import Calculator, CalculationError


class CalculatorLibrary(object):
    """Test library for testing Calculator business logic.

    Interacts with the calculator directly using its `push` method.
    """

    def __init__(self):
        self._calc = Calculator()
        self._result = ''

    def push_button(self, button):
        """Pushes the specified `button`.

        The given value is passed to the calculator directly. Valid
buttons
        are everything that the calculator accepts.

        Examples:
        | Push Button | 1 |
        | Push Button | C |

        Use `Push Buttons` if you need to input longer expressions.
        """
        self._result = self._calc.push(button)

    def push_buttons(self, buttons):
        """Pushes the specified `buttons`.

        Uses `Push Button` to push all the buttons that must be
given as
        a single string. Possible spaces are ignored.

        Example:
        | Push Buttons | 1 + 2 = |
        """
        for button in buttons.replace(' ', ''):
            self.push_button(button)

    def result_should_be(self, expected):
        """Verifies that the current result is `expected`.

        Example:
        | Push Buttons     | 1 + 2 = |
        | Result Should Be | 3       |
        """
        if self._result != expected:
            raise AssertionError('%s != %s' % (self._result,
expected))

    def should_cause_error(self, expression):
        """Verifies that calculating the given `expression` causes
an error.
```

```
        The error message is returned and can be verified using, for
example,
        `Should Be Equal` or other keywords in `BuiltIn` library.
```

LIITE 2 (2/2)

```
         Examples:
        | Should Cause Error | invalid            |
|
        | ${error} =         | Should Cause Error | 1 / 0
|
        | Should Be Equal    | ${error}           | Division by
zero. |
        """
        try:
            self.push_buttons(expression)
        except CalculationError, err:
            return str(err)
        else:
            raise AssertionError("'%s' should caused an error" %
expression)
```

LIITE 3

keyword_driven.txt

```
*** Settings ***
Documentation     Example test cases using the keyword-driven
testing approach.
...
...   All tests contain a workflow constructed from keywords in
...   `CalculatorLibrary`. Creating new tests or editing existing
...   is easy even for people without programming skills.
...
...   This kind of style works well for normal test automation.
...   If also business people need to understand tests, using
...   _gherkin_ style may work better.
Library           CalculatorLibrary

*** Test Cases ***
Push button
    Push button    1
    Result should be    1

Push multiple buttons
    Push button    1
    Push button    2
    Result should be    12

Simple calculation
    Push button    1
    Push button    +
    Push button    2
    Push button    =
    Result should be    3

Longer calculation
    Push buttons    5 + 4 - 3 * 2 / 1 =
    Result should be    3

Clear
    Push button    1
    Push button    C
    Result should be    ${EMPTY}    # ${EMPTY} is a built-in
variable
```

LIITE 4

data_driven.txt

```
*** Settings ***
Documentation     Example test cases using the data-driven testing approach.
...
...               Tests use `Calculate` keyword created in this file, that in
...               turn uses keywords in `CalculatorLibrary`. An exception is
...               the last test that has a custom _template keyword_.
...
...               The data-driven style works well when you need to repeat
...               the same workflow multiple times.
...
...               Notice that one of these tests fails on purpose to show how
...               failures look like.
Test Template     Calculate
Library           CalculatorLibrary

*** Test Cases ***    Expression    Expected
Addition              12 + 2 + 2    16
                      2 + -3        -1

Subtraction           12 - 2 - 2    8
                      2 - -3        5

Multiplication        12 * 2 * 2    48
                      2 * -3        -6

Division              12 / 2 / 2    3
                      2 / -3        -1

Failing               1 + 1         3

Calculation error     [Template]    Calculation should fail
                      kekkonen      Invalid button 'k'.
                      ${EMPTY}      Invalid expression.
                      1 / 0         Division by zero.

*** Keywords ***
Calculate
    [Arguments]    ${expression}    ${expected}
    Push buttons    C${expression}=
    Result should be    ${expected}

Calculation should fail
    [Arguments]    ${expression}    ${expected}
    ${error} =    Should cause error    C${expression}=
    Should be equal    ${expected}    ${error}    # Using `BuiltIn` keyword
```

LIITE 5

gherkin.txt

```
*** Settings ***
Documentation     Example test case using the gherkin syntax.
...
...               This test has a workflow similar to the keyword-driven
...               examples. The difference is that the keywords use higher
...               abstraction level and their arguments are embedded into
...               the keyword names.
...
...               This kind of _gherkin_ syntax has been made popular by
...               [http://cukes.info|Cucumber]. It works well especially when
...               tests act as examples that need to be easily understood also
...               by the business people.
Library           CalculatorLibrary

*** Test Cases ***
Addition
    Given calculator has been cleared
    When user types "1 + 1"
    and user pushes equals
    Then result is "2"

*** Keywords ***
Calculator has been cleared
    Push button    C

User types "${expression}"
    Push buttons    ${expression}

User pushes equals
    Push button    =

Result is "${result}"
    Result should be    ${result}
```