OULUN SEUDUN
AMMATTIKORKEAKOULU

Juhani Riekki

SYSTEEMIKOMPONENTIN JATKUVA INTEGROINTI

SYSTEEMIKOMPONENTIN JATKUVA INTEGROINTI

Juhani Riekki
Opinnäytetyö
Kevät 2013
Tietotekniikan koulutusohjelma
Oulun seudun ammattikorkeakoulu

ALKULAUSE

Opinnäytetyö tehtiin Nokia Siemens Networks Oy:lle. Työn ohjaavana opettajana toimi Veikko Tapaninen ja valvojana Nokia Siemens Networks Oy:n puolelta insinööri Kari Sivonen. Työssä suurena apuna toimi koko ohjelmisto-tiimi, jolle työ tehtiin.

Oulussa 17.5.2013

Juhani Riekki

TIIVISTELMÄ

Oulun seudun ammattikorkeakoulu
Tietotekniikan koulutusohjelma, sulautetut ohjelmistot

Tekijä: Juhani Riekki
Opinnäytetyön nimi: Systeemikomponentin jatkuva integrointi
Työn ohjaaja: Veikko Tapaninen
Työn valmistumislukukausi ja -vuosi: Kevät 2013
Sivumäärä: 34 + 3 liitettä

Tämän opinnäytetyön tavoitteena oli ottaa käyttöön jatkuva integrointi eräälle Nokia Siemens Networks Oy:n tukiasemaohjelmiston systeemikomponentille. Sen avulla voidaan varmistaa, että kehitettävä ohjelmisto on mahdollisimman laadukas ja kattavasti testattu ja sitä voidaan toimittaa eteenpäin ylemmille ohjelmistokerroksille nopeammin kuin manuaalisesti käännettynä ja testattuna.

Systeemikomponentti tarvitsee toimiakseen myös muita systeemikomponentteja niin käännös- ja linkkausvaiheessa kuin testauksessakin. Testauksessa kaikki systeemikomponentit ladataan testattavaan laitteistoon ja suoritetaan systeemikomponenttitason testit.

Työssä paneuduttiin versionhallintatyökaluun, itse ohjelmiston käännökseen, testaukseen, tukiasemalaitteistoihin ja jatkuvan integroinnin työkaluihin. Lisäksi piti tuntea itse ohjelmistonkehitys. Haasteena olivat useat eri tukiasematyypit ja kaksi eri käyttöjärjestelmää, Linux ja kaupallinen käyttöjärjestelmä.

Jatkuvalla integroinnilla kaikki testaus saatiin automatisoitua pieniä poikkeuksia lukuun ottamatta. Testitulokset saivat tämän myötä paremman näkyvyyden ja ohjelmiston tason seuranta parani merkittävästi. Testien suoritusaika lähes puoliintui testausalustojen tuplaamisen johdosta. Tämä mahdollistaa tarvittaessa useamman testatun ohjelmistojulkaisun toimittamisen yhä lyhyemmässä ajassa.

Asiasanat: jatkuva integrointi, Jenkins, ketterät menetelmät

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Software Development

Author: Juhani Riekki
Title of thesis: Continuous Integration of a System Component.
Supervisor: Veikko Tapaninen
Term and year of completion: Spring 2013
Pages: 34 + 3 appendices

The aim of this Bachelor´s thesis was to introduce continuous integration to Nokia Siemens Networks´ base station software's system component. It is used to ensure that the developed software is of high quality and fully tested and it may be delivered to upper software layers faster than built and tested manually.

The system component needs other system components in a linking phase as well as in building and testing phases. In the testing phase all system components are loaded to the testing hardware and system component level tests are executed.

This thesis was focused on version control software, building methods of the software, testing, base station hardware and continuous integration tools. Challenges of this thesis were many different types of base stations and two different operating systems, Linux and a commercial real-time operating system.

With continuous integration, all testing and building were automated excluding minor exceptions. Test results had better visibility and the monitoring of software state improved significantly. The executing time of tests almost halved because of automation and doubling the test environments. Software releases can now be delivered faster and more often than without continuous integration.

Keywords: continuous integration, Jenkins, agile methods

SISÄLLYS

LYHENTEET JA MÄÄRITELMÄT

CI	Continuous Integration, jatkuva integraatio
HTML	HyperText Markup Language, hyperlinkkien kuvauskieli
IP	Internet Protocol
Job	Työyksikkö
JRE	Java Runtime Environment
Noodi	Suoritinyksikkö, tietokone
SCM	Software Configuration Management
SSH	Secure Shell, salattuun tietoliikenteeseen tarkoitettu protokolla
SVN	Subversion, versionhallintajärjestelmä
Trunk	Versionhallinnan tiedostopuu, jonka versiotieto päivittyy aina tiedostojen päivittyessä
UDP	User Datagram Protocol, yhteydetön protokolla
VPN	Virtual Private Network, virtuaalinen, näennäisesti yksityinen verkko
WAR	Web application ARchive
XML	Extensible Markup Language, Kuvauskieli tiedon jäsentämistä varten

1 JOHDANTO

Nokia Siemens Networks Oy on yksi maailman johtavista tietoliikenne-infrastruktuurin laite-, ohjelmisto- ja palvelutoimittajista ja toimii etulinjassa jokaisen sukupolven mobiiliteknologiassa (Nokia Oyj 2013, hakupäivä 22.3.2013; Nokia Siemens Networks 2013, hakupäivä 22.3 2013). Tämän opinnäytetyön tavoitteena on suunnitella ja toteuttaa eräälle Nokia Siemens Networks Oy:n tukiasemaohjelmiston systeemikomponentille ketterien ohjelmistokehitysmenetelmien mukainen jatkuva integrointi.

Työn tavoitteena on toteuttaa täysin automaattinen järjestelmä, joka huomaa ohjelmiston muutokset versionhallinnassa, käynnistää tämän seurauksena ohjelmiston käännöksen ja lopulta testaa sen oikeassa tuotteessa. Aikaisemmin ohjelmisto koottiin ennen integrointia manuaalisesti ja paketoinnista vastaava henkilö toimitti ohjelmistopaketin testaajalle. Testaaja latasi ohjelmiston testilaitteistoon, suoritti testauksen manuaalisesti ja raportoi testitulokset sähköpostin välityksellä. Työn valmistuessa nämä vaiheet jäävät pois ja henkilöt voivat keskittyä ohjelmiston laadun parantamiseen.

Työn merkittävin liikkeellepanija oli testiajan riittämättömyys jatkuvasti kasvavassa ohjelmistossa. Ohjelmisto haarautuu aika ajoin ja nämä ohjelmistohaarat testataan aivan kuten ohjelmiston pääkehityshaarakin. Ajan kuluessa ylläpidettäviä ohjelmistohaaroja, brancheja, on lukuisia ja niiden päivittyessä on ne myös testattava. Testattavia tuotteita on haarasta riippuen kaksi tai useampia ja testiaikaa kuluu noin kaksi tuntia jokaista tuotetta kohden.

2 JATKUVA INTEGROINTI

Jatkuva integrointi on ohjelmistokehityskäytäntö, jossa jokainen tiimin jäsen integroi kirjoittamaansa koodia pääohjelmistoon useasti, vähintään päivittäin, mikä johtaa useisiin integrointeihin päivässä. Jokainen integrointi verifioidaan automaattisella käännöksellä ja testauksella, jotta integroinnista johtuvat virheet huomataan mahdollisimman aikaisin. (Fowler 2006.)

Jatkuva integrointi on osa Extreme Programming -menetelmää (XP), joka puolestaan on yksi ketteristä menetelmistä. Ketterä ohjelmistonkehitys on joukko menetelmiä, joille on yhteistä ohjelmiston pienet julkaisut nopeissa sykleissä, läheinen yhteistyö asiakkaan ja kehittäjän välillä ja kyky tehdä viime hetken muutoksia eli nopea reagointi. Ketteriä menetelmiä ovat XP:n lisäksi muun muassa Scrum ja Feature-driven development. (Abrahamsson, Salo, Ronkainen & Warsta 2002, s.18–27.)

Perinteisesti integrointi ajoittuu tiettyyn ajanhetkeen, jolloin myös ohjelmistotiimit uskovat oman tuotteensa olevan valmis. Useiden kehittäjien ja tiimien muutoksien integrointi ohjelmistotuotteeseen saattaa olla kuitenkin todella ongelmallista ja voi kestää päiviä, jopa kuukausia, ennen kuin tuote on valmis toimitettavaksi eteenpäin. Jatkuvalla integroinnilla pyritään pääsemään tästä ongelmasta eroon pitämällä kaikkien kehittäjien koodi integroituna ja testattuna versionhallinnassa valmiina julkaistavaksi. Tavoitteena on julkaisukelpoinen ohjelmisto milloin tahansa, lukuun ottamatta muutaman edellisen tunnin työskentelyä. (Shore & Warden, 2008, s.183–190.)

2.1 Jatkuvan integroinnin tausta

Tyytymättömänä suunnittelun, määrittelyn ja dokumentoinnin runsaaseen määrään joukko ohjelmistokehittäjiä kehitti 90-luvulla niin sanotut ketterät kehitysmenetelmät (agile methods). Näiden uusien menetelmien ansiosta ohjelmistokehittäjät pystyivät keskittymään itse ohjelmiston kehittämiseen sen

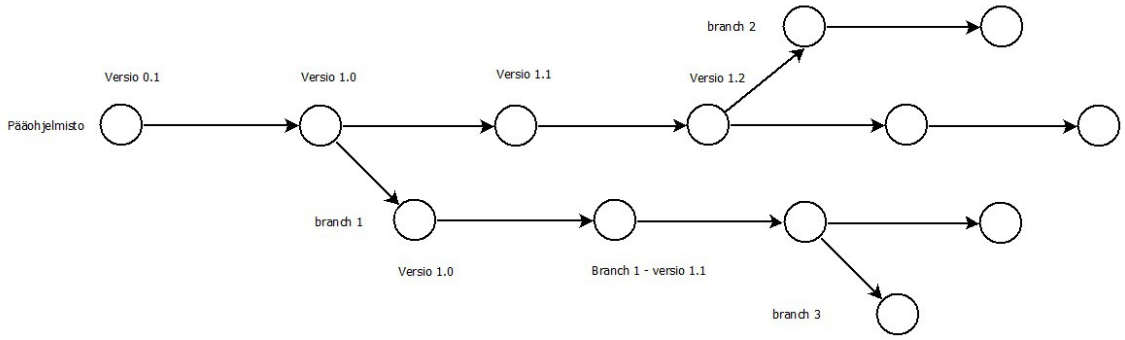
suunnittelun ja dokumentoinnin sijaan. Ketterät menetelmät yleisesti luottavat iteratiiviseen lähestymistapaan ohjelmiston määrittelyssä, kehityksessä ja toimituksessa. Ne on suunniteltu ensisijaisesti tukemaan liiketoiminnan sovellusten kehittämistä, jossa systeemitason vaatimukset muuttuvat äkillisesti ohjelmistokehitysprosessin aikana. (Sommerville 2007, s.396.)

2.2 Jatkuvan integroinnin periaatteet ja käytännöt

Jatkuva integrointi on luonteeltaan samanlainen kuin muutkin ketterien käytäntöjen menetelmät ja yhteistä niille onkin sama päämäärä eli toimiva ohjelmisto ja lyhyet syklit ohjelmistonkehityksessä. Sisällöltään menetelmät poikkeavat kuitenkin toisistaan. Perinteiseen ohjelmistonkehitykseen verrattuna jatkuvalla integroinnilla saavutetaan useita etuja:

- Virheet havaitaan aikaisessa vaiheessa.
- Virheen korjaus ja vian paikantaminen on helpompaa.
- Toimiva ohjelmisto on julkaistavissa lähes milloin tahansa.
- Muutokset ovat paremmin hallittavissa. (Sommerville 2007, s.396–401)

Jotta jatkuva integrointi onnistuisi, on muutamia periaatteita, joita on syytä noudattaa. Yksi yhtenäinen talletuspaikka ohjelmistolle on näistä ehkä oleellisin. Kaikki tiimin jäsenet päivittävät tiedostojaan siis vain yhteen paikkaan. (Fowler 2006.) Ohjelmiston kehitykselle on yleistä, että ohjelmisto haarautuu (branching) jossain vaiheessa, esimerkiksi uudelle tuotteelle koodatun tuen takia tai jos halutaan asiakaskohtainen ylläpitohaara, johon saa viedä vain tietynlaisia päivityksiä ja korjauksia. Ohjelmistohaarojen määrä pitäisi kuitenkin pitää minimissä ylläpidettävyyden takia. Haarautuminen esitellään kuviossa 1.



KUVIO 1. Branching

Automaattinen ohjelmiston kääntäminen on avainehto jatkuvalle integroinnille. Sen avulla varmistetaan, että versionhallinnassa on aina kääntyvä versio ohjelmistosta. Tämä ei takaa sitä, että ohjelmisto olisi toimiva, mutta ainakin sille on paremmat edellytykset. Viallisesta käännöksestä ilmoitetaan automaattisesti edellisen muutoksen tekijälle ja näin ohjelmiston laatu säilyy parempana. (Fowler 2006.)

Hitaat käännökset ovat yleisin ongelma jatkuvassa integroinnissa. Käännös pitäisi pitää mahdollisuuksien mukaan alle 10 minuutin mittaisena. Uusissa projekteissa tämä on helpompi toteuttaa, vanhoihin projekteihin nopean käännöksen käyttöönotto ei välttämättä ole niin helppoa. Synkronisessa jatkuvassa integroinnissa odotetaan tuloksia ennen seuraavan muutoksen viemistä versionhallintaan, ja tätä periaatetta on vaikea noudattaa, jos käännösaika on paljon mainittua suurempi. Asynkronisessa jatkuvassa integroinnissa käännöksen valmistumista ei odoteta, vaan siinä tulokset tulevat käännöksen jälkeen. Asynkroninen jatkuva integrointi on ongelmallisempi, koska siinä tulee viallisia käännöksiä, vaikka uusia muutoksia on jo tehty. (Shore & Warden, 2008, s.183–190.)

Kommunikaatio ja näkyvyys ovat myös avainsanoja jatkuvalle integroinnille. Näkyvyyttä esittää Jenkins-työkalussa web-sivu, jossa on punaisia tai vihreitä ”laava-lamppuja”, joista jokainen näkee kunkin työn tilan ja sen, kuinka kauan työ on siinä tilassa ollut. Vihreällä indikoidaan onnistunutta käännöstä, punaisella epäonnistunutta. Näkymästä voi rakennella yleisen näkymän ja

jokainen käyttäjä voi luoda myös useita henkilökohtaisia näkymiä. Useamman haaran hallinnoimiseen voi tehdä niin kutsutun managerinäkymän, jolle kerätään yksi tieto jokaisesta halutusta käännöksestä. Tämä yksi tieto kuvaa koko prosessin läpimenoa käännöksineen ja testeineen. Testeistä ja käännöksistä on mahdollista kerätä metriikkoja ja esittää ne jatkuvan integroinnin työkaluissa sisäänrakennettujen ominaisuuksien tai liitännäisten kautta. Näin esimerkiksi käännösaikaa ja testien määrää on helppo seurata. Web-sivuilta voi myös seurata käynnissä olevaan käännökseen tehtyjä muutoksia sekä muutoshistoriaa yhdistettynä käännöshistoriaan. (Fowler 2006.)

Testaus jatkuvassa integroinnissa pitää suorittaa täysin samanlaisessa laitteistossa kuin mihin ohjelmistojulkaisu on tarkoitettu. Jos testit ajetaan erilaisessa ympäristössä kuin esimerkiksi asiakkaalle on annettu, ei toimivuutta voida taata ja vikojen toistettavuus voi olla hankalaa. Määrittelyjä ja muita ympäristötekijöitä ei välttämättä ole mahdollista kloonata, mutta ympäristöt on pyrittävä mallintamaan niin tarkasti kuin mahdollista. (Fowler 2006).

Jatkuva integrointi ei poista vikoja, mutta tekee niiden löytämisen paljon helpommaksi. Jos toimivan ohjelmiston päälle viety päivitys ei mene kaikista integroinnin vaiheista läpi, vika on todennäköisimmin viimeisimmässä päivityksessä ja vian etsintä voidaan kohdistaa jopa yhteen tiedostoon. Versionhallintatyökalua apuna käyttäen muutoksesta voi tehdä vertailun, jonka avulla vika todennäköisesti löytyy. (Fowler 2006.)

3 KÄYTETTÄVÄT TYÖKALUT JA YMPÄRISTÖT

3.1 Työkalut

Jatkuva integrointi toteutettiin käyttämällä Jenkins-työkalua (http://jenkins-ci.org), joka on ilmainen Java-pohjainen avoimen lähdekoodin sovellus. Jenkins tarjoaa tuen monelle eri versionhallintatyökalulle ja on monipuolinen ja laajennettavissa oleva sovellus. Jenkins tukee myös liitännäisiä, joita voi ladata Jenkinsin sivulta tai ohjelmoida itse.

Versionhallintatyökalu on Subversion (SVN, http://subversion.apache.org), joka myös on avoimeen lähdekoodiin perustuva työkalu. Subversioniin määritellään pääkehityshaaralle oma kansio, johon tulee hakemistorakenne branches, tags ja trunk. Trunk on paikka, jossa lähdekoodi kansiorakenteineen sijaitsee. Trunk-kansion versiotieto päivittyy aina, kun mikä tahansa tiedosto viedään ohjelmistoon, ja tämän versiotiedon pohjalta Subversionista voi ladata halutun version ohjelmistosta.

Testauksen työkaluna on Nokia Siemens Networksin kehittämä Javalla tehty sovellus JavaTesteri, jota voi käyttää niin konsolista kuin graafisesta käyttöliittymästä. JavaTesterillä suoritetaan xml-tiedostoa (liite 1), johon on koottu lista ajettavista testeistä (liite 2). Tätä kutsutaan testisetiksi. Suoritettuaan kaikki testisetin testit se tuottaa ajosta JUnit XML -muotoisen testiraportin (liite 3).

3.2 Ohjelmistoympäristöt

Tukiasemalaitteistoissa on käytössä teknologiasta riippuen joko kaupallinen reaaliaikakäyttöjärjestelmä tai Linux. Tukiasemaohjelmistot käännetään ja linkataan molemmille käyttöjärjestelmille eri komennoin käyttäen eri työkaluja. Testit ajetaan molemmille ympäristöille.

Testausta varten systeemikomponentilla on oma testikomponentti, joka käännetään ja linkataan varsinaiseen ohjelmistoon. Testikomponentti sisältää

testikoodit ja käännöksiin tarvittavat tiedostot sekä itse testiscriptit. Testikomponentti on ladattavissa versionhallinnasta ja sijaitsee samassa hakemistorakenteessa kuin itse testattava ohjelmistokin. Ohjelmisto varustettuna testikomponentilla ladataan tukiasemaan joko debuggerilla tai testiohjelmalla. Testit suoritetaan JavaTesterillä, jolle kerrotaan, mikä testisetti pitää millekin laitteistolle ajaa.

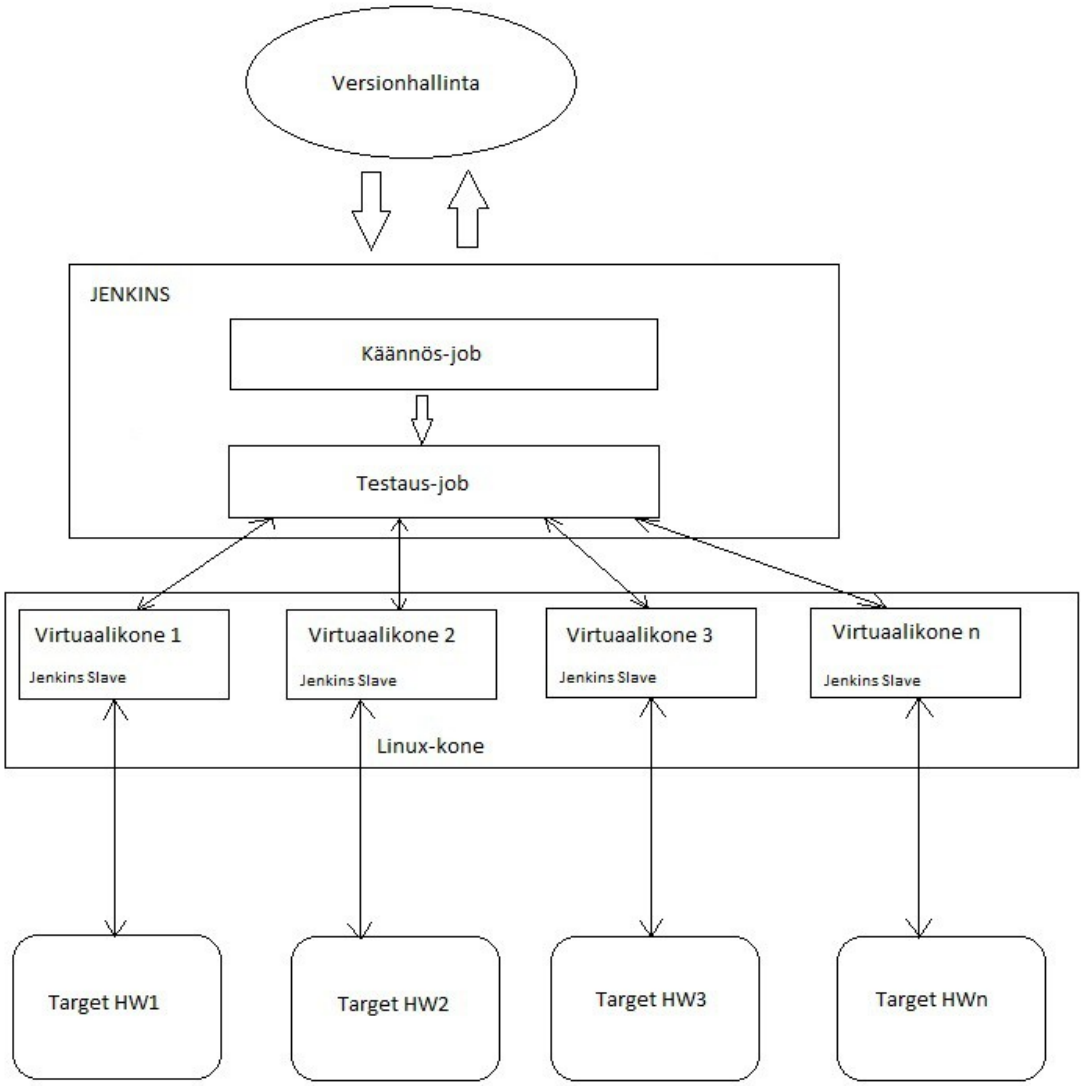
3.3 Laitteistot

Testattavat laitteistot ovat Nokia Siemens Networks Oy:n kehittämiä tukiasemalaitteistoja, jotka sijaitsevat NSN:n tiloissa. Yhteen testattavaan laitteistoon kuuluu yleensä kaksi piirikorttia, joissa on oma suoritin, muisti ja muita piirejä. Kortit viestivät keskenään verkon yli ja myös testit suoritetaan verkon välityksellä.

Laitteet on koottu telineisiin testauslaboratorioon, jonne jatkuvaa integrointia suorittavalta koneelta on virtuaalikoneiden kautta yhteys. Aikaisemmassa vaiheessa yhteyteen käytettiin VPN-yhteyttä virtuaalikoneelta testilaitteistoihin VPN-laitteiden kautta, mutta siitä luovuttiin kuorman ja laitteistojen määrän kasvaessa.

Jokaiselle testattavalle laitteistolle on oma testisettinsä, koska laitteistot poikkeavat oleellisesti toisistaan niin komponenttien kuin ohjelmistonkin osalta. Yhden geneerisen testisetin tekeminen on näin ollen lähes mahdotonta. Testisetit erotetaan toisistaan kuvaavalla nimellä, jotta väärää testisettiä ei ajeta vahingossa väärässä laitteistossa.

Testilaitteistot, Jenkins-kone ja versionhallinta ovat yhteydessä toisiinsa verkon välityksellä. Kokoonpano esitellään kuviossa 2.

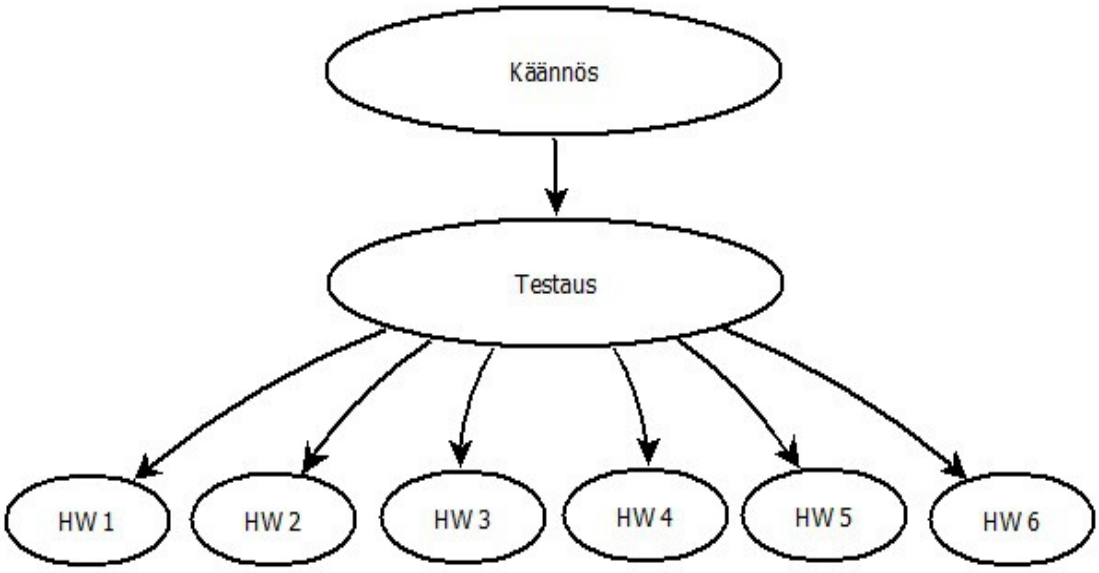


KUVIO 2. Laitteiston kokoonpano

4 TOTEUTUS

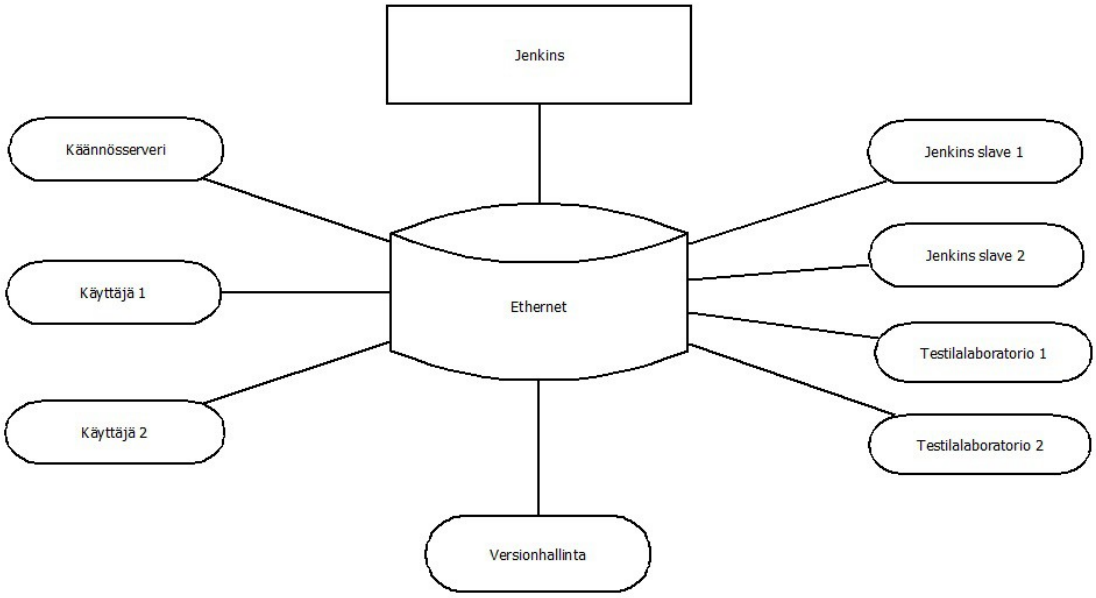
Jatkuvan integroinnin prosessi toteutettiin tässä työssä Jenkins-ohjelmistolla. Jenkinsiin määritellään tarvittava määrä työyksiköitä, jobeja, joita yhdistämällä saadaan luotua toimiva jatkuvan integraation prosessi. Työyksiköiden määrää ei ole rajattu mitenkään, mikä mahdollistaa myös massiivisten ohjelmistojen jatkuvan integroinnin tässä työssä kuvatulla tavalla. Useamman käyttöjärjestelmän aiheuttamia haasteita voidaan hallita jatkuvassa integroinnissa määrittelemällä käyttöjärjestelmäkohtaiset käännökset ja testaus ja yhdistämällä niiden tulokset lopussa.

Ohjelmiston automaattinen käännös ja linkkaus toteutetaan käyttämällä olemassa olevia käännösskriptejä käännös-jobissa. Tämä määritellään siten, että muutos versionhallinnassa aiheuttaa jobin ajamisen ja onnistunut käännös käynnistää toisen jobin, tässä tapauksessa testaus-jobin. Testaus-jobi suorittaa testit testilaitteistossa. Testiajon jälkeen tulokset ovat nähtävillä testiajon suorittaneen jobin sivulla. Yksinkertainen toimintakaavio esitelty on kuviossa 3, jossa on kuvattu työvaiheiden kytkentää toisiinsa.



KUVIO 3. Jenkinsin toimintakaavio

Jatkuvan integroinnin prosessi on tehtävä siten, että käyttäjällä ja laitteistolla ei ole pakko olla ennalta määrättyä sijaintia (kuvio 4). Käyttäjän on voitava tehdä töitä tarvittaessa myös kotoa ja yrityksen toisista toimipisteistä. Tämä tuo työhön joustavuutta ja kiireellisessä tilanteessa vastuuhenkilön ei ole pakko olla aina yrityksen toimipisteessä ratkaisemassa ongelmia.

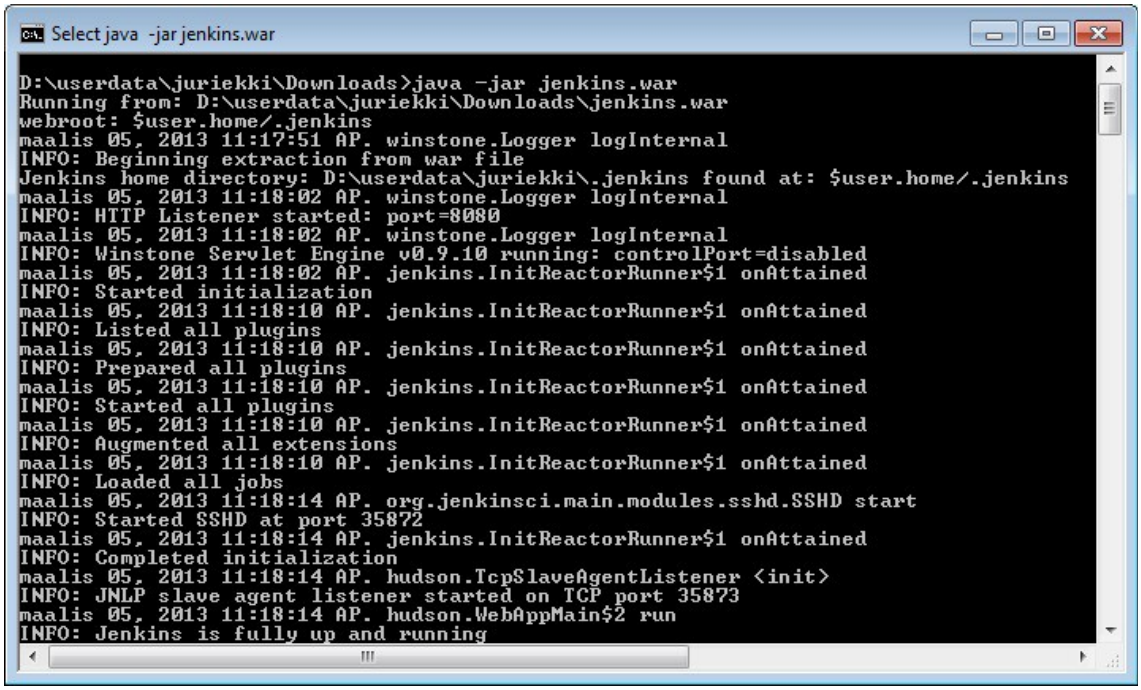


KUVIO 4. Jatkuvan integoinnin käyttö on mahdollista verkon välityksellä sijainnista riippumatta

4.1 Toteutus systeemikomponentissa

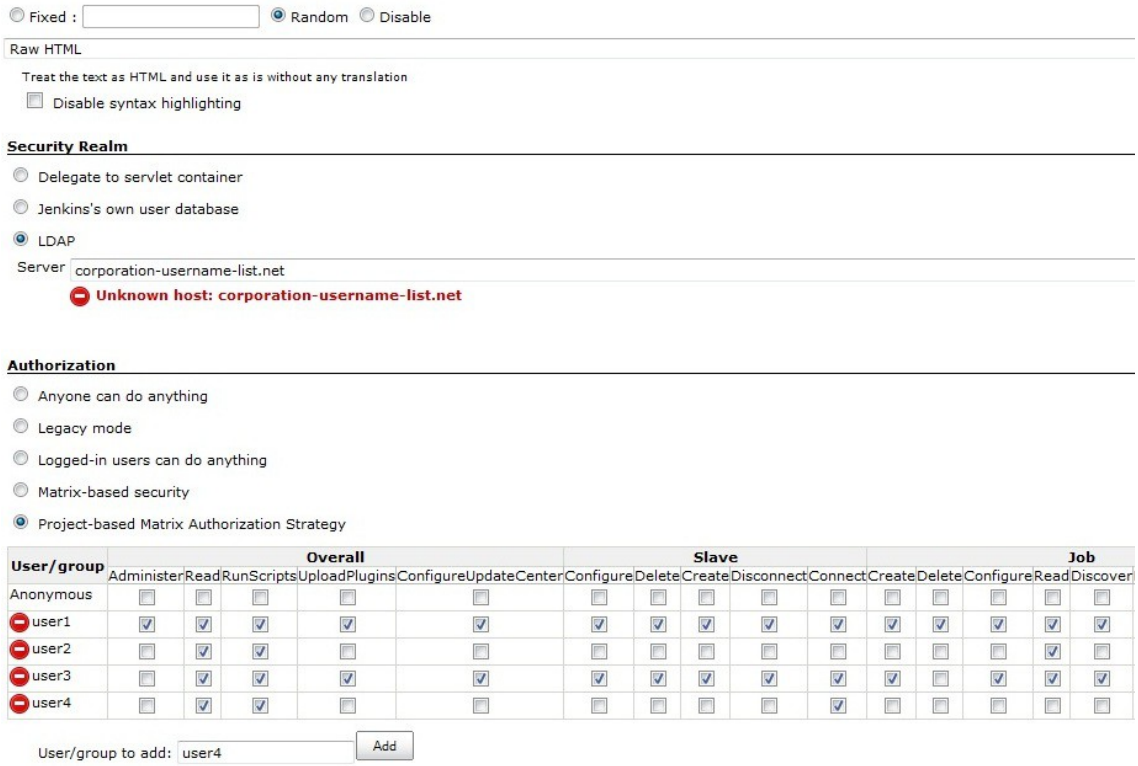
4.1.1 Asennus

Ensimmäinen vaihe oli asentaa ja määritellä Jenkins. Asennuspaketti ladataan osoitteesta http://jenkins-ci.org/. Asennus suoritetaan komennolla java -jar jenkins.war (JRE 1.5 tai uudempi). Onnistunut asennus näkyy kuviossa 5.



KUVIO 5. Jenkinsin asentaminen

Jenkins on tämän jälkeen asennettu ja käynnissä. Seuraavaksi määritellään Jenkinsin asetukset. Se tapahtuu web-sivun kautta, joka on asennuksen jälkeen oletusosoitteessa http://localhost:8080. Osoite vaihdetaan halutuksi osoitteeksi, joka on Jenkins-konetta varten varattu. Samalla määritellään kotihakemisto, jota Jenkins ja sille luodut jobit käyttävät. Jobikohtaisessa määrittelyssä voi muuttaa hakemiston halutuksi, mutta jos sitä ei tehdä, käytetään oletuskotihakemistoa. Samoin määritellään globaalit ympäristömuuttujat kuten kääntäjät sijainteineen, verkkolevyjen jaot palvelimille, käyttäjälistat ja sähköpostilistat.



KUVIO 6. Käyttäjäoikeuksien määrittäminen

Käyttäjälistan määrittelyllä voi määrätä, ketkä henkilöt saavat tehdä muutoksia asetuksiin, luoda, tuhota ja käynnistää jobeja. Sähköpostilistat luodaan vikatilanteita varten ja jokaiselle jobille on hyvä laittaa vastuuhenkilö, joka pitää huolen, että tarvittaviin toimenpiteisiin ryhdytään heti vian ilmettyä. Kuviossa 6 määritellään tarvittavat käyttöoikeudet käyttäjille.

Toinen vaihe on määritellä tarvittavat jobit ja niiden asetukset. Ensimmäiseksi luodaan käännös-jobi, johon määritellään kotihakemisto, versionhallinnan trunkin polku ja itse kääntäminen. Käännös suoritetaan komennoilla Jenkinsin ”Execute shell” -ikkunassa ja käännöksestä vastaavat tiedostot tulevat trunkin mukana. Käännöksen jälkeisiin toimintoihin lisätään haluttujen tiedostojen talletus testausta ja julkaisua varten, samoin määritellään, mitkä jobit ajetaan käännös-jobin jälkeen, ja määritellään sähköpostiosoitteet, joihin lähetetään tieto epäonnistuneesta käännöksestä. Versionhallinnan polun lisäksi määritellään aikaväli, jolloin versionhallinnan muutoksia käydään tiedustelemassa. Tämä aikaväli määritellään tarpeen mukaan, jatkuvassa

kehitysvaiheessa olevalle ohjelmistolle pieni aikaväli, ylläpidossa olevalle suurempi turhan kuorman välttämiseksi.

Seuraavaksi luodaan testaus-jobi. Testaus-jobia varten pitää ensin olla testiympäristöt ja niitä käyttävät Jenkins slave -noodit luotuna. Käytännössä tämä on tehty siten, että jokaiselle testiraudalle luodaan oma virtuaalikone Linux-koneessa. Virtuaalikone on ohjelmallisesti toteutettu tietokone, jossa voidaan suorittaa ohjelmia kuten oikeassa tietokoneessa. Virtuaalikone käynnistetään ja sille asetetaan dynaaminen verkko-osoite, jota Jenkins-kone voi käyttää yhteydenottoon. Jenkinsin päänäkymästä valitaan Manage Jenkins -> Manage Nodes -> New Node. Noodille annetaan kuvaava nimi, kuvaus mitä noodi tekee, yhtäaikaisten suorittajien määrä ja itse yhteyden muodostusta varten määritelty dynaaminen verkko-osoite ja käyttäjätunnukset. Samalla voi myös antaa parametreja Javalle, esimerkiksi muistin maksimimäärä. Yhtäaikaisten suorittajien määräksi on laitettava yksi, koska testirautoja on vain yksi noodin takana ja testiajot suoritetaan yksi kerrallaan. Todellisuudessa useampi kuin yksi Jenkins-jobi voi käyttää samaa testilaitteistoa johtuen useasta ohjelmistotrunkista. Tällä määrittelyllä estetään toisen ohjelmistohaaran testiajon käynnistyminen, kunnes ensin aloittanut testiajo on suoritettu loppuun. Tämän jälkeen Jenkins slave on määritelty ja voidaan käynnistää. Käynnistys suoritetaan Jenkinsin Manage Nodes -sivulta ja käynnistys voidaan suorittaa SSH:n yli tai esimerkiksi Javalla, mutta tässä tapauksessa on käytetty SSH-yhteyttä. Virtuaalikoneessa pitää siis olla SSH-tuki. Noodi on nyt käyttövalmis ja sitä voi käyttää Jenkins-jobeista.

Kuviossa 7 on esitelty onnistunut noodin käynnistys. Tämä prosessi käynnistyy virtuaalikoneella ja jää taustalle ajoon, kunnes sitä komennetaan Jenkins-koneelta. Dynaamista osoitetta käytettäessä on syytä varmistaa, että osoite on saatavilla ennen noodin käynnistystä; jos dynaaminen osoite ei ole ehtinyt rekisteröityä nimipalvelimelle, seuraa virheilmoitus.

```
[03/04/13 08:25:21] [SSH] Opening SSH connection to fi-ou-psci112.dynamic.nsn-net.net:22.
[03/04/13 08:25:21] [SSH] Authenticating as psci/******.
[03/04/13 08:25:22] [SSH] Authentication successful.
[03/04/13 08:25:30] [SSH] The remote users environment is:
BASH=/bin/bash
BASHOPTS=cmdhist:extquote:force_fignore:hostcomplete:interactive_comments:progcomp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_EXECUTION_STRING=set
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="4" [1]="2" [2]="24" [3]="1" [4]="release" [5]="x86_64-pc-linux-gnu")
BASH_VERSION='4.2.24(1)-release'
DIRSTACK=()
EUID=1000
GROUPS=()
HOME=/home/psci
HOSTNAME=fi-ou-psci112
HOSTTYPE=x86_64
IFS=$' \t\n'
LANG=en_US.UTF-8
LANGUAGE=en_US:en
LOGNAME=psci
MACHTYPE=x86_64-pc-linux-gnu
MAIL=/var/mail/psci
OPTERR=1
OPTIND=1
OSTYPE=linux-gnu
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
PIPESTATUS=([0]="0")
PPID=25597
PS4='+ '
PWD=/home/psci
SHELL=/bin/bash
SHELLOPTS=braceexpand:hashall:interactive-comments
SHLVL=1
SSH_CLIENT='10.145.9.5 40547 22'
SSH_CONNECTION='10.145.9.5 40547 10.145.10.170 22'
TERM=dumb
UID=1000
USER=psci
_=']'
[03/04/13 08:25:31] [SSH] Checking java version of java
[03/04/13 08:25:31] [SSH] java -version returned 1.7.0_07.
[03/04/13 08:25:31] [SSH] Starting sftp client.
[03/04/13 08:25:31] [SSH] Copying latest slave.jar...
[03/04/13 08:25:31] [SSH] Copied 284,160 bytes.
[03/04/13 08:25:31] [SSH] Starting slave process: cd '/home/psci/jenkins' && java -Xmx1g -XX:MaxPermSize=1g
<===[JENKINS REMOTING CAPACITY]===>channel started
Slave.jar version: 2.22
This is a Unix slave
Copied maven-agent.jar
Copied maven3-agent.jar
Copied maven3-interceptor.jar
```

KUVIO 7. Noodin käynnistys virtuaalikoneessa

Virtuaalikoneesta voidaan muodostaa yhteys testattavaan laitteistoon monella tapaa. Eräs tapa on määritellä jokaiselle raudalle oma IP-osoite, johon virtuaalikoneesta otetaan yhteys. Tämä ei kuitenkaan ole tässä tapauksessa paras ratkaisu, koska tavoitteena on, että testilaitteisto voidaan vaihtaa lennosta, ja tämä tapa vaatisi määrittelyä testilaitteistoon, ennen kuin testejä pystyy suorittamaan. Toisia tapoja muodostaa yhteys ovat VPN-yhteys VPN-laitteella tai kytkimellä, jossa kytkimen tietyt portit määritellään tietyille virtuaalikoneille. Tässä työssä käytetään jälkimmäistä tapaa. Yhteys testilaitteistoon todetaan toimivaksi kirjautumalla virtuaalikoneeseen SSH:lla ja

lähettämällä testiraudan IP-osoitteeseen ICMP echo request -paketti, johon testilaitteiston kytkin vastaa. Jollei yhteydenmuodostus onnistu, tarkistetaan virtuaalikoneen palomuurin säännöt ja lisätään sinne tarvittavat oikeudet asetuksiin TCP:lle ja UDP:lle tulevaan ja lähtevään liikenteeseen. Jenkinsin testaus-jobit, joita ajetaan virtuaalikoneissa, on syytä määritellä siten, että jobit voidaan ajaa juuri luodulla ja käynnistetyllä virtuaalikoneella ilman mitään lisäalustuksia. Tämä säästää aikaa tulevaisuudessa, koska virtuaalikoneisiin joudutaan joskus ajamaan päivityksiä tai muuttamaan ne käyttämään eri Linux-versiota tms.

Useampaa laitteistoa testattaessa voidaan jokaiselle laitteelle luoda oma jobi tai tehdä niin sanottu multikonfiguraatiojobi. Multikonfiguraatiojobi käynnistää testit usealla noodilla yhtä aikaa ja käyttäytyy ylätasolla kuten yksi jobi. Sen etuna onkin näkyvyys eli siitä näkee yhdellä vilkaisulla ylätasolta, ovatko kaikki testit kaikilla ajettavilla raudoilla menneet läpi. Jos testauksessa on tullut esiin ongelma, tämän jobin auki klikkaamalla näkee, missä testinoodissa ongelma on ollut.

Multikonfiguraatiojobille määritellään luodut noodit, jotka ovat testilaitteistoihin yhteydessä. Seuraava vaihe määrittelyssä on ladata versionhallinnasta kaikki testauksessa tarvittavat työkalut, kuten JavaTesteri, UDP-kuuntelija ja itse testiskriptit joita JavaTesterillä ajetaan. Nämä voidaan ladata testaus-jobille ”Execute shell” -ikkunassa käyttäen ”svn export” -komentoa. SVN–komennoille annetaan parametreina ”-non-interactive” ja ”-trust-server-cert”, joilla vältytään SVN-kyselyiltä, joihin ei voi automaattiympäristössä vastata. Työhakemisto tyhjennetään ja työkalut ja testiskriptit ladataan joka kerta ennen testiajon suoritusta. Tämä tapa osoittautui ylläpitovaiheessa parhaaksi, koska testityökalut, lokitusohjelmat ja testiskriptit muuttuvat ja niihin tehdään vikakorjauksia. Muutoksien tekeminen kaikkiin virtuaalikoneisiin veisi aikaa ja samalla inhimillisten virheiden mahdollisuudet moninkertaistuvat. Samassa shell-ikkunassa määritellään myös, millä noodilla ajetaan mikin testiskripti if-elif-else-valintarakenteella (kuvio 8).

Execute shell

Command

```
#!/bin/bash

#create directory for test results!
mkdir test_results
mkdir log

svn export --username=xxx --password=xxx --non-interactive --trust-server-cert
https://svn_repositry_address/udp_logger
svn export --username=xxx --password=xxx --non-interactive --trust-server-cert
https://svn_repositry_address/JaveTesteri

chmod -R 777 *

if [ "$slave" == "HW1" ]; then
    export logfile=HW1
    ./udp_logger -p 51001 log/$logfile.log &
    java -jar JavaTesteri.jar -c -f Test/HW1.xml; let test_result=$?

elif [ "$slave" == "HW2" ]; then
    export logfile=HW2
    ./udp_logger -p 51001 log/$logfile.log &
    java -jar JavaTesteri.jar -c -f Test/HW2.xml; let test_result=$?

elif [ "$slave" == "HW3" ]; then
    export logfile=HW3
    ./udp_logger -p 51001 log/$logfile.log &
    java -jar JavaTesteri.jar -c -f Test/HW3.xml; let test_result=$?

elif [ "$slave" == "HW4" ]; then
    export logfile=HW4
    ./udp_logger -p 51001 log/$logfile.log &
    java -jar JavaTesteri.jar -c -f Test/HW4.xml; let test_result=$?

elif [ "$slave" == "HW5" ]; then
    export logfile=HW5
    ./udp_logger -p 51001 log/$logfile.log &
    java -jar JavaTesteri.jar -c -f Test/HW5.xml; let test_result=$?

elif [ "$slave" == "HW6" ]; then
    export logfile=HW6
    ./udp_logger -p 51001 log/$logfile.log &
    java -jar JavaTesteri.jar -c -f Test/HW6.xml; let test_result=$?

else
  echo "testing under construction"
fi

pkill -f udp_logger
sleep 2
exit $test_result
```

KUVIO 8. Multikonfiguraatiojobin if-else-elif-valintarakenne

Testijobin loppuun määritellään vielä, mitkä tiedostot halutaan säilyttää, kuten lokit ja tulokset. Lisäksi listataan niiden henkilöiden sähköpostiosoitteet, ketkä jobin seurannasta ovat vastuussa. Testitulokset voidaan näyttää graafisesti käyttämällä Jenkinsin ”Publish JUnit test result report” -ominaisuutta, jolle annetaan polku mistä testitulokset löytyvät. Testitulosten on oltava JUnit XML -formaatissa. Tässä työssä käytettävä JavaTesteri tekee tuloksista automaattisesti JUnit XML -tiedoston jokaisen ajon jälkeen.

4.1.2 Käyttöönotto

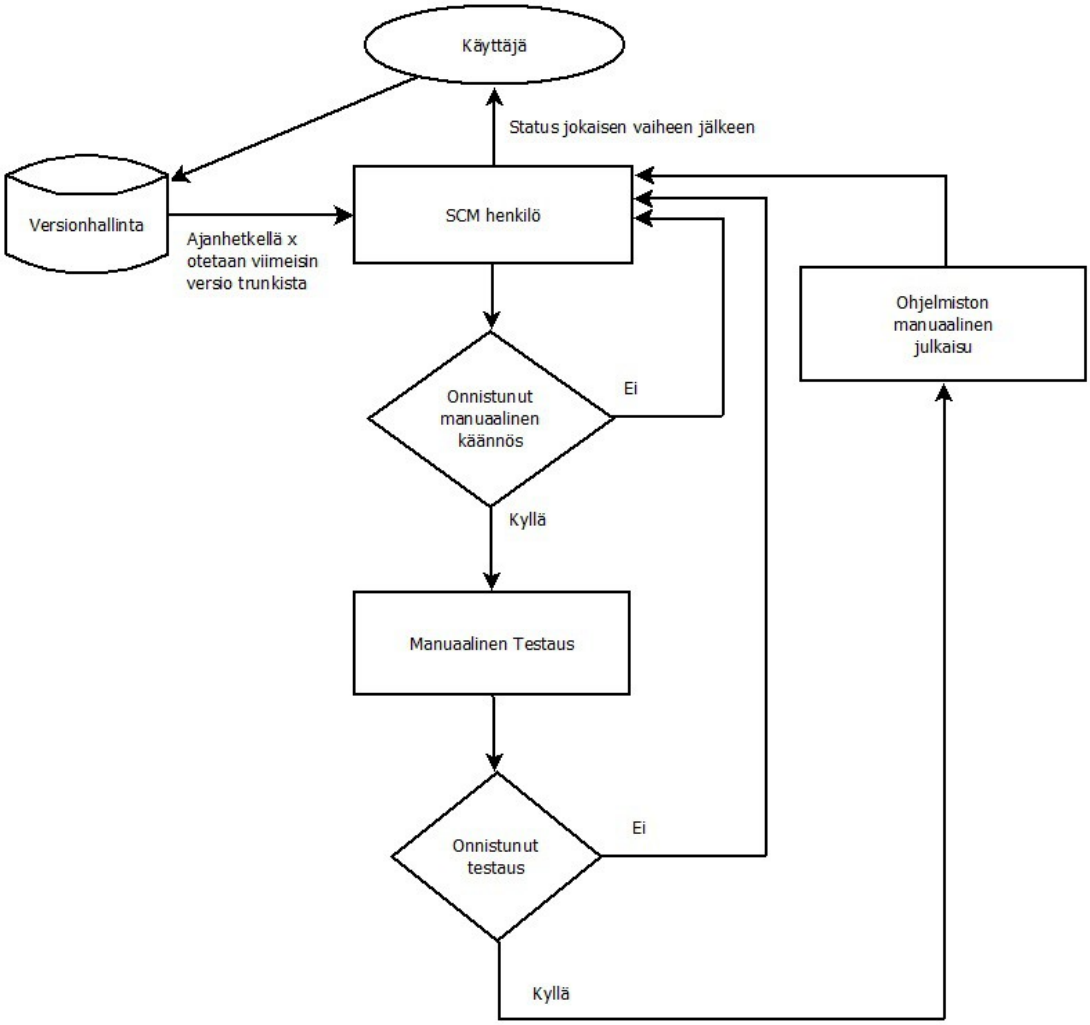
Määrittelyn ja tarvittavien jobien luomisen jälkeen jatkuva integrointi on valmis käytettäväksi. Versionhallinnan muutos käynnistää käännös-jobin ja läpi mennessään se käynnistää testaus-jobin. Käännös-jobin epäonnistuessa jobi lähettää sähköpostia määrätyille henkilöille sekä niille, jotka ovat tehneet muutoksia edellisen onnistuneen suorituksen jälkeen. Tällöin testaus-jobia ei käynnistetä. Epäonnistunut testaus-jobi lähettää myös sähköpostia samalla periaatteella kuin käännös-jobi. Kun käännös-jobi ja siitä seuraava testaus-jobi ovat menneet onnistuneesti läpi, tuloksena on julkaisukelpoinen ohjelmisto. Julkaisua varten tallessa on käännös-jobista testaukseen otetut tiedostot, joilla testit on onnistuneesti suoritettu ja ne voidaan toimittaa eteenpäin. Myös ohjelmiston trunkin versiotieto säilötään, jolloin kuka tahansa voi ottaa tietyn, testit läpäisseen version trunkista omalle koneelleen ja tehdä muutoksia sitä vasten. Käännöksen tuotokset voidaan siis tuottaa identtisenä kun trunkin versio tiedetään.

Jatkuva integrointi ei näy käyttöönottovaiheessa ohjelmiston kehittäjälle lainkaan, jos versionhallintaohjelma on sama kuin aikaisemmin käytetty. Ensimmäinen epäonnistunut käännös tai testaus, joka lähettää sähköpostia muutoksen tehneelle henkilölle, on ensimmäinen merkki sen olemassaolosta. Käyttöönotto pitääkin tiedottaa kehittäjätiimille hyvissä ajoin, jotta kehittäjät osaavat varautua nopeaan palautteeseen ja myös korjata viallisen tuotteen heti. Myös asennoituminen pitää saada positiiviseksi esimerkiksi pitämällä koulutus jatkuvan integroinnin tuomista eduista.

Manuaaliseen integrointiin verrattuna kehittäjälle tuleva palaute tulee suoraan jatkuvan integroinnin työkaluista eikä paketoijalta tai testaajalta. Palaute on välitöntä ja vian paikantaminen ja korjaaminen yksittäisen kehittäjän kannalta on helpompaa. Paketoinnista vastaavaa henkilöä ei tarvita enää samassa mittakaavassa, korkeintaan sisällön hallinnassa ja järjestelmän ylläpidossa.

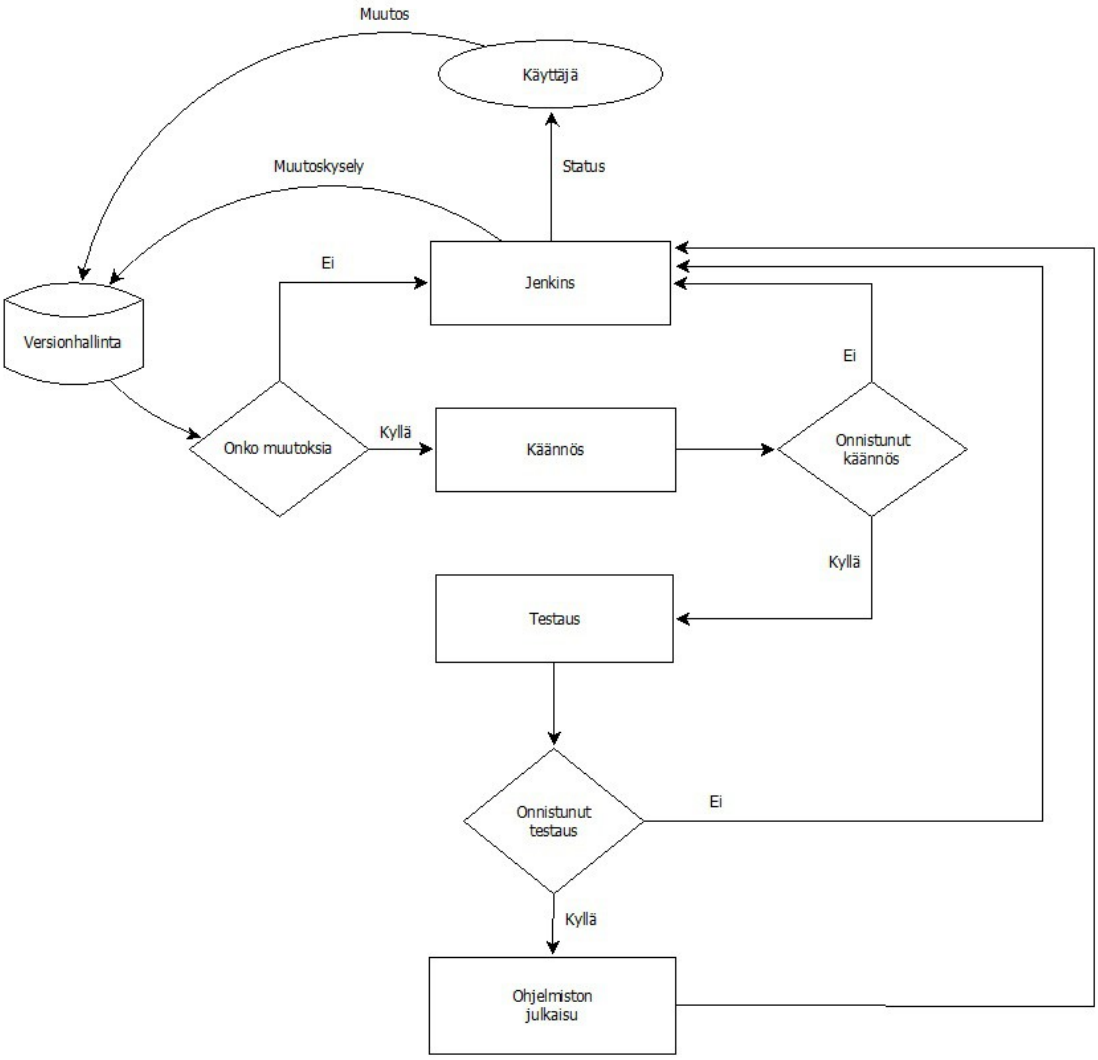
Testaajan työksi jää testisettien ylläpitäminen ja päivittäminen sekä testituloksien seuraaminen ja hän toimii testaus-jobien ylläpitäjänä. Vian ilmetessä testaaja toimii esitutkijana, koska hänellä on aiempaa kokemusta ja tietoa järjestelmästä ja hän osaa tehdä tarvittavat toimenpiteet tietyn tyyppisten vikojen poissulkemiseksi. Testaajalle tulee aina sähköposti viallisesta testituloksesta, ja tarvittaessa hän ottaa yhteyttä viallisen komponentin vastuuhenkilöön.

Manuaalista prosessia esitellään kuviossa 9, jossa jokainen nuoli kuvaa manuaalista toimintaa. Tässä toimintamallissa SCM- ja testaushenkilön vastuut ovat suuret. Henkilökohtaiset virheet jossain kohtaa prosessia aloittavat koko toiminnan alusta.



KUVIO 9. Toimintakaavio manuaalisesta integroinnista.

Jatkuvan integroinnin prosessi (kuvio 10) minimoi virheitä automatiikan ansiosta. Ainoa manuaalinen toiminto tässä mallissa on käyttäjän tiedoston päivitys versionhallintaan.



KUVIO 10. Toimintakaavio jatkuvasta integroinnista

4.2 Riippuvuudet ja ongelmat

Edellisessä luvussa kuvattu toteutus on toimiva itsenäiselle systeemikomponentille. Tässä työssä jouduttiin kuitenkin ottamaan huomioon muita systeemikomponentteja ja testauksessa tarvittiin myös niiden käännöstuotoksia. Käytännössä jo käännös-jobin linkkausvaiheessa tarvitaan kirjastoja toisesta systeemikomponentista ja ne kopioidaan ennen käännöksen aloittamista kyseisen systeemikomponentin vastaavasta käännös-jobista. Jotta kopiointi onnistuu, kopioitavat tiedostot on oltava määritelty arkistoitavaksi myös jobissa, josta kopiointi suoritetaan. Samoin testausvaiheessa tarvitaan kolmannen systeemikomponentin käännöstuotoksia ja nämä kopioidaan testaus-jobissa kyseisen komponentin käännös-jobista.

Testauksessa integroidaan jatkuvasti omaa systeemikomponenttia sekä varmistetaan muiden systeemikomponenttien yhteensopivuus omaan komponenttiin nähden. Oman komponentin sisäisten ongelmien lisäksi testauksessa paljastuvat rajapintaongelmat useamman komponentin välillä.

Systeemikomponentin sisäiset ongelmat johtuvat yleensä uudesta ominaisuudesta, joka vaatii muutoksia useampaan kuin yhteen komponenttiin. Päivityksiä viedään versionhallintaan niiden valmistuessa ja komponentit voivat näin olla hetkellisesti yhteensopimattomia toistensa kanssa. Toinen tyypillinen ongelma on puutteellinen virheenkorjaus, joka on kokeiltu toimivan vain tietyssä tilanteessa tai tietyssä laitteistossa.

Ohjelmiston ylätason uudet vaatimukset vaikuttavat monesti useampaan systeemikomponenttiin ja näiden toteutushetket ja versionhallintaan päivitykset aiheuttavat yleensä ongelmia. Esimerkiksi systeemikomponentti x tarvitsee systeemikomponentin y funktiota ja kutsuu sitä koodissa. Systeemikomponentti y ei ole vielä vienyt koko funktion toteutusta versionhallintaan ja tämä näkyy joko linkkausvaiheen vikailmoituksena tai testattavassa raudassa pahimmillaan kernel-virheenä ja kaataa testiraudan. Näiden ongelmien minimointiin ja ratkaisemiseen on jatkossa käytettävä entistä enemmän aikaa. Testilaitteistojen

uudelleen ohjelmoimiseen menee aikaa vian laadusta riippuen minuutista muutamaan tuntiin, ja jos jumiin menneitä testauslaitteistoja on useampia kuin yksi, tämä aika tietysti kertautuu.

Ongelmia aiheuttavat myös verkkovirheet, versionhallinnan toimimattomuus syystä tai toisesta, Java, tilan loppuminen palvelimilta jne. Java-virheet johtuvat yleensä muistin loppumisesta. Tällöin Jenkinsille on määritelty joko liian vähän muistia tai liian paljon muistia tai muistia vuotaa esimerkiksi viallisesta liitännäisestä. Muistivirheet kaatavat suorittajansa, joko itse Jenkinsin tai Jenkins slave -noodin. Nämä on tutkittava aina, koska todennäköisyys uusimiseen on suuri, ellei jopa täysin varma. Muistin loppuessa Jenkins ilmoittaa siitä konsolissa OutOfMemoryError-virheilmoituksella. Tästä ilmoituksesta johtuva muistiongelma tutkitaan lisäämällä kaatuneen suorittajan Javan käynnistykseen parametri ”-X:+HeapDumpOnOutOfMemoryError”, joka tuottaa koosteen muistin loputtua. Koosteesta voidaan selvittää muistin vuotamisen syy.

Käyttöönoton jälkeen jatkuvassa integroinnissa ilmeni myös muutama negatiivinen asia. Ensimmäinen oli satunnainen roskaposti, jota jatkuvan integroinnin työkalu lähettää ”syyttömille” käyttäjille. Tähän tilanteeseen joudutaan, kun ensimmäinen viallinen käännös ilmenee eikä sitä saada heti korjattua. Viallisen käännöksen päälle tehdyistä päivityksistä lähtee kaikille päivityksen tehneille ihmisille sähköpostia epäonnistuneesta ajosta, kunnes alkuperäinen vika on saatu korjattua. Toimivan muutoksenkin tehneet ihmiset ovat siis nyt vikalistalla. Yksi ketterien menetelmien periaatteista onkin, että rikkinäisen ohjelmiston päälle ei saa tehdä muita päivityksiä kuin sen hetkiseen vikaan liittyvän korjauksen.

Toinen merkille pantava haitta on hieman monimutkaisempi ratkaistava. Useaan komponenttiin vaikuttavaa rajapintamuutosta tehdessä integrointi menee yleensä rikki siihen asti, kunnes kaikki komponenttien väliset riippuvuudet on korjattu. Tämä vaikuttaa myös muiden projektissa työskentelevien työhön, koska heidän tekemänsä muutokset jäävät myös

kääntämättä ja testaamatta. Näin saattaa siis kestää pahimmassa tapauksessa yli vuorokausi, että yksittäinen muutos on saatu testattua. Tämän ongelman ratkaisemiseen tarvitaan sisällönhallinnan käyttöönottoa kaikissa systeemikomponenteissa.

Yhteiset rajapinnat aiheuttavat joskus myös sen, että yhden systeemikomponentin käännös ei toimi, koska siellä on riippuvuus toiseen systeemikomponenttiin. Jos toisen systeemikomponentin käännös on rikki syystä tai toisesta, on koko jatkuvan integroinnin prosessi rikki. Tässä tilanteessa pitäisi pystyä hallitusti peruuttamaan muutoksia kummastakin systeemikomponentista.

5 TESTAUS

Järjestelmän testaamiseksi täytyy tuntea niin tuotekoodia kuin testikoodia ja ymmärtää kokonaisuus ja muutosten tuomat vaikutukset. Helpoiten systeemin testaus tapahtuu muuttamalla testiskriptien parametreja. Näin muutos ei vaikuta testikoodiin eikä tuotekoodiin; syöttämällä vääriä parametreja saadaan testit odotetusti epäonnistumaan ja korjaamalla parametrit testit menevät taas läpi. Toisaalta tämä ei ole tarkoitus, vaan tarkoitus on, että jatkuva integrointi löytää uusia vikoja ja pitää huolen, että vanhoja, jo korjattuja vikoja ei pääse inhimillisten virheiden takia läpi. Toisin sanoen uuden vian ilmetessä pitää aina päivittää myös testisettiä tai luoda uusi testi ja ottaa se käyttöön.

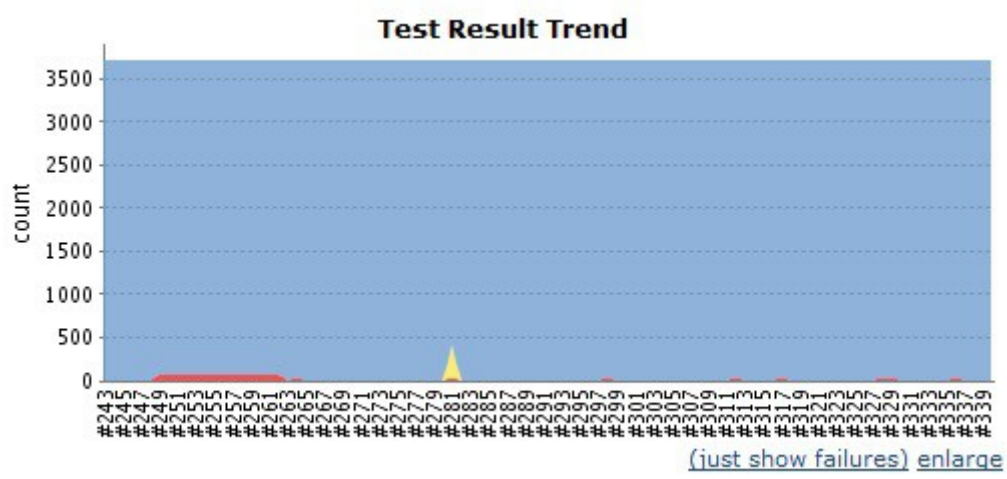
Testausta varten voi myös luoda versionhallintaan oman hiekkalaatikon, jolloin ei ole vaaraa että tehdyt muutokset menevät viralliseen tuotteeseen. Tässä tapauksessa myös tuotekoodiin voi tehdä muutoksia. Tämä on kuitenkin aika vaivalloista ja saattaa jumiuttaa testauslaitteistot.

Parhaiten testaus hoituu ottamalla toimivaksi todettu jatkuva integrointi käyttöön ja seuraamalla, miten se toimii ja löytää vikoja. Vian ilmetessä nähdään testaus-jobista epäonnistunut testiajo ja voidaan katsoa, mitä muutoksia edelliseen onnistuneeseen ajoon on. Näin pystytään joko peruuttamaan muutos tai korjaamaan mahdollisimman nopeasti viallinen komponentti. Testaus-jobi tekee myös lokia, josta vika voi selvitä. Lokia tutkimalla ja vertaamalla sitä vian aiheuttaneeseen muutokseen vika on yleensä helppo todentaa ja tilanteen kiireellisyydestä riippuen joko korjata tai peruuttaa muutos toimivaan versioon.

6 TULOKSET

Tulokset käyttöönotetusta jatkuvasta integroinnista ovat todella positiiviset. Tämä menetelmä ei tietenkään poista kaikkia vikoja, mutta nyt voidaan reagoida löydettyihin vikoihin heti, eikä vasta julkaisua ennen tapahtuvassa integroinnissa ja integrointitestauksessa. Vian löydyttyä ylemmällä ohjelmistokerroksella päivitetään omia testikoodeja niin, että sama vika ei pääse toistumaan, ja näin ollen jatkuvaa ohjelmiston laadun paranemista tapahtuu koko ajan. Samoin testaustrendiä on helppo seurata Jenkinsin piirtäessä testituloksista grafiikkaa kaikista testiajoista (kuvio 11). Myös testauskattavuus nousee aina kun uusi vika löytyy ja kattavuuteen kiinnitetäänkin nykyään entistä enemmän huomiota. Osasyynä tähän on jatkossa käyttöönotettavat työkalut, joilla testauskattavuutta mitataan.

Testaukseen kuluva aika saatiin testilaitteistojen kaksinkertaistumisen johdosta puolitettua ja niiden hallinta helpottui. Inhimillisten virheiden määrä testaustasolla on lähellä nollaa automatiikan johdosta. Myös ohjelmiston koostaminen automaattisesti poistaa virhetekijöitä. Aiemmin testauksessa oli monta kohtaa, jossa virheitä tapahtui johtuen eri testaajien erilaisista testilaitteistoista, käytännöistä jne. Nykyään testauslaitteisto on samanlainen jokaisella testauskierroksella ja vikojen ilmeneminen pystytään aina toistamaan.



KUVIO 11. Testituloksia

7 JATKOKEHITYSMAHDOLLISUUDET

Testaus tullaan jatkossa pilkkomaan tarpeen mukaan vieläkin useampaan osaan suoritusajan nopeuttamiseksi entisestään. Kaksinkertainen määrä testauslaitteistoja lähes puolittaa testaukseen menevän ajan. Tämä on helppo toteuttaa jakamalla testiskriptien testit osiin ja lisäämällä Jenkins-slaveja, virtuaalikoneita ja testauslaitteistoja. Laitteistojen rajallinen määrä on oikeastaan ainoa rajoittava tekijä.

Samoin testaus jaetaan useampaan vaiheeseen: Ensimmäisessä vaiheessa ladataan testattava ohjelmisto ja kokeillaan lähteekö yksikkö toimimaan lähettämällä testipaketti laitteiston verkkokytkimelle. Onnistuneen yhteyden jälkeen ohjelmiston systeemikomponenteille lähetetään testiviesti, jolla varmistutaan että palvelut ovat toiminnassa. Näin varmistetaan, että ohjelmisto käynnistyy ja se voidaan ladata kaikille testilaitteistoille ilman pelkoa laitteiston jumiin menemisestä. Tässä ensimmäisessä vaiheessa on tietenkin myös mahdollista, että ladattava ohjelmisto on epäkelpo eikä laitteisto käynnisty ollenkaan. Sen tarkoituksena onkin toimia puskurina isommalle määrälle testilaitteistoa, ja on huomattavasti nopeampi ohjelmoida uudelleen muutama laitteisto kuin yli 10 yksikköä.

Systeemikomponenttien toiminta toisiinsa nähden pitäisi saada myös paremmin varmistettua ennen testaamisen aloittamista. Turhaa työtä tulee paljon jos useampaa kuin yhtä systeemikomponenttia koskeva vaatimus on toteutettu vain yhdessä komponentissa ja käännös ja testaaminen aloitetaan silti. Sama pätee systeemikomponentin sisäisiin muutoksiin. Tämä parannus on kuitenkin todella vaikea toteuttaa, koska käännös-jobi aloittaa käännöksen aina huomatessaan versionhallinnassa uuden version trunkista. Näin ollen isommat muutokset täytyisi tehdä omassa hiekkalaatikossaan ja se ei ole jatkuvan integroinnin periaatteiden mukaista.

Testauksen koodikattavuus ja sen näkyvyys on myös työlistalla. Kattavuutta voidaan mitata esimerkiksi Linuxin gcovilla tai Testwellin kehittämällä CTC++-työkalulla. Näistä jälkimmäinen on itselleni tutumpi ja sitä voi käyttää molemmissa tukiaseman käyttöjärjestelmissä. CTC++ instrumentoi koodikattavuuden kohteena olevat tiedostot, ja aina kun testikoodi käy instrumentoidussa tiedostossa, CTC++ kirjoittaa siitä lokiin. Lopuksi CTC++ muodostaa HTML-pohjaisen raportin, jonka voi julkaista testaus-jobissa testin suorituksen lopuksi.

Tulevaisuudessa ohjelmistosuunnittelijoiden avuksi on tarkoitus ottaa käyttöön koodianalyysityökalu. Se kertoo ohjelmiston laadusta, löytää ohjelmistovirheitä kuten käyttämättömiä muuttujia ja muistivuotoja ja luo niistä raportin. Tätä seuraamalla ja sääntöjä käännöksien välille luomalla jatkuvasta integroinnista saadaan entistä tehokkaampaa.

8 YHTEENVETO

Integrointi on ohjelmistokehityksen kriittisimpiä vaiheita ja siihen kuluvaa aikaa on pyritty minimoimaan monilla eri keinoilla. Yhteiset rajapinnat ja niiden jatkuva kasvaminen, samoin kuin integroitavien komponenttien lisääntyminen on todella haastavaa hallita pelkästään versionhallinnalla ja manuaalisella testauksella.

Tässä insinöörityössä luotiin eräälle systeemikomponentille jatkuva integrointi, joka sisältää automaattisen ohjelmiston käännöksen, linkkauksen ja testauksen oikeassa laitteistoympäristössä. Julkaistavan ohjelmiston manuaalinen testaus saatiin näin lopetettua kokonaan, samoin inhimilliset virheet minimoitua. Työ valmistui ja onnistui odotusten mukaan.

Suurin ja merkittävin muutos tällä työllä tuli itse testaukseen, jota suoritettiin manuaalisesti aina kun tarvittavat muutokset oli toteutettu versionhallintaan. Testauspyyntöjä tuli päivittäin ja uusintatestauspyyntöjä ohjelmiston laadusta riippuen yhtä paljon. Manuaalisesti suoritettu testaus jätti aina inhimillisen virheen mahdollisuuden. Samoin testaus saattoi sattua ajankohdalle, jolloin testaushenkilöitä ei ollut paikalla. Jatkuva integrointi on aina saatavilla ja testitulokset nähtävillä testien päätyttyä riippumatta testaushenkilöiden aikatauluista. Sen voi myös käynnistää käsin milloin tahansa ilman että tuotekoodiin on tullut muutosta.

Opinnäytetyön jälkeen jatkuva integrointi on ollut enenevässä määrin lisääntymässä koko yrityksessä ja virallisia ohjelmistojulkaisujakin luultavasti tullaan hoitamaan jatkuvan integroinnin kautta. Myös tässä työssä käsitellyt systeemikomponenttien välisistä riippuvuuksista johtuvat ongelmat on jo ratkaistu.

LÄHDELUETTELO

Abrahamsson P., Salo O., Ronkainen J. & Warsta J. P. 2002. Agile software development methods. Review and analysis. VTT Publications.

Agile Alliance 2013. Hakupäivä 12.3.2013, http://www.agilealliance.org.

Apache Subversion 2013. Hakupäivä 15.3.2013, http://subversion.apache.org.

Fowler, Martin. 2006. Continuous Integration. Hakupäivä 16.2.2013, http://www.martinfowler.com/articles/continuousIntegration.html.

Meet Jenkins. 2012. Hakupäivä 1.2.2013, http://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins.

Nokia Oyj 2013. Vuoden 2012 viimeisen neljänneksen ja koko vuoden 2012 katsaus. Hakupäivä 22.3.2013, http://press.nokia.fi/2013/01/24/nokia-oyjn-vuoden-2012-viimeisen-neljanneksen-ja-koko-vuoden-2012-katsaus.

Nokia Siemens Networks Oy 2013. Hakupäivä 22.3.2013, http://www.nokiasiemensnetworks.com/about-us/company.

Shore, J. & Warden, S. P. 2008. The art of agile development
O`Reilly Media

Sommerville, Ian. P. 2007. Software Engineering. 8th edition.
Pearson Education Limited.

HW1.xml

```
<SCENARIO xmlns:xi="http://www.w3.org/2001/XInclude">
   <INFO>
      <NAME>ReleaseTestHW1</NAME>
      <BRIEF>Release Test set for HW1</BRIEF>
   </INFO>
   <CONNECTION>
      <IP>&targetHW1_IpAddress;</IP>
      <PORT>&targetHW1_Port;</PORT>
      <TYPE>TCP</TYPE>
   </CONNECTION>
   <TESTCASES>
      <xi:include href="InitializeHW.xml"/>
      <xi:include href="SwDl.xml"/>
      <xi:include href="ReleaseTest1.xml"/>
      <xi:include href="ReleaseTest2.xml"/>
      <xi:include href="ReleaseTest3.xml"/>
      <xi:include href="ReleaseTest4.xml"/>
      <xi:include href="ReleaseTest5.xml"/>
      <xi:include href="ReleaseTest6.xml"/>
      <xi:include href="ReleaseTest7.xml"/>
      <xi:include href="ReleaseTest8.xml"/>
      <xi:include href="ReleaseTest9.xml"/>
      <xi:include href="Reset.xml"/>
   </TESTCASES>
</SCENARIO>
```

ReleaseTest1.xml

```
<SCENARIO>
  <INFO>
    <NAME>ReleaseTest1</NAME>
    <BRIEF></BRIEF>
  </INFO>
  <CONNECTION>
    <IP>&targetHW1_IpAddress;</IP>
    <PORT>&targetHW1_Port;</PORT>
    <TYPE>TCP</TYPE>
  </CONNECTION>
  <TESTCASES>
    <CASE>
      <INFO>
        <UINAME>ReleaseTest1 - Startup for task1</UINAME>
        <BRIEF></BRIEF>
      </INFO>
      <MESSAGE>
        <TBTSHEADER>
          <BOARD>&tested_board;</BOARD>
          <CPU>&tested_cpu;</CPU>
          <SUBSYSTEMNAME>Startup</SUBSYSTEMNAME>
          <TESTCASENAME>StartupCheck</TESTCASENAME>
        </TBTSHEADER>
        <PARAM name="rec_task" valueType="u16">&tested_task;</PARAM>
        <PARAM name="rec_cpu" valueType="u8">&tested_cpu;</PARAM>
        <PARAM name="rec_board" valueType="u8">&tested_board;</PARAM>
        <PARAM name="loop_type" valueType="u32">0</PARAM>
      </MESSAGE>
    </CASE>
    <CASE>
      <INFO>
        <UINAME>ReleaseTest1 - Startup for task2</UINAME>
        <BRIEF></BRIEF>
      </INFO>
      <MESSAGE>
```

```
            <TBTSHEADER>
               <BOARD>&tested_board;</BOARD>
               <CPU>&tested_cpu;</CPU>
               <SUBSYSTEMNAME>Startup</SUBSYSTEMNAME>
               <TESTCASENAME>StartupCheck</TESTCASENAME>
            </TBTSHEADER>
            <PARAM name="rec_task" valueType="u16">&tested_task;</PARAM>
            <PARAM name="rec_cpu" valueType="u8">&tested_cpu;</PARAM>
            <PARAM name="rec_board" valueType="u8">&tested_board;</PARAM>
            <PARAM name="loop_type" valueType="u32">0</PARAM>
         </MESSAGE>
      </CASE>
   </TESTCASES>
</SCENARIO>
```

Junit_XML_HW1.xml

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<testsuite errors="0" failures="0" name="HW1" skipped="0" tests="33" time="843.968" timestamp="2013-03-10T16:33:04">
<properties>
<property name="sw.name" value="JavaTesteri"/>
<property name="sw.version" value="0.9"/>
<property name="sw.date" value="201301301640"/>
<property name="java.version" value="1.7.0_09"/>
<property name="java.home" value="/usr/lib/jvm/java-7-openjdk-amd64/jre"/>
<property name="os.name" value="Linux"/>
<property name="os.arch" value="amd64"/>
<property name="user.dir" value="/home/jenkins/workspace/MAINBRANCH/slave/HW1"/>
<property name="user.timezone" value="Europe/Helsinki"/>
</properties>
<testcase classname="InitializeHW" name="0: Init1" row="0" time="0.097"/>
<testcase classname="InitializeHW" name="1: Init2" row="1" time="0.664"/>
<testcase classname="InitializeHW" name="2: init3" row="2" time="0.513"/>
<testcase classname="InitializeHW" name="3: Init4" row="3" time="52.144"/>
<testcase classname="SwDl" name="4: Updatesw1" row="4" time="12.988"/>
<testcase classname="SwDl" name="5: Updatesw2" row="5" time="22.459"/>
<testcase classname="SwDl" name="6: Updatesw3" row="6" time="22.319"/>
<testcase classname="SwDl" name="7: Updatesw4" row="7" time="483.841"/>
<testcase classname="SwDl" name="8: Updatesw5" row="8" time="0.035"/>
<testcase classname="SwDl" name="9: Updatesw6" row="9" time="0.061"/>
<testcase classname="ReleaseTest1" name="10: Startup for task1" row="10" time="0.06"/>
<testcase classname="ReleaseTest1" name="11: Startup for task2" row="11" time="0.06"/>
<testcase classname="ReleaseTest2" name="12: Subcase1" row="12" time="0.06"/>
<testcase classname="ReleaseTest2" name="13: Subcase2" row="13" time="0.06"/>
<testcase classname="ReleaseTest3" name="14: Subcase1" row="14" time="0.028"/>
<testcase classname="ReleaseTest3" name="15: Subcase2" row="15" time="0.202"/>
<testcase classname="ReleaseTest4" name="16: Subcase1" row="16" time="58.748"/>
<testcase classname="ReleaseTest4" name="17: Subcase2" row="17" time="40.093"/>
<testcase classname="ReleaseTest4" name="18: Subcase3" row="18" time="60.739"/>
<testcase classname="ReleaseTest4" name="19: Subcase4" row="19" time="40.095"/>
<testcase classname="ReleaseTest5" name="20: Subcase1" row="20" time="0.035"/>
<testcase classname="ReleaseTest5" name="21: Subcase2" row="21" time="0.062"/>
<testcase classname="ReleaseTest6" name="22: Subcase1" row="22" time="1.029"/>
<testcase classname="ReleaseTest7" name="23: Subcase1" row="23" time="5.047"/>

<testcase classname="ReleaseTest7" name="24: Subcase2" row="24" time="0.062"/>
<testcase classname="ReleaseTest7" name="25: Subcase3" row="25" time="0.059"/>
<testcase classname="ReleaseTest7" name="26: Subcase4" row="26" time="0.06"/>
<testcase classname="ReleaseTest8" name="27: Subcase1" row="27" time="14.045"/>
<testcase classname="ReleaseTest8" name="28: Subcase2" row="28" time="14.037"/>
<testcase classname="ReleaseTest8" name="29: Subcase3" row="29" time="14.037"/>
<testcase classname="ReleaseTest8" name="30: Subcase4" row="30" time="0.062"/>
<testcase classname="ReleaseTest9" name="31: Subcase1" row="31" time="0.049"/>
<testcase classname="ReleaseTest9" name="32: Subcase2" row="32" time="0.06"/>
<testcase classname="ReleaseTest9" name="33: Subcase3" row="33" time="0.058"/>
</testsuite>