TEKNIIKKA JA LIIKENNE

Tietotekniikka

Ohjelmistotekniikka

INSINÖÖRITYÖ

TESTAUS JATKUVAN INTEGROINNIN OSANA METROPOLIAN OHJELMISTOTUOTANTOPROJEKTEISSA

Työn tekijä: Aki Vuokoski
Työn ohjaajat: Auvo Häkkinen
Erja Nikunen

Työ hyväksytty: ___. ___. 2011

Auvo Häkkinen
yliopettaja

ALKULAUSE

Tämä työ on tehty Metropoliassa metropolialaisille. Toivottavasti joku ohjelmistotekniikkaa opiskeleva hyötyy tämän työn vinkeistä. Tein googlettamisen jo puolestasi.

Kiitän yliopettajia Auvo Häkkistä ja Erja Nikusta kesätyöprojektin ja insinöörityön ohjauksesta. Projektipartnerini Aleksi Lukkarinen ansaitsee erityiskiitoksen, sillä ilman hänen teknistä osaamistaan moni tyhmä kysymys olisi jäänyt vastausta vaille. Kiitän myös Jussi Alhorinnettä kielenhuollosta.

Helsingissä 27.4.2011

Aki Vuokoski

TIIVISTELMÄ

Työn tekijä: Aki Vuokoski

Työn nimi: Testaus jatkuvan integroinnin osana Metropolian ohjelmistotuotantoprojekteissa

Päivämäärä: 27.4.2011 **Sivumäärä:** 49 s. + 1 liitettä

Koulutusohjelma: Tietotekniikka

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Työn ohjaaja: yliopettaja Auvo Häkkinen

Työn ohjaaja: yliopettaja Erja Nikunen

Insinöörityön tavoitteena oli tutkia automatisoidun testauksen mahdollisuuksia automatisoidulla koonti- ja testausjärjestelmällä, jota kehitettiin Metropolia Ammattikorkeakoulun ohjelmistotuotantoprojektikurssia varten. Automatisoinnin teoria perustuu jatkuvaan integrointiin. Tutkimuksen kriteereihin kuului huomioida ohjelmistotekniikan opiskelijoiden tietotaso ja kurssin aikataulu, jotta opiskelijat pääsisivät työn tietojen pohjalta helposti alkuun. Työ rajattiin kattamaan Java-ohjelmointiprojektien testauksen.

Koonti- ja testausjärjestelmän pohjaksi valittiin jatkuvan integroinnin työkalu Hudson ja koontityökalu Maven 2. Hudson osoittautui varsin helppokäyttöiseksi onnistuneen asennuksen jälkeen. Maven 2 vaatii opiskelijalta konfiguraatiotiedoston hallintakykyä, joten työssä keskityttiin vahvasti käytännön esimerkkeihin.

Työkaluissa ja menetelmissä hyödynnettiin avoimen lähdekoodin ratkaisuja. Normaali JUnit yksikkötestaus ja staattinen analyysi todettiin helpoksi ottaa käyttöön osaksi jatkuvaa integraatiota. Tämä on opiskelijoiden oman kiinnostuksen pohjalta mahdollista laajentaa kattamaan tietokantatestauksen DBUnitilla ja XML-testauksen XMLUnitilla. Verkkopohjaisten projektien hyväksymistestien tekeminen havaittiin varsin pienellä vaivalla mahdolliseksi Selenium-testauksessa, jos projektilaisilla on aikaa testiympäristön raskaampaan konfiguroimisvaiheeseen.

Maven-pohjaisten testausratkaisujen saatavuus on vaihtelevaa, joten kaikkia testaustyylejä ei opiskelijoille suositeltu. Penetraatiotestauksen, mutaatiotestauksen ja Swing-hyväksymistestauksen automatisointi vaatii turhan paljon vaivaa ohjelmistotuotantoprojektikurssin puitteisiin.

Ohjelmistotuotantoprojekteissa testaus jää usein muiden töiden alle pelkäksi suunnitelmaksi. Tämän työn esimerkit auttavat opiskelijat helposti alkuun automatisoidussa testauksessa, jolloin uuden tiedon opiskeluun menee vähemmän aikaa ja motivaatio testaukseen voi kasvaa.

Avainsanat: testaus, jatkuva integraatio, automatisointi, Maven, Java

ABSTRACT

Name: Aki Vuokoski

Title: Testing as Part of Continuous Integration in Metropolia's Software Development Projects

Date: 27.4.2011 **Number of pages:** 49

Department: Information technology

Study Programme: Software engineering

Instructor: Auvo Häkkinen, principal lecturer

Instructor: Erja Nikunen, principal lecturer

The purpose of this final year project was to research the possibilities of automated testing within automated build and test environment developed for Software development projects in Metropolia University of Applied Sciences. The automation is based on Continuous Integration and the focus is at Java projects. The software engineering students' technical knowledge and the course schedule are noted in the study criteria.

The build and test environment software base was chosen to include the continuous integration tool Hudson and build tool Maven 2. Hudson was proven to be an easy-to-use tool after a successful installation. Maven 2 requires basic knowledge of Maven's configuration file.

The chosen tools and procedures use open source solutions. Basic JUnit unit testing and static analysis were discovered to be easily utilized with continuous integration. This can be expanded to include database testing with DBUnit and XML testing with XMLUnit. Acceptance testing for web software projects allocates some time in order to configure the test environment but making the tests was shown to be otherwise quite effortless with Selenium.

The availability of Maven based test solutions varies. Therefore not all testing can be recommended to the students. The automation of penetration testing, mutation testing and acceptance testing for Swing applications requires too much effort within the project schedule.

The examples in this final year project should help the students getting started with the automated testing. The less time wasted on learning the basics may motivate to actually do some testing during the software project.

Keywords: testing, continuous integration, automation, Maven, Java

SISÄLLYS

1 JOHDANTO

Metropolia Ammattikorkeakoulussa ohjelmistotekniikan opiskelijat tekevät pienissä ryhmissä laajemman ohjelmistotuotantoprojektin. Tällä kurssilla yksi henkilö, usein opettaja, toimii ryhmän asiakkaana ja toinen opettaja teknisenä neuvonantajana. Ulkopuolinen yritysasiakas on myös mahdollisuus. Kurssi on kaikista lähinnä todellista työelämän projektia ja koetaan usein hyvin opettavaisena.

Kurssi toteutettiin aiemmin vesiputousmallin mukaisena ohjelmistoprojektina. Vasta noin vuoden ajan mukaan on tuotu ketteriä menetelmiä, erityisesti SCRUM:ia, mukailevia työskentelytapoja. Kouluprojektille on varattu lukujärjestyksestä 10 tuntia viikossa yhden lukukauden verran. Lisäksi oletetaan, että opiskelijat työskentelevät itsenäisesti omalla ajallaan. Varsinaista SCRUM-projektia on mahdoton toteuttaa kouluympäristössä, koska ei työskennellä toimistotyöaikojen puitteissa. Projekti jakautuu viiteen sprinttiin, joista ensimmäinen on lyhyempi määrittelyvaihe. Tämän jälkeen suunnittelu, toteutus, testaus ja dokumentointi tulisi toteuttaa jokaisen sprintin osalta silloista tuotetta ja asetettuja vaatimuksia vastaavaksi. Asiakas priorisoi sprintin työt, jolloin voidaan keskittyä tärkeimpiin ominaisuuksiin ja varmistaa vähimmäisvaatimusten toteutuminen.

Projektissa hyödynnettävän tekniikan osalta opiskelijoille on aiemmin annettu käyttöön pelkästään tietovarasto versionhallinnasta ja MySQL-tietokanta. SCRUM-tyylinen kirjanpito on toteutettu paperilla ja Excel-taulukkolaskentaohjelmalla. Näin niukka varustelu harvemmin vastaa todellista työelämän ohjelmistokehitysympäristöä.

Jatkuva integrointi on monessa yrityksessä arkipäivää. Niukimmillaan se on ohjelman automatisoitua koontia aina, kun tietovarastoon tehdään muutoksia. Laajennettuna mukaan tulee myös automatisoitu yksikkötestaus ja integrointitestaus sekä dokumentaation tuottaminen. Menetelmän käyttöönotto ohjelmistotuotantoprojekteissa vaatisi minimissään pelkän koontipalvelimen, mutta kurssi olisi heti lähempänä yrityselämän todellisuutta, ja opettaisi käytännössä asian, joka vain mainitaan koulun teoriatunneilla.

Tässä työssä käydään läpi Java-pohjaisten ohjelmistoprojektien testauksen automatisointia jatkuvan integroinnin järjestelmässä. Työn ohessa ilmenee,

millainen on jatkuvaan integraatioon soveltuva järjestelmä. Esimerkkijärjestelmänä on Metropolian ohjelmistotuotantoprojektin opiskelijoille kehitetty automatisoitu koonti- ja testausjärjestelmä. Itse järjestelmä ei ole automatisoitu, vaan se antaa avaimet automatisoituun ohjelmistokehitykseen, missä opiskelijoiden versionhallintaan viety uusi ohjelmistoversio kootaan automaattisesti koontipalvelimella. Samaan prosessiin on mahdollista liittää erilaisia testiajoja.

Aluksi aihetta alustetaan luvussa kaksi, missä kerrotaan tarkemmin, mitä on jatkuva integrointi ja mihin sillä pyritään. Luvussa kolme käydään läpi koonti- ja testausjärjestelmän toiminnan kannalta oleellisia asioita, kuten millainen on toimiva palvelinkokonaisuus. Siinä käsitellään tarkemmin jatkuvaan integrointiin soveltuvat työkalut Hudson ja Maven. Hudsonista on asennusohjeistusta, jotta vastaavanlaista järjestelmää kehittävä voi välttää yleisiä ongelmia. Maven on hyvin oleellinen koontityökalu, jonka hyödyntäminen on yksi työn keskeisimpiä teemoja myös testauskappaleissa. Tämän vuoksi luku kolme sisältää myös Mavenin asetustiedoston rakentamisen perusteet.

Luku neljä käsittelee testauksen automatisointia. Kun jatkuvaan integrointiin kykenevä järjestelmä on pystytetty, tulee ohjelmoijan tietää, mitä erilaisia testauslajeja on ja kuinka niitä voidaan hyödyntää automatisoituna. Valikoiduissa ratkaisuissa on huomioitu Metropolian ohjelmistotuotannon opiskelijoiden tekniikantuntemus ja ohjelmistotuotantoprojektikurssin aikataulu. Lisäksi työkalut perustuvat avoimeen lähdekoodiin. Luku on jaettu staattiseen ja dynaamiseen testaukseen. Luvussa viisi tutustutaan järjestelmän tuottamaan dokumentaatioon, ja lopussa on yhteenveto havainnoista ja tuloksista.

2 JATKUVA INTEGROINTI

Jatkuva integrointi ohjelmistokehityksessä pyrkii poistamaan ajatusmallin, missä ohjelmisto syntyy eri vaiheiden summana. Esimerkkinä perinteisestä ohjelmistokehityksestä voidaan pitää vaikkapa vesiputousmallia, missä integrointi tulee moduulien ohjelmoinnin jälkeen ja integraatiotestaus vasta yksikkötestauksen perässä. Jatkuvassa integroinnissa kaikki vaiheet tulevat jokaisen käännöksen yhteydessä ja lopputulos on teoriassa julkaisukelpoinen ja testattu ohjelma [1]. Varsinkin ohjelmistokehityksen integraatiovaihe,

missä tuotteen eri moduulit yhdistetään toimivaksi kokonaisuudeksi, on vaikeasti arvioitava. Tämä monesti työlään ja aikaavievän vaiheen tarve katoaa jatkuvassa integraatiossa, koska integrointi tapahtuu pienissä osissa koko projektin ajan.

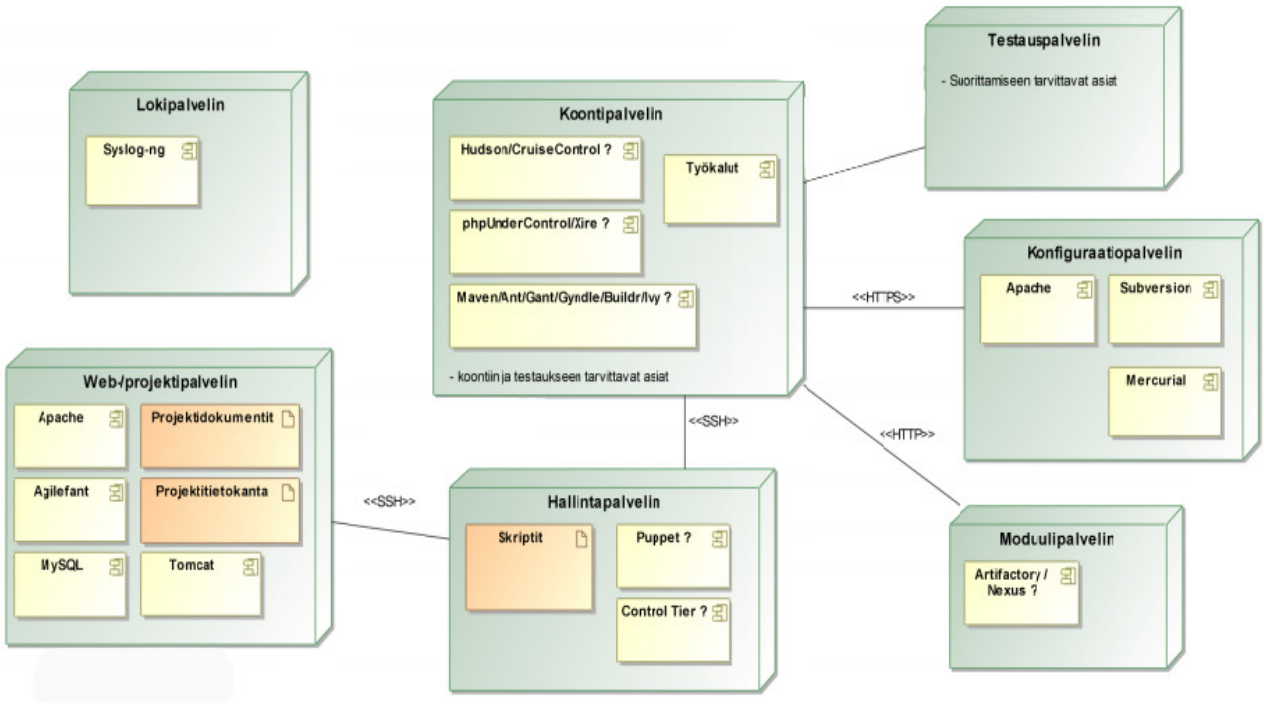
Kaikkien vaiheiden läpikäyminen manuaalisesti olisi erittäin aikaavievää. Tämän vuoksi yksi jatkuvan integroinnin perusajatuksia on se, että kaikki mikä voidaan automatisoida, automatisoidaan. [1] Tarvittavien skriptien kirjoittamiseen ja koontiympäristön konfigurointiin menee työtunteja, mutta työaika voitetaan takaisin, koska mitään ei tarvitse tehdä kahdesti. Pienemmissä projekteissa kaikki automatisointi voidaan suorittaa aina tietovaraston päivittyessä, mutta suuremmissa projekteissa vaiheita voidaan ajastaa yön aikana tapahtuviksi. Ajastetulla integraatiolla voidaan siis jakaa jatkuvan integraation resurssitaakkaa esimerkiksi raskaimpien testien kohdalla.

Jatkuvassa integraatiossa ohjelmoija työstää omaa osuuttaan projektista. Tämän jälkeen tietovarastosta päivitetään muiden tekemät muutokset ja varmistetaan ohjelmiston toiminta. Mahdollisten yhteensopivuusvirheiden korjaamisen jälkeen ohjelmoija lisää oman tuotoksensa osaksi tietovarastoa. Tässä vaiheessa koontipalvelin herää ja kokoaa koko projektin vielä kerran. Tähän sisältyy yleensä myös automatisoitu testaus. Ohjelmistokehitysryhmä saa välittömästi palautteen mahdollisista ongelmista, joiden korjaamisesta tulee välitön toimenpide. Tietovarastosta löytyy siis aina toimiva ja integroitu ohjelmistotuote.

3 VERSIONHALLINTA JA KOONTI

Tämän insinöörityön lähtökohtana oli automatisoidun koonti- ja testausjärjestelmän kehittäminen Metropolia Ammattikorkeakoulun ohjelmistotekniikan opiskelijoille. Vaikka tässä työssä keskitytään testaamiseen, on oleellista käydä läpi myös järjestelmän muita osia, jotta työ toimii myös itsenäisenä kokonaisuutena. Aleksi Lukkarinen on kirjoittanut tämän insinöörityön valmistumisen aikana omaa insinöörityötään, joka käsittelee samaa järjestelmää laajemmin yleisellä tasolla.

Mahdollisimman kattavaa koonti- ja testausjärjestelmää voidaan havainnollistaa alkuperäisellä suunnitelmakaaviolla (kuva 1).



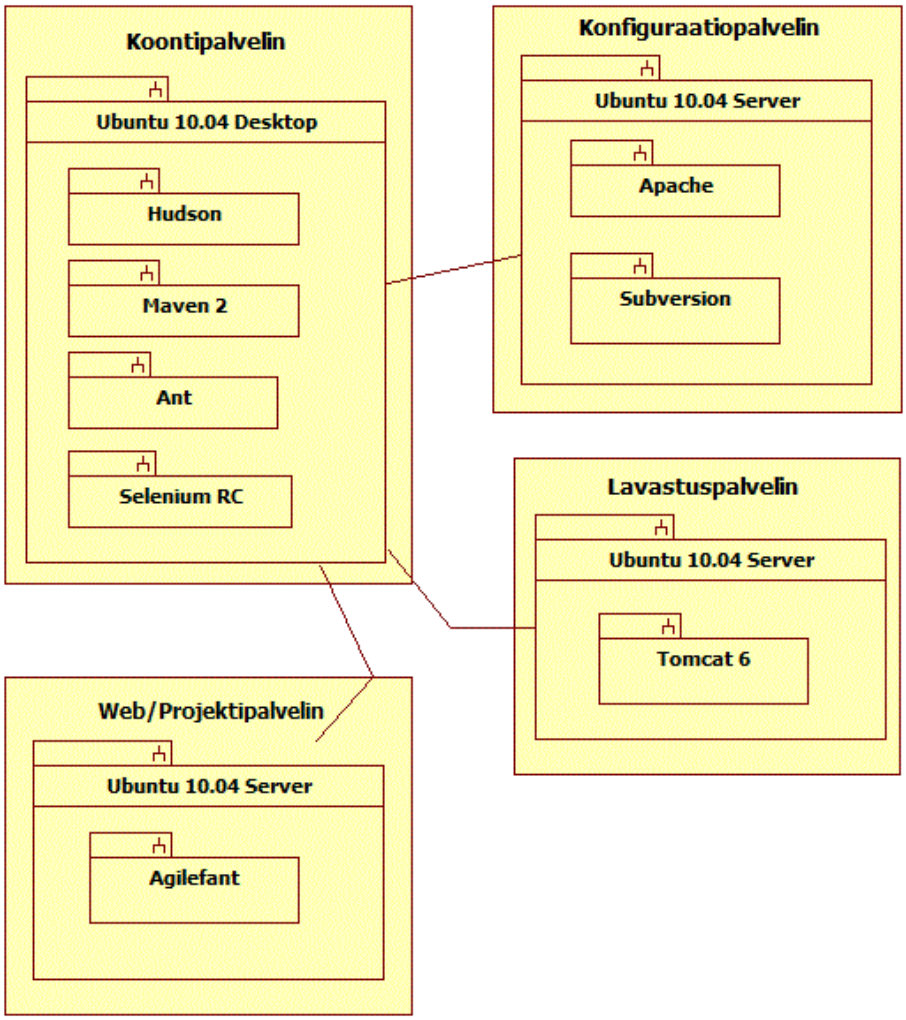
Kuva 1. Alkuperäinen ehdotus koonti- ja testausjärjestelmäksi

Kuva esittää alkuperäistä suunnitelmaa kattavasta koonti- ja testausjärjestelmästä Metropolian ohjelmistotuotantoprojekteille. Palvelimien yhteydessä on ehdotelmia työkaluista, joita voisi kyseisillä palvelimilla hyödyntää. Näiden työkalujen välisiä vertailuja käydään tarkemmin läpi aiheeseen liittyvien lukujen yhteydessä.

Alkuperäisessä suunnitelmassa oli mukana koontipalvelin, johon suunniteltiin koontityökaluja myös PHP-projektien koontia varten. Koontipalvelin kuuntelee konfiguraatiopalvelinta (versionhallintapalvelin), josta haetaan ohjelman tuorein versio koontia- ja testausta varten. Projektin sovellus käynnistetään väliaikaisesti testauspalvelimella hyväksymistestausta varten. Moduulipalvelimelle suunniteltiin ohjelmia (Artifactory tai Nexus), jotka tallentavat koonninaikana tarpeelliset paketit talteen, jotta niitä ei tarvitsisi ladata jokaiselle projektille uudestaan. Hallintapalvelin olisi hallinnoinut kommunikaatiota palvelimien välillä ja olisi toiminut keskitettynä skriptikirjastona. Lokipalvelin olisi keskittänyt järjestelmän lokitiedostot. Web-projektipalvelin on projektei-

hin liittyvien julkisten tiedostojen esittelypalvelin, ja siellä pidetään kirjaa työmääristä Agilefant-ohjelmalla.

Tätä kirjoittaessa moduuli-, hallinta- ja lokipalvelin eivät olleet osana lopullista järjestelmää. Ne jäävät jatkokehitysmahdollisuuksiksi. Lopullinen ohjelmistotuotantoprojekteille aikaan saatu palvelinratkaisu on nähtävissä kuvasta 2, joka on yksinkertaistettu esitys todellisesta järjestelmästä.



Kuva 2. Lopullinen koonti- ja testausjärjestelmä

Kuvan 2 esittämää järjestelmää käydään yksityiskohtaisesti läpi tämän työn edetessä. Samalla esitellään minimijärjestelmä kyseisessä vaiheessa. Kuvasta 1 haetaan vaihtoehtoisia ratkaisuja työkaluvalinnoille, kun perustellaan lopullisia päätöksiä. Kaikilla palvelimilla koontipalvelinta lukuun ottamatta oli käyttöjärjestelmänä Ubuntu 10.04 Server. Koontipalvelimille asennettiin

Ubuntu 10.04 Desktop, koska graafinen käyttöliittymä on oleellinen verkkoselainten käynnistämiseksi testien ajaksi.

Keskeisin kuvissa 1 ja 2 esiintyvistä palvelimista on koontipalvelin. Kuvassa 1 esitetään erilaisia ohjelmistoratkaisuja, joita palvelimelle voisi asentaa.

Testien ajaminen kulkee yleensä käsi kädessä koontiprosessin kanssa. Koonnin läpi meneminen osoittaa ohjelman ajokelpoiseksi, ja testien läpäiseminen vahvistaa sen, että ohjelma tekee sitä, mitä sen oletetaan tekevän. Testien automatisointi osaksi koontiprosessia on normaali käytäntö jatkuvassa integroinnissa. Siksi on oleellista käydä läpi myös koontia testauksen lisäksi, erityisesti, kun otetaan huomioon, että testien ajaminen aktivoidaan usein samoilla työkaluilla kuin koonti.

Tässä luvussa käydään läpi jatkuvan integroinnin työkaluja, joilla koonti aktivoidaan automaattisesti. Samalla otetaan kantaa asennusprosessissa ilmeneviin mahdollisiin sudenkuoppiin. Tämän jälkeen valitaan koontityökalut, joilla koonti suoritetaan ja testaustyökalut saadaan aktivoitua. Kolmanneksi otetaan kantaa versionhallintaohjelmiston valintaan ja lopuksi käydään läpi koontityökalun kannalta oleellisen asetustiedoston rakenne.

3.1 Jatkuvan integroinnin työkalut

Koontipalvelimen selkärangaksi valittiin Hudson [2], jonka uusimmat versiot tunnetaan nykyään nimellä Jenkins, koska nimi vaihtui tämän insinöörityön kirjoittamisen aikana. Ohjelmaan viitataan jatkossa nimellä Hudson. Koonti voidaan aktivoida Hudsonin verkkokäyttöliittymän kautta manuaalisesti, mutta jatkuvassa integraatiossa se yleensä tarkkailee versionhallinnassa tapahtuvia muutoksia. Hudson ei itsessään toteuta koontia, joten se tarvitsee avukseen esim. Antin [3] tai Mavenin [4], joista molemmat asennettiin koontipalvelimelle.

Hudson on Java-pohjainen verkkosovellus, joten se vaatii WAR-pakkauksia ajavan palvelinohjelmiston alleen. Hudsonin asentamiseen ja käyttöönottoon on monia eri tapoja, mutta kokeiden tuloksena todettiin, että ongelmia ilmenee helposti erityisesti päivitysten kohdalla.

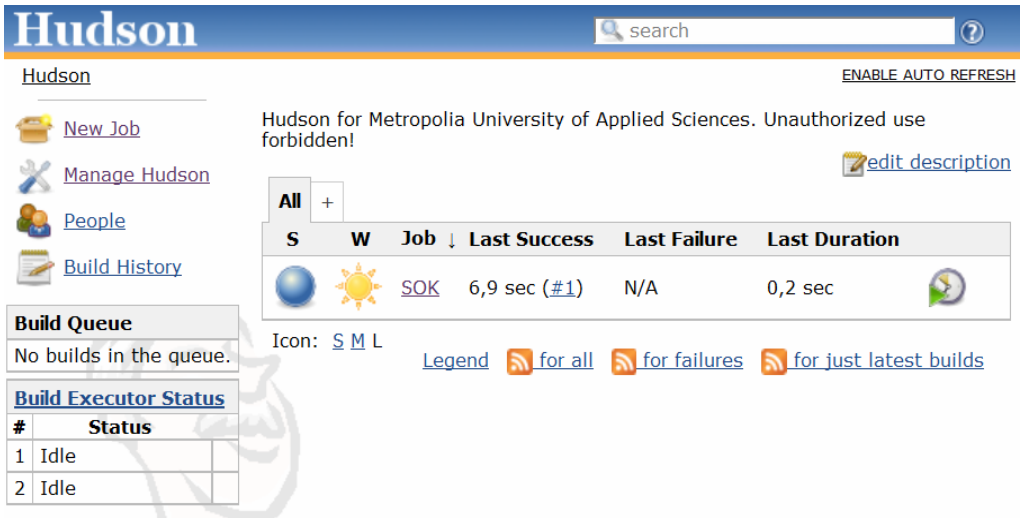
Ensimmäinen asennus tehtiin Tomcat 6:n alle. Alussa oli epävarmaa, onko koontipalvelimella tarvetta ajaa muita verkkosovelluksia. Kaikki olisi voitu ajaa yhdeltä palvelimelta. Ratkaisussa Hudson asennettiin Tomcatin ROOT-sovellukseksi korvaamaan oletuksena asentunut hallintasivusto. Ongelmaksi nyt ja myös myöhemmin muodostui koulun verkkoliikenteeseen asetetut rajoitukset. Tomcatin oletusportti on 8080, jonka kautta kulkeva liikenne pysähtyy palomuureihin. Ainoa taatusti toimiva portti oli http-liikenteelle tarkoitettu 80. Ubuntu 10:ssä ainoastaan root-käyttäjällä on oikeus käynnistää porttia 80 kuunteleva prosessi, ja Tomcatin käynnistämistä roottina pidetään yleisesti huonona toimintatapana [5;6]. Hudson saatiin tällä tavoin käynnistymään. Tomcat-ratkaisua hankaloitti kuitenkin se, että koko ohjelman päivitys tulisi tehdä itsekirjoitetuilla shell-skripteillä poistamalla vanha Hudson ja lataamalla uusi tilalle. Lopullinen ratkaisu hylätä Tomcat saavutettiin, kun Hudsonin liitännäisten asentaminen ja päivittäminen eivät toimineet odotetusti. Liitännäislistojen hakeminen ei toiminut aina ollenkaan. Hudson ajettuna Tomcat 6:n alla on liian ongelmallinen ja epävarma ratkaisu. Todettiin myös, että koontipalvelin ei välttämättä aja muita verkkosovelluksia, jolloin perusteet jatkaa tällä ratkaisulla olivat vähäiset.

Hudsoniin on upotettu Winstone -palvelin. Käynnistäminen onnistuu siis ilman ulkopuolista palvelinta. Tietokoneessa, johon Java on asennettu, Hudsonin voi käynnistää yksinkertaisesti komennolla ”Java -jar hudson.war”, missä hudson.war on oletettu tiedostonimi Hudsonin ajettavalle paketille. Teoriassa yksinkertainen ratkaisu hankaloituu, kun Hudson tulisi päivittää. Jälleen ainoa mahdollisuus on skriptata uuden paketin haku vanhan tilalle. Kolmannen paremman käyttöönottotavan ollessa mahdollinen tämä tapa hylättiin hankalana.

Ubuntu 10.04 voi asentaa ohjelmia debian-pakkauksista. Hudsonille on tehty debian-julkaisu, jonka avulla päivittäminenkin on helppoa, kun asetetaan Hudsonin tietovaraston avain (repository key) käyttöön komennoilla ”wget -O /tmp/key http://hudson-ci.org/debian/hudson-ci.org.key” ja ”sudo apt-key add /tmp/key”. Asennus suoritetaan komennolla ”sudo dpkg --install hudson.deb”, missä hudson.deb on Hudsonin asennustiedosto. Tämän jälkeen päivitys tapahtuu aina komennoilla ”apt-get update” ja ”apt-get install hudson”.

Seuraava ylitettävä ongelma olikin aiemmin mainitut suljetut portit. Sisäänrakennetun Winstone-palvelimen oletusportti on sama kuin Tomcatissa, eli 8080. Hudson ei tällä asennuksella voi käyttää porttia 80, vaikka sen yrittäisi määritellä asetustiedostoihin. Ongelman kiertäminen oli kuitenkin mahdollista parilla tavalla. Portiksi määriteltiin 8081, joka jättää yleisen oletusportin 8080 vapaaksi varmuuden vuoksi. Ensimmäinen ratkaisu on asentaa Apache-palvelin kuuntelemaan porttia 80 ja välittämään liikenne porttiin 8081. Toinen helpompi ja valittu ratkaisu on käyttää iptables-palomuuriohjelmaa. Asetetaan sääntö, jossa kaikki 80 porttiin tulevat kyselyt välitetään porttiin 8081, jota Hudson kuuntelee.

Kuten kuvassa 1 huomataan, on Hudsonille mainittu vaihtoehtona ”Cruise Control” -niminen ohjelma. Se on täysin toimiva ohjelma, jolla pystyy samaan kuin Hudsonilla. Suurena etuna Hudsonilla on kuitenkin siisti ja helppokäyttöinen webkäyttöliittymä töiden käsittelyyn (kuva 3).



Kuva 3. Hudsonin käyttöliittymän etusivu

Kuvan käyttöliittymästä näkee heti aloitussivulta projektin tilan. Sininen pallo kertoo että edellinen koonti onnistui. Aurinko on säätilakuvake, joka muuttuu projektin yleisen vakauden heikentyessä puolipilviseksi ja siitä edelleen ukkosmyrskyksi.

Cruise Control on enemmän XML-muotoisiin asetustiedostoihin pohjautuva järjestelmä. Opiskelijoiden tulisi kuitenkin oppia hyödyntämään annettuja työkaluja varsin lyhyessä ajassa, joilla kriteereillä Hudson on huolettomampi vaihtoehto.

3.2 Koontityökalut

Java-projektien koontityökaluista yleisimpiä ovat Apache Software Foundation Maven ja Ant. Mavenin etuna on yksinkertaisempi käyttöönotto, kun taas Ant on joustavampi erityisvaatimusten alla.

Mavenin keskeisiin ominaisuuksiin kuuluu kyky hallinnoida koontia, dokumentointia ja testausta yhden keskeisen tietolähteen perusteella. Tietovarastoon luodaan XML-muotoinen pom.xml-tiedosto, missä määritellään Java-projektin riippuvuudet ja kirjastot. Mavenilla koonnin lopputuloksena on esim. JAR- tai WAR-paketti sekä mahdolliset dokumentit, joihin voi kuulua Javadoc ja testausraportit riippuen pom.xml:ään määritellyistä asetuksista ja elinkaaren (lifecycle) päätepisteen valinnasta.

Mavenin elinkaari määrittelee koontiprosessin vaiheisiin. Kaaria on kolme. Ensimmäinen puhdistuselinkaari (clean lifecycle) koostuu vain kolmesta vaiheesta: *pre-cleanista, cleanista, post-cleanista*. Esisiivous (pre-clean) valmistelee siivousprosessia varten. Varsinainen siivous (clean) toteuttaa sen, eli poistaa edellisen käännöksen tuotokset, ja jälkisiivous (post-clean) viimeistelee siivousprosessin.

Toinen kaari on oletuselämänkaari (default lifecycle), joka on laajin kaikista ulottuen koonnin ja pakkaamisen kautta testaukseen sekä pakkauksen käyttöönottoon. Kaikki 23 vaihetta on listattu seuraavaan taulukkoon.

Taulukko 1. Mavenin oletuselinkaari, eli default lifecycle

validate	Varmistaa että projekti on oikea ja että kaikki tarvittava tieto on saatavilla.
initialize	Alustaa koonnin esimerkiksi asettamalla asetuksia tai luomalla hakemistoja.
generate-sources	Generoi lähdekoodit jotka sisällytetään käännökseen (compile).
process-sources	Käsittelee lähdekoodia esim. päivitystiedostojen (patch files) suorittaminen.
generate-resources	Generoi tarvittavat resurssit sisältymään pakkaukseen.
process-resources	Käsittelee ja kopioi tarvittavat resurssit kohdehakemistoon valmiina pakattavaksi.
compile	Kääntää lähdekoodin.

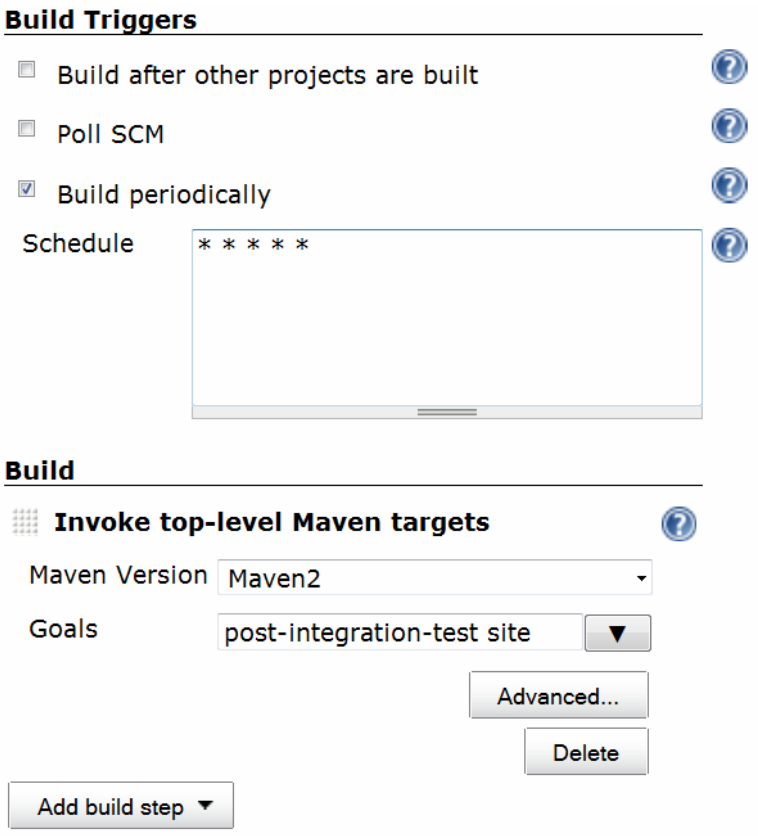
process-classes	Jälkikäsittelee käännöksen luomat tiedostot.
generate-test-sources	Generoi testien lähdekoodit sisältymään niiden käännökseen.
process-test-sources	Käsittelee testien lähdekoodit.
generate-test-resources	Generoi testiresurssit.
process-test-resources	Käsittelee ja kopioi testien lähdekoodin kohdehakemistoonsa.
test-compile	Kääntää testien lähdekoodin.
process-test-classes	Jälkikäsittelee testiluokkien käännöksen luomat tiedostot.
test	Yksikkötestien ajaminen. Näiden testien ei tulisi vielä vaatia ohjelman pakkausta eikä käyttöönottoa.
prepare-package	Suorittaa tarvittavat operaatiot pakkaamista valmistellessa.
package	Pakkaa käännetyn koodin jaettavaa muotoon, kuten esim. JAR- tai WAR-pakettiin.
pre-integration-test	Suorittaa tarvittavat toimenpiteet, kun valmistellaan integraatiotestausta. Tässä vaiheessa yleensä pystytetään tarvittava ympäristö, missä ko. testit voidaan ajaa.
integration-test	Ajetaan integraatio-testit. Joskus ohjelma käynnistetään vasta tässä vaiheessa ennen testejä.
post-integration-test	Suorittaa integraatiotestauksen jälkitoimenpiteet, kuten esim. ohjelman- ja testiympäristön alas ajaminen.
verify	Varmistaa että pakkaus on kelpoinen (valid) ja täyttää laatukriteerit.
install	Asentaa pakkauksen paikalliseen tietovarastoon, jolloin se voidaan asettaa riippuvuudeksi muille paikallisille projekteille.
deploy	Kopioi pakkauksen etätietovarastoon, jolloin se on jaettu muiden kehittäjien kesken.

Oletuselinkaaresta näkee helposti tavallisen koontiprosessin vaiheet. Maven-koonnin käynnistyksessä määritellään, mihin vaiheeseen asti koontia suoritetaan ja pom.xml:ssä voidaan asettaa jotkut vaiheet ohitettaviksi. Tällöin käynnistysparametrilla *package* käännetään, yksikkötestataan ja pakataan ohjelma, mutta integraatiotestaus ja sen jälkeen tulevat vaiheet jäävät pois.

Koonnin ohessa syntyy usein dokumentaatiota, minkä kokoamista varten Mavenissa on kolmas elämänkaari nimeltä sivustoelinkaari (site lifecycle), jonka lopputuotteena on verkkosivusto, missä on hakemisto eri raporteille. Se koostuu neljästä vaiheesta: *pre-sitesta, sitesta, post-sitesta, site-*

deploysta. Ensin käynnistetään tarvittavat prosessit sivuston luomiselle, jonka jälkeen itse sivusto luodaan. Lopuksi viimeistellään sivusto ja otetaan se käyttöön määritellyllä palvelimella.

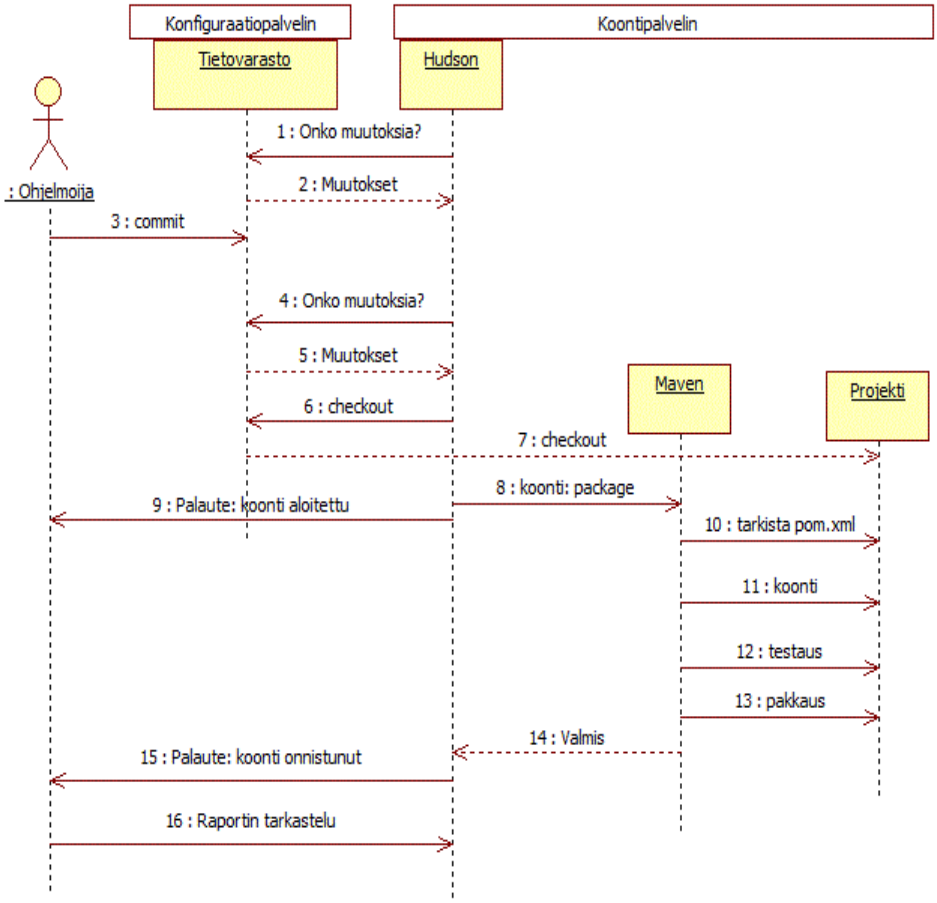
Maven 3:sta oli kirjoitushetkellä tarjolla beta-versio, jonka käyttöönottoa kokeiltiin. Se toimi koonnin osalta odotetusti, mutta siitä puuttui kokonaan raportointimoduulin (koko sivustoelinkaari) toiminta, joten lopulta päädyttiin vakaaseen kakkosversion. Mavenin valintaa edisti sen yhteensopivuus Hudsonin kanssa. Hudsonissa on itsessään pohjamalli Maven-pohjaiselle projektille, tai sen voi itse määritellä osaksi koontia asetuksista (kuva 4).



Kuva 4. Työn määrittelyä Hudsonissa

Kuvasta 4 voidaan myös huomata koontiaikataulun asetukset (Build periodically), missä hyödynnetään Unix-pohjaista ”cron” –ajastuspalvelua. Neljä tähteä tarkoittaisi sitä, että Hudson tiedustelee koonnin tarvetta kerran minuutissa.

Hudsonin ja Mavenin yhteistoimintaa järjestelmässä havainnollistetaan sekvenssidiagrammin kaltaisessa kuvassa 5.



Kuva 5. Hudsonin ja Mavenin yhteistoiminta

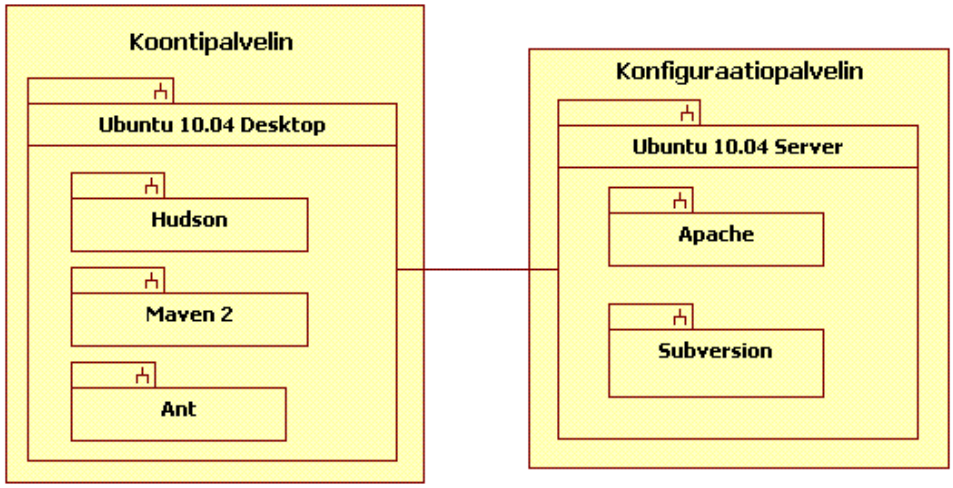
Kuvassa koontipalvelimella sijaitseva Hudson tiedustelee konfiguraatiopalvelimen tietovarastossa tapahtuvia muutoksia. Yksi ohjelmointitiimin jäsenistä siirtää uuden version ohjelmasta tietovarastoon, ja muutoksen huomatessaan Hudson lataa sen koontipalvelimelle. Tätä tuotetta esitetään kuvassa ”Projekti”-palkilla. Hudson käynnistää Mavenin, joka aloittaa koonnin, testauksen ja lopuksi pakkaa projektin. Hudson saa tiedon, että koonti on valmis, jolloin se antaa verkkokäyttöliittymässä palautteen prosessista.

3.3 Versionhallintatyökalut

Koonti- ja testausjärjestelmän koontipalvelin on yhteydessä konfiguraatiopalvelimeen (kuva 2). Helpommin ymmärrettävä nimi voisi olla versionhallintapalvelin. Palvelin sisältää tietovarastoja, joissa säilytetään ohjelmistokoodia ja dokumentoinnin tuotteita.

Versionhallintaohjelmistoksi valittiin Subversion [7]. Ratkaisu on helppo perustella sillä, että Subversion on kirjoitushetkellä Metropolian ohjelmistotekniikan opiskelijoille ennestään tuttu, koska kyseisen ohjelman käyttö kuuluu koulun opetukseen. Vaihtoehtona on mainittu Mercurial [8], mutta se poikkeaa Subversionista mm. vähemmän keskitetyllä tietovarastollaan, joka voi aiheuttaa hämmennystä vasta-alkajille [9]. Uuden järjestelmän opettelu ei ole tässä tapauksessa kannattavaa.

Nyt voidaan tiivistää luvun 3 tilanne tarkastelemalla kuvassa 6 esiteltyä järjestelmän ensimmäistä välivaihetta, joka on minimikokonaisuus, millä tähän mennessä käydyt asiat on toteutettavissa.



Kuva 6. Palvelinratkaisun ensimmäinen välimuoto

Konfiguraatiopalvelin on lopullisessa muodossaan, missä Subversion on asennettuna. Apachen http-palvelin mahdollistaa laajennetun http-protokollan hyödyntämisen versionhallintaan yhteyttä otettaessa. Koontipalvelimella muutoksia kuuntelee Hudson, jolla koonnin voi suorittaa hyödyntämällä Mavenia, Antia tai molempia. Lisäyksiä koontipalvelimen kokoonpanoon tulee luvussa 4.7.2, missä käydään läpi verkkosovellustestausta.

3.4 Pom.xml:n rakenne

Mavenin toimintaa ohjataan pom.xml-tiedoston kautta. Tähän tiedostoon viitataan monesti tämän työn aikana, joten on oleellista käydä läpi sen perusrakenne. Kokonainen esimerkkitiedosto löytyy työn lopusta liitteenä.

Kaikki tiedot menevät *project*-määrityksen sisään, jonka perään liitetään projektin tiedot (käydään läpi kohta). Loput määritykset jakautuvat osioihin *dependencies, build* ja *reporting.* Seuraava esimerkki toimii pohjana:

```
<?xml version="1.0" encoding="UTF-8"?>
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
<dependencies> </dependencies>
<build> <build>
<reporting> </reporting>
</project>
```

Ennen kuin riippuvuudet määritellään *dependencies*-osiossa, tulee kertoa, minkä projektin pom.xml on kyseessä. Määritellään *modelVersion,* joka kertoo pom.xml-mallin. Ohjelmistoilla, joilla on julkinen tietovarasto, hakemistopolku määräytyy ryhmän- ja projektin nimen sekä versionumeron mukaan. Ne määritellään *groupId-, artifactId-* ja *version*-määritysten mukaan. Javaprojekteissa ohjelmisto pakataan koonnin yhteydessä, joten pakkauksen tyyppi määritellään kohdassa *packaging.* Normaalisovellus on *jar* ja verkkosovellus ***war***. Lopuksi projektille määritellään vapaamuotoisempi nimi (*name*), jossa ei tarvitse ajatella hakemistopolun kirjoittamista. Seuraava esimerkki havainnollistaa määrittelyjen täyttämistä:

```
<modelVersion>4.0.0</modelVersion>
<groupId>ohtu.k2010.ryhma3</groupId>
<artifactId>uusi-projekti</artifactId>
<version>1.0</version>
<packaging>war</packaging>
<name>Uusi Projekti</name>
```

Edellisen esimerkin tietovaraston hakemistopolku olisi standardien mukaan $M2_REPO/ohtu/k2010/ryhma3/uusi-projekti/1.0. Tämä ei kuitenkaan Metropolian ohjelmistotuotantoprojekteissa ole erityisen oleellista.

Projektiin liittyvät riippuvuudet vaihtelevat sen mukaan, mitä tekniikoita siinä hyödynnetään, jolloin *dependencies*-osio koostuu useammasta *dependency*-määrityksestä. Riippuvuudesta kerrotaan samoja asioita, joita määriteltiin edellisessä esimerkissä omalle projektille.

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.8.0</version>
  <scope>test</scope>
</dependency>
```

Esimerkissä, missä määriteltiin yksikkötestauskehys JUnit osaksi projektia, on yksi uusi määritys: *scope.* Se määrää sen, missä vaiheessa ja millä tavalla riippuvuus on oleellinen. Mahdollisia arvoja ovat:

- Compile – riippuvuutta tarvitaan koonnin aikana (oletus).
- Provided – oletetaan että esim. JDK hankkii riippuvuuden ajon aikana.
- Runtime – riippuvuutta tarvitaan vain ajon aikana, mutta ei koonnissa.
- System – kuin provided, mutta ohjemoijan itse tulee määritellä *system-Path*, eli missä riippuvuuden sisältämä paketti sijaitsee.
- Test – riippuvuus on oleellinen vain testausvaiheessa.

Koonnin ja dynaamisen testauksen lisäresurssit sekä niiden toimintatapa määritellään *build*-osiossa. Se sisältää pääasiassa liitännäisiä (*plugin*), jotka kootaan *plugins*-kokoelmaan. Seuraava esimerkki havainnollistaa liitännäisen käyttöönoton:

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>2.0.2</version>
    <configuration>
      <source>1.5</source>
      <target>1.5</target>
    </configuration>
  </plugin>
</plugins>
```

Esimerkki esittää koontiliitännäisen käyttöönoton. Liitännäiset vaativat yleensä myös ohjeistusta, milloin ja miten ne ajetaan, mutta koontiliitännäinen on poikkeus. Näistä on parempia esimerkkejä myöhemmissä testauskappaleissa.

Raporttiosio *reporting* toimii liitännäisten aktivointialustana samankaltaisesti kuin *build*. Siellä vaikutetaan Mavenin tuottamiin raportteihin esim. aktivoimalla Javadoc- tai staattisen testauksen CheckStyle-liitännäisen. Seuraava esimerkki havainnollistaa liitännäisen käyttöönoton:

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-javadoc-plugin</artifactId>
    <version>2.7</version>
    <configuration>
      <!-- Ei asetuksia -->
    </configuration>
  </plugin>
</plugins>
```

Esimerkki aktivoisi Javadoc-dokumentaation luonnin muun dokumentaation yhteydessä. Koska liitännäisiä voi aktivoida sekä *build*- että *reporting*-osioissa, tulee ohjelmoijan varmistaa liitännäisen dokumentaatiosta, kumpaan osioon tarvittava liitännäinen menee. Osa liitännäisistä toimii molemmissa osioissa, mutta käyttäytyminen muuttuu valinnan mukaan.

4 JAVA-SOVELLUSTEN TESTAUSTYÖKALUT

Java-sovellusten testaukseen on olemassa monenlaisia työkaluja. Näistä työkaluista osa ei toimi hyvin jatkuvan integraation järjestelmässä, joten valikointia tulisi suorittaa.

Tässä luvussa on etsitty avoimeen lähdekoodiin perustuvia ratkaisuja, jotka toimivat automatisoidusssa Maven-projektissa. Valinnoissa on myös huomioitu Metropolian ohjelmistotuotantoprojektin opiskelijoiden tekninen osaami-

nen ja kurssin aikataulu. Toimiville ratkaisuille on annettu myös käytännön ohjeistus.

Luvun ensimmäinen osio käsittelee staattista testausta, missä kirjoitettua koodia analysoidaan, ilman, että ohjelmaa ajetaan. Loput luvut kuuluvat dynaamisen testauksen piiriin.

Dynaaminen testaus poikkeaa staattisesta analyysistä siten, että nyt ohjelmakoodia tulee myös ajaa, tai ohjelman pitää olla käynnissä testejä varten. Dynaamisissa testeissä tarkastellaan, että kirjoitettu koodi toimii oikein.

Kattavan dynaamisen testauksen automatisoinnin tärkeyttä ei voi aliarvioida, sillä yhdessä paikassa tehty muutos voi rikkoa jotain kohteessa, mitä ohjelmoija ei itse tule edes ajatelleeksi. Tämä aiheuttaa mahdollisesti mittavia taloudellisia vahinkoja ja paljon enemmän tuhlattua aikaa ongelmaa ratkoessa jälkikäteen.

4.1 Staattinen testaus

Staattinen testaus, joka tunnetaan myös staattisena analyysina, on ohjelmakoodin tutkimista sitä suorittamatta. Koodin automatisoituun tarkasteluun on monia työkaluja, mutta staattisen testauksen piiriin lasketaan usein myös manuaalinen koodin ja dokumenttien tarkastelu. Tässä työssä otetaan kantaa vain automatisoituun staattiseen testaukseen.

Staattisen testauksen työkalu voi tehdä monia eri asioita, olettaen, että koodia ei suoriteta. Ohjelman syntaksia voidaan tarkkailla antaen ohjelmoijalle huomautuksia huonoista ohjelmointityyleistä, jotka yksinkertaisimmillaan voisivat tarkoittaa liian pitkää yhtenäistä koodiriviä (hankaloittaa lähdekoodin lukemista tiettyjä standardeja noudattavissa tekstieditoreissa). Oleellisemmin se tarkoittaa esim. taikanumeroista (magic number) huomauttamista, jotka ovat ohjelmassa käytettäviä nimeämättömiä vakioita (lukujen merkitys hämärtyy ja vaikeuttaa koodin lukemista). Osa työkaluista mittaa ohjelmaa erilaisten metriikoiden (metrics) avulla, joilla selvitetään ohjelman monimutkaisuutta tai kattavuusmitoilla esimerkiksi kommentoinnin kattavuutta. Työkalu voi myös hakea kopioitua koodia huomauttaen raportissaan, jos ohjelmasta löytyy samanlainen koodinpätkä leikattuna ja liimattuna toistuvasti.

Metriikat ovat hieman ristiriitainen osa ohjelmistoja. Niiden periajatus on tuoda ohjelmistoihin laskettavuutta muiden tieteiden tavoin. Hankaluuksia kuitenkin luo ohjelmistojen omalaatuisuus, jolloin on vaikea määritellä, mitkä mitat ovat oikeasti hyödyllisiä. [10.] Ohjelman toimivuuden suhteen tärkeä metriikka on kattavuusmitoista koodikattavuus (code coverage), missä tutkitaan lähdekoodin testauksen kattavuutta. Jos lähdekoodissa on alueita, joihin testit eivät koskaan yllä, jää mahdolliset ongelmat huomioimatta. Onkin hyvä, jos jo staattisen testauksen vaiheessa pystytään puuttumaan asiaan, mutta nämä ongelmakohdat on myös mahdollista havaita mahdollisessa mutaatiotestauksessa, josta lisää luvussa 4.3.

Java-ohjelmien staattiseen testaukseen on olemassa vuodesta 2001 asti kehitetty CheckStyle [11]. Alkujaan se etsi koodista vain sommitteluvirheitä, mutta nykyään ominaisuuksia löytyy useita. Tarkistus on helppo automatisoida koonnin yhteyteen monilla eri työkaluilla, joihin lukeutuu Hudsonin ja Mavenin liitännäiset. Näiden etujen vuoksi CheckStyle valittiin osaksi automatisoitua koontijärjestelmää, ja käyttöönotto selvitettiin opiskelijoille järjestelmän dokumentaatiossa, sillä CheckStyleä ei varsinaisesti asenneta järjestelmään. Se sisältyy JAR-pakkaukseen *checkstyle-5.3-all.jar*.

CheckStyle koostuu moduulikokoelmasta, joita voi tarpeen mukaan aktivoida tai poistaa käytöstä. Tarkistuksiin kuuluu mm. seuraavaa:

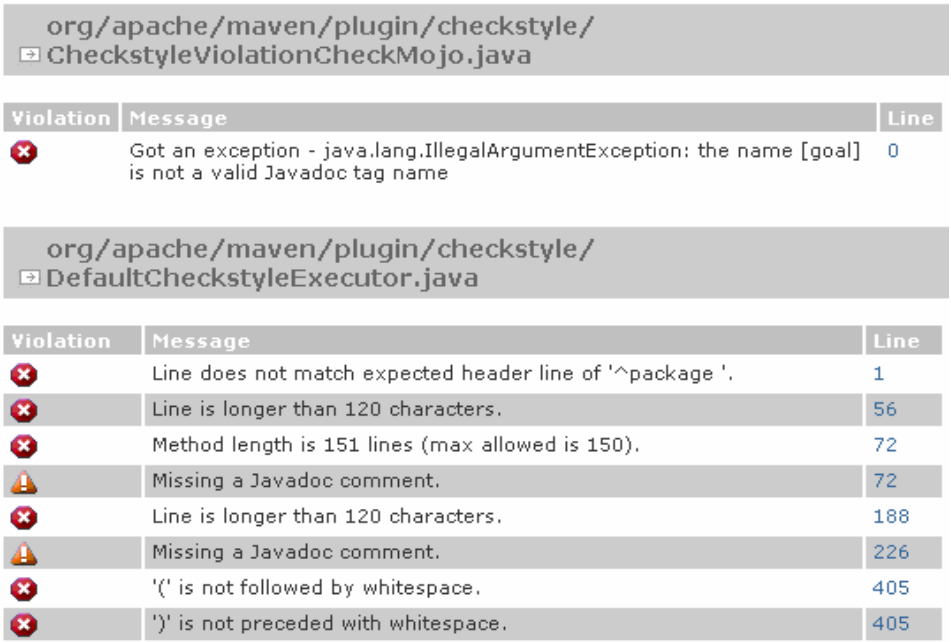
- javadoc-kommenttien olemassaolo
- metodien ja attribuuttien nimeämistyylit
- parametrien määrän ja koodirivin pituus
- pakolliset otsikot (header)
- pakettien tuonnin- (import), luokkien-, näkyvyysmuuttujien- (scope) ja opastejaksojen (instruction block) käytön tarkistus
- välilyönnit joidenkin merkkien välillä
- hyvien käytäntöjen mukainen luokkarakenne
- duplikaattikoodi
- lähdekoodin monimutkaisuutta laskevia metriikoita.

CheckStyle ajamiseen on paljon vaihtoehtoja. Ohjelmoija voi käynnistää sen halutessaan suoraan komentoriviltä, se voidaan aktivoida Hudsonista tai Antista, ja monissa kehitysympäristöissä on omat liitännäiset CheckStylelle.

Automatisoidussa koonti- ja testausjärjestelmässä CheckStylen raportti on helposti saatavilla Mavenin avulla oletusasetuksin. Projektin pom.xml –tiedostoon tulee tehdä seuraavanlainen lisäys.

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>
        maven-checkstyle-plugin
      </artifactId>
      <version>2.5</version>
    </plugin>
  </plugins>
</reporting>
```

Tämä aktivoi CheckStyle-pluginin. Kuten asetuksista huomataan, lisätään tämä aktivointi pom.xml:n raportointiosioon. Raportin saa koonnin yhteydessä, jos aktivoidaan myös sivustoelinkaari parametrilla *site*. Esimerkki raportista on nähtävissä kuvasta 7.



org/apache/maven/plugin/checkstyle/
CheckstyleViolationCheckMojo.java

Violation	Message	Line
⊗	Got an exception - java.lang.IllegalArgumentException: the name [goal] is not a valid Javadoc tag name	0

org/apache/maven/plugin/checkstyle/
DefaultCheckstyleExecutor.java

Violation	Message	Line
⊗	Line does not match expected header line of '^package '.	1
⊗	Line is longer than 120 characters.	56
⊗	Method length is 151 lines (max allowed is 150).	72
⚠	Missing a Javadoc comment.	72
⊗	Line is longer than 120 characters.	188
⚠	Missing a Javadoc comment.	226
⊗	'(' is not followed by whitespace.	405
⊗	')' is not preceded with whitespace.	405

Kuva 7. Ote CheckStyle Maven-liitännäisen luomasta raportista

Kuvan raportista nähdään pienet huomautukset merkattuna huutomerkillä. Tyyli- ja ohjelmointivirheet näkyvät rasteina.

CheckStylen asetuksia muokataan xml-muotoisella asetustiedostolla. Tämä saadaan osaksi liitännäistä lisäämällä seuraava tieto.

```
<configuration>
  <configLocation>checkstyle.xml</configLocation>
</configuration>
```

Tässä esimerkissä oletetaan, että lähdekoodin mukana toimitetaan checkstyle.xml-niminen asetustiedosto. Liitännäisen mukana toimitetaan neljä valmista asetuskokonaisuutta:

- config/sun_checks.xml
- config/maven_checks.xml
- config/turbine_checks.xml
- config/avalon_checks.xml.

Ensimmäinen on oletusasetukset sisältävä asetustiedosto. Toinen on Maven-projekteille optimoitu asetustiedosto. Kolmantena ovat turbine servlet -kehystä hyödyntäville projekteille tarkoitetut asetukset, ja viimeisenä vuonna 2004 suljettua Apache Avalon kehystä hyödyntäville projekteille. Tarkemmat tiedot niiden eroista pitää hakea itse tiedostoista, sillä niiden konkreettisia eroja ei ole dokumentoitu. [12]

CheckStyle ei yksin tee kaikkea, mutta se ei myöskään sulje pois muita työkaluja. Staattinen testaus toteutetaan toisinaan monen eri testaustyökalun yhdistelmänä, missä eri työkalut täydentävät toisiaan [13]. Esimerkkinä aiemmin mainittu koodikattavuuden tutkiminen puuttuu CheckStylesta, joten on mahdollista valita joku toinen työkalu rinnalle paikkaamaan tätä puutetta.

Käytännön esimerkkinä koodikattavuutta voidaan tutkia jcoverageen perustuvalla Coberturalla [14]. Siitä on toteutettu myös Maven-liitännäinen, jonka käyttöönotto on yhtä yksinkertaista kuin CheckStylenkin.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>cobertura-maven-plugin</artifactId>
  <version>2.4</version>
</plugin>
```

Oheiset XML-määritykset lisätään pom.xml:n reporting-osioon. Näin tehtynä liitännäinen käyttää suositeltuja oletusasetuksia.

4.2 Yksikkötestaus

Yksikkötestaus on matalan tason dynaamista testausta. Testi kohdistuu tavallisesti vain yhteen metodiin, jolloin varmistutaan, että juuri tämä pieni osa ohjelmaa toimii, kuten ohjelmoija on ajatellut. Parhaimmillaan yksikkötestin kirjoittaminen on hyvin helppoa, jos metodin parametrit ja palautusarvo ovat yksinkertaiset. Vaikeimmillaan yksikkötestaus on hyvin vaativaa työtä, kun työskennellään esimerkiksi monimutkaisempien Java Enterprise Edition (JEE) ohjelmistojen parissa. Niissä syötteet ja tulokset ovat usein XML-muotoisia ja yhdenkin metodin taustalla on sekä aikaa että laskentatehoa kuluttavia prosesseja ja valtava tietokanta.

Java-ohjelmistoilla yksikkötestauksessa suosituimpia kehyksiä ovat JUnit [15] ja TestNG [16]. Vertailtuna molemmat kehykset ovat käytössä hyvin samankaltaisia, varsinkin JUnit 4:n lisätessä huomautuksiin (annotation) pohjautuvan tavan hallita testien ajamista. Ominaisuuksia tarkastellessa jotkut ovat kallistuneet TestNG:n puolelle päätyen jopa johtopäätöksen, ettei JUnitille olisi enää tarvetta [17]. Maltillisemmat mielipiteet [18] taas väittävät, että molemmille on paikkansa, ja niitä voisi käyttää rinnakkain. Yleisesti TestNG toimii hyvin korkeamman tason testeissä sekä suuremmissa testikokonaisuuksissa, kun taas JUnitin pääpaino on nimenomaan yksikköjen testauksessa matalalla tasolla. Kehysten samankaltaisuuden vuoksi voidaan tämän työn puitteissa keskittyä paremmin tunnettuun JUnittiin.

Molemmat on helppo lisätä osaksi Maven-koontia Surefire-liitännäisellä, jolla voidaan myös muuttaa testaukseen liittyviä asetuksia. Surefiren saa käyttöön pom.xml:n *build*-osiossa uudella plugin-määreellä.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.7</version>
</plugin>
```

Yksikkötestien ajamiseksi riippuvuuksiin tulee lisätä myös *dependencies*-osioon JUnitin osalta seuraavanlainen määritys:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.8.1</version>
  <scope>test</scope>
</dependency>
```

TestNG:n vastaava on seuraavanlainen:

```
<dependency>
  <groupId>org.testng</groupId>
  <artifactId>testng</artifactId>
  <version>4.7</version>
  <scope>test</scope>
  <classifier>jdk15</classifier>
</dependency>
```

Huomion arvoista määrityksissä on *scope*, joka arvolla *test* määrittää, että nämä riippuvuudet ovat oleellisia vasta testausvaiheessa. Oletusarvo olisi *build*, jolloin koonnissa oletettaisiin yksikkötestauksen riippuvuuksien olevan oleellisia koonnin aikana.

Yksikkötestien kirjoittaminen kohdistaa testin puhtaimmillaan vain yhteen luokkaan. Tämä voi muodostua ongelmaksi, sillä monissa luokissa on viitteitä muihinkin luokkiin. Tällöin yksikkötestistä tulee raskaampi, ja sen voidaan jopa ajatella kuuluvan integraatiotestaukseen. Erityisen kriittisiä ovat luokat, joilla on riippuvuus tietokantaan. Jatkuva yhteys tietokantaan hidastaisi testikokonaisuuden ajamista, tai mahdollisesti aiheuttaisi vääriä hälytyksiä, kun tietokannassa ilmenee ongelma, eikä suinkaan sitä kutsuvassa luokassa.

Ratkaisu voi löytyä matkijaolioista (mock object). Niiden tarkoitus on vastata toisen luokan tekemään kutsuun kuten oikea olio, kuitenkaan toteuttamatta olion oikeaa toiminnallisuutta. Ne tulevat erityisen tärkeiksi testivetoisessa kehityksessä (Test-driven development, TDD), missä testien tulee mennä läpi monesti paljon aikaisemmin, kuin tarpeellisia luokkia on edes toteutettu. TDD perustuu ideologiaan, missä testit kirjoitetaan aina ennen varsinaista toteutusta.

Yksi tapa matkijaolioiden käyttöönottoon on jMock [19]. Se on Java-kirjasto, joka helpottaa matkijaolioiden luontia suoraan lennosta, ilman, että ohjelmoijan tarvitsisi itse kirjoittaa mitään yksikkötestin ulkopuolella. Seuraava esimerkki havainnollistaa matkijaolion käyttöä tilanteessa, missä *Kurssi*-luokan metodi laskee kurssille osallistuneiden jäsenmaksut yhteen.

```
public int laskeTuotto(){
  final int kurssimaksu = 20;
  return kurssimaksu * dao.haeOsallistujat();
}
```

Maksun suuruus osallistujaa kohden on vakio 20 €, mutta osallistuneiden määrän luokka hakee DAO:n kautta tietokannasta. Luodaan siis DAO:sta matkijaolio välttääksemme tietokantayhteyden.

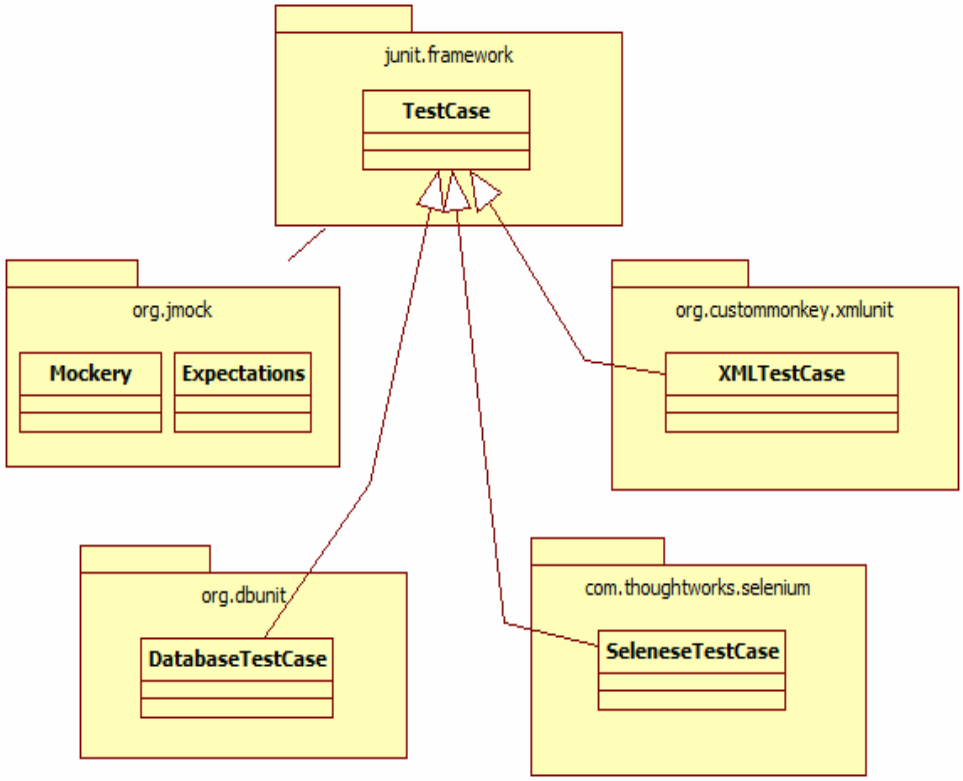
```
import org.jmock.*

@RunWith(JMock.class)
public class KurssiTest extends TestCase{
  Mockery context = new JUnit4Mockery();
  @Test
  public void testaaKurssinTuotonLasku(){
    final DAO dao = context.mock(DAO.class);
    context.checking(new Expectations() {{
      oneOf(DAO).haeOsallistujat();
      will(returnValue(5));
    }});
    Kurssi kurssi = new Kurssi();
    assertEquals(kurssi.laskeTuotto(), 100);
  }
}
```

Esimerkissä *DAO*-luokan matkijaolio *dao* palauttaa arvon 5, kun kutsutaan metodia haeOsallistujat(). DAO saa pyynnön *Kurssi*-luokan laskeTuotto()-metodin aikana. Silloin *kurssi* saa tietokantayhteyden sijaan valmiiksi määritellyn arvon. Metodin toiminnan kannalta tällä ”huijauksella” ei ole mitään merkitystä, mutta tiedetään jonkin olevan pielessä jos 5*20 ei olekaan 100.

Jotta matkijaoliot toimisivat Mavenin kanssa, tulee pom.xml:ään lisätä riippuvuus *dependencies*-osioon.

```
<dependency>
  <groupId>org.jmock</groupId>
  <artifactId>jmock-junit4</artifactId>
  <version>2.5.1</version>
</dependency>
```

JUnitia on laajennettu muihinkin erikoistuneisiin tarkoituksiin, joita käsitellään seuraavissa luvuissa. Kuva 8 esittää UML-kaavion JUnitin laajennuksista.



Kuva 8. JUnit TestCase ja sen laajennukset

JMock ei ole suora JUnitin laajennus, mutta se tekee yhteistyötä JUnitin kanssa. *XMLTestCase* laajentaa JUnitista XML-testaukseen soveltuvamman. *DatabaseTestCase* yksinkertaistaa tietokantatestausta. *SeleneseTestCase* mahdollistaa Selenium-hyväksymistestien ajamisen JUnitin kautta.

4.3 Mutaatiotestaus

Hyvien yksikkötestien kirjoittaminen voi olla vaativaa työtä. Monesti kirjoitettujen testien oletetaan olevan hyviä ja kattavia, eikä parantamisen varaa ole, vaikka asia ei näin olisikaan. ”Kuka siis vartioi vartioitamme?”

Mutaatiotestaus auttaa löytämään heikkouksia jo kirjoitetuista testeistä. Siinä testataan testejä tekemällä lähdekoodiin pieniä muutoksia, eli luomalla alkuperäisestä koodista kokoelma mutantteja. Muutokset kohdistuvat yksinkertaisimmillaan vertailuihin, esimerkiksi vaihtamalla suuruusvertailu päinvastaiseksi. Mutaatiotesti on hyväksytty, jos mutanttikoodi ei mene läpi. Tilanne, jossa kaikki yksikkötestit menevät läpi mutaation jälkeen, koetaan osoittavan yhtä seuraavista heikkouksista:

- Osa koodista on turhaa, sillä sitä ei ajeta ohjelman aikana.
- Kyseistä osaa lähdekoodista ei testata lainkaan.
- Yksikkötestien tarkkuudessa on parantamisen varaa.

Mutaatiotestauksen huonona puolena pidetään sitä, että kun suureen ohjelmaan lisätään useita mutaatioita, prosessi vaatii useita koonteja ja sen myötä testaukseen kulutettu aika on suuri. Tämä ongelma on kuitenkin nykytekniikalla huomattavasti pienempi suhteutettuna modernien työasemien ja -palvelimien laskentatehoon. [20] Kouluympäristössä laskentaresurssien määrä on kuitenkin rajallinen. Mutaatiotestauksen hyödyllisyys voidaan kyseenalaistaa koulun ohjelmistotuotantoprojekteissa myös siksi, että osa mutaatiotestauksen eduista saavutetaan jo staattisella testauksella.

Kattavien avoimeen lähdekoodiin perustuvien ratkaisujen puute hankaloittaa mutaatiotestausta. Varteenotettaviksi vaihtoehdoiksi nousee vain Jumble [21] ja Javalanche [22]. Näistä Jumble on pidemmälle kehitetty, mutta heikosti automatisoitu. Sitä voi käyttää komentorivin kautta tai Eclipse-liitännäisenä. Jumblen Maven-liitännäinen ei ole saavuttanut ensimmäistäkään julkaisua ja on jäänyt esi-alpha-asteelle. Javalanche taasen on keskeneräinen. Siinä on keskitytty enemmän automatisointiin Antia hyödyntäen. Sekin vaatii melkoisesti alkutoimenpiteitä [23], joten nopeaan tahtiin etenevässä projektissa hyöty jää vähäiseksi.

4.4 XML-testaus

XML-dokumentit ovat nykyisin hyvin laajalti levinneitä. Tässäkin työssä esitellyistä tekniikoista suurin osa sisältää XML:ää jossakin muodossa. Yksikkötestauksen luvussa mainittiin Java EE:n suuri XML-riippuvuus. Tieto siirtyy usein, varsinkin verkkosovelluksissa, XML-muotoisena. On siis oleellista pystyä testaamaan XML-dokumentteja käsitteleviä luokkia.

Monesti testissä ollaan kiinnostuneita metodin palauttamasta arvosta. Jos palautettava arvo on XML-dokumentti, sen vertailu perinteisin keinoin menee monimutkaiseksi. Helpotusta tarjoaa XPath-kieli, jonka avulla voidaan käydä läpi XML-muotoisen tiedon sisältöä. XPathilla on oma API:nsa, *javax.xml.xpath*, jota voi hyödyntää myös JUnit-testeissä, kun tutkitaan jäsennettyjä XML-dokumentteja. [24, s. 266] Työtä helpottamaan on kuitenkin luotu myös JUnit-laajennus nimeltä XMLUnit [25]. Sillä voi esimerkiksi vertailla XML-dokumentteja, tehdä XPath-vertailuja dokumentin osiin tai kelpoistaa (validate) XML:ää DTD-määrityksiä- tai Skeemaa (schema) vastaan.

Käyttöä voidaan havainnollistaa J.B Rainsbergerin kirjassa ”JUnit Recipes” esiintyvästä esimerkkiä mukailevalla versiolla [24, s.266, s.268]. Oletetaan, että luokka *XMLMuunnin* ottaa vastaan parametrina ihmisen etu- ja sukunimen sisältävän *Henkilo*-olion ja palauttaa siitä seuraavanlaisen XML-muunnoksen.

```
<?xml version=1.0?”>
<henkilo>
  <etunimi>Pekka</etunimi>
  <sukunimi>Peltola</sukunimi>
</henkilo>
```

Halutaan tarkistaa, onko palautetussa XML-dokumentissa solmu (node) ”henkilo” ja että nimi on ”Pekka Peltola”, joten luodaan normaalista JUnitin TestCasesta laajennettu XMLTestCase.

```
import java.io.StringWriter;
import org.custommonkey.xmlunit.XMLTestCase;

public class HenkiloXmlMuunnosTest extends XMLTestCase{
  public void testaaPekkaPeltola(){
    Henkilo henkilo = new Henkilo(“Pekka”, “Peltola”);
    XMLMuunnin muunnin = new XMLMuunnin();
    StringWriter tulos = new StringWriter();
    muunnin.muuta(henkilo, tulos);
    String xmlmerkkijono = tulos.toString();

    assertXpathExists(”/henkilo”, xmlmerkkijono);

    assertXpathEvaluatesTo(
      ”Pekka”,
      ”/henkilo/etunimi”,
      Xmlmerkkijono);

    assertXpathEvaluatesTo(
      ”Peltola”,
      ”/henkilo/sukunimi”,
      Xmlmerkkijono);
  }
}
```

Testissä *assertXpathExists* tarkistaa, löytyykö ”henkilo”-solmu ja *assertXpathEvaluatesTo* tarkistaa, ovatko etu- ja sukunimi oikeat. Usein mielikuva XML-muotoisesta tiedosta on nimenomaan ”*.xml”-päätteiset tiedostot, mutta kuten tässä esimerkissä huomataan, ei tietoa suinkaan aina tallenneta tiedostoon ennen käsittelyä.

Koska XMLUnit on jatkettua JUnittia, on sen hyödyntäminen Maven-projektissa mahdollista vain jos JUnitin vaatimat riippuvuudet toteutuvat. Ohjeet tähän löytyvät luvusta 4.2.1. Uusille pakkauksille tulee lisätä myös riippuvuus *dependencies*-osioon pom.xml:ssä.

```
<dependency>
    <groupId>xmlunit</groupId>
    <artifactId>xmlunit</artifactId>
    <version>1.3</version>
</dependency>
```

XML-testausta voidaan hyödyntää myös HTML-sivujen testaukseen. Niiden rakenne on samankaltainen, ja esimerkiksi XHTML pohjautuu nimenomaan XML-syntaksiin.

4.5 Penetraatiotestaus

Penetraatiotestaus on varsin uusi ja kehittyvä testauksen muoto, eikä se rajoitu pelkästään ohjelmistotuotteisiin vaan organisaation moniin eri osa-alueisiin. Aihetta rajaten keskitytään kuitenkin vain ohjelmistojen testaamiseen. Penetraatiotestaus on aktiivista testaamista, missä etsitään heikkoja kohtia, joita mahdollinen pahantahtoinen taho voisi hyödyntää. Aktiivinen tarkoittaa tässä tapauksessa sitä, että riskejä ei etsitä esim. staattisesti lähdekoodista vaan jo ajossa olevasta sovelluksesta. Erityisen oleellista on pitää huolta verkkosovellusten tietoturvasta, kun näkyvyys ulottuu internetiin asti. [26] Etsittäviä heikkouksia ovat mm. URL-manipulaatio, SQL-injektio, cross site skriptaus, muistissa sijaitsevat salasanat, istunnon kaappaus (session hijacking), puskurin ylivuoto (buffer overflow), verkkopalvelimen asetukset ja käyttöliittymän kaappaaminen (”clickjacking” tai UI redressing).

Penetraatiotestaus ei ole parhaassa asemassa, kun jatkuva integrointi toteutetaan avoimen lähdekoodin ohjelmistoilla. Yksittäisiä työkaluja on montaa eri sorttia [27], joista osa on myös ilmaisia, mutta suuri osa on kaupallisia työkaluja - varsinkin ne, joiden automatisointiin on panostettu (esim. Codenomiconin Defensics) [28]. Pääasiallisesti penetraatiotestit tehdään siis omassa testausvaiheessaan.

Metropolian automatisoituun koonti- ja testausjärjestelmään on vaikea suositella mitään yhtä työkalua, josta saisi automatisoituna suuremman hyödyn kuin ajatuksen kanssa testien ajamisen osana jotakin sprinttiä. Suositut tuotteet ovat myös usein puhtaasti Windows-pohjaisia, jolloin Ubuntu-pohjainen järjestelmä ei voi niitä ajaa.

Haluttuja testaustyökaluja on kuitenkin mahdollista ajaa automatisoituna, vaikka Maven-liitännäisiä ei suoraan ole tarjolla. Avuksi tarvitaan kuitenkin juuri tämänkaltaisia tilanteita varten asennettu Ant. Ant tarjoaa ajokomennon *exec*, millä voi suorittaa haluamiaan prosesseja. Mavenilla Ant-komentoja pääsee suorittamaan Antrun-liitännäisellä, joka vaatii seuraavanlaiset pom.xml –määritykset *build*-osioon.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.6</version>
  <executions></executions>
</plugin>
```

Yksiselitteisiä ohjeita prosessien käynnistämiseen on mahdoton antaa, mutta uuden ajon voi lisätä *executions*in sisään seuraavaa kuvitteellista esimerkkiä mukaillen, missä käynnistetään ohjelma ”ptestprogram”, jonka tuloste tallennetaan tekstitiedostoon. Ohjelmalle annetaan myös parametri ”/scan http://1.0.0.1/index.html”.

```
<executions>
  <execution>
    <phase>integration-test</phase>
      <configuration>
        <tasks name="Penetraatio testi">
           <exec
             dir="${basedir}"
             executable="${basedir}/ptestprogram "
             output="${basedir}/t/tulos.txt">
             <arg value="/scan"/>
             <arg value="http://1.0.0.1/index.html"/>
           </exec>
        </tasks>
       </configuration>
       <goals>
         <goal>run</goal>
       </goals>
     </execution>
   </executions>
```

Tässä luodaan uusi suoritus (execution), jolle määritellään elinkaaren vaihe (phase). Valittuna on integraatiotestaus, sillä tässä vaiheessa verkkosovellus on mahdollisesti käynnistetty. Tästä lisää hyväksymistestauksen luvussa 4.7.1. Tapahtumalle on määritetty nimi (task name), ja ajokomento (exec) kertoo, missä suoritettava ohjelma on ja millä argumenteilla se käynnistetään. Tämä rajoittaa valittavia ohjelmia, sillä jos kaikkea tarvittavaa ei pysty suorittamaan pelkällä komentorivillä, ei automatisointi toimi.

4.6 Tietokantatestaus

Tietokantatestaus on yksi niistä testauksen osa-alueista, mikä jää helposti vähälle huomiolle. Tietokannan toiminta testataan yksinkertaisella SQL-kyselyllä eikä aiheeseen palata kuin paljon myöhemmin. Jos tietokannan osalta ilmeneekin jokin virhe, sitä ei välttämättä huomata lainkaan ennen kuin on liian myöhäistä. Tietokantatestauksen voi automatisoida, jolloin saadaan jatkuvaa palautetta siitä, löytyykö tietokannasta se, mitä kuuluukin ja että kaikki toimii. [29]

Testaus tapahtuu JUnit-yksikkötestausta laajentavalla DbUnit-kehyksellä. Nyt sen sijaan, että korvataan tietokanta matkijaolioilla, voidaan käsitellä itse tietokannan tietoja. DbUnitin etu perustuu siihen, että ennen testiä tietokantaa voidaan täyttää sopivalla tiedolla, ja testin päätteeksi siivota se alkuperäiseen asetelmaansa. [30] Tietokannan dataa esitetään XML-muodossa, joka sitten lisätään tietokantaan testiä varten. DbUnit antaa vaihtoehdoiksi toteuttaa tämän suoraan yksikkötestien koodissa, tai sitten koontityökalujen ohjeistuksessa, mihin sekä Ant että Maven soveltuvat. XML-määritystiedosto *henkilot.xml* voisi näyttää seuraavanlaiselta.

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
  <HENKILO  hid='1'
            syntynyt='1983-01-06'
            etunimi='Pekka'
            sukunimi='Peltola' />
```

```
  <HENKILO  hid='2'
            syntynyt='1992-05-04'
            etunimi='Heidi'
            sukunimi='Jokiuoma' />
</dataset>
```

Tiedosto lisäisi HENKILO-tauluun kaksi henkilöä. Nämä XML-tiedot on mahdollista kirjoittaa käsin, mutta kattavan testidatakokonaisuuteen kirjoittamiseen kuluisi mahdollisesti huomattava määrä aikaa. DbUnit mahdollistaa tiedon tuomisen XML-muotoon myös valmiiksi täytetystä tietokannasta, joten testidatan voi lisätä jouhevasti sopivalla ohjelmakoodilla tai työkalulla.

Pienessä ohjelmistotuotantoryhmässä, kuten Metropolian ohjelmistotuotantoprojektissa tietokantatestauksen voi suorittaa yhdellä testitietokannalla, kun taas suuremmissa projekteissa tietokantoja voi olla käytössä jopa neljä [29]. Tietyistä pelisäännöistä on kuitenkin hyvä sopia, että koko ryhmä tietää, mitä tietokannassa kuuluu olla. Tilanne, missä ohjelmistotuotteen testaus menee pieleen tyhjennetyn tietokannan vuoksi, ei ole lainkaan tavaton. Ongelmaan voikin hakea ratkaisua automatisoidun tietokantatestauksen ohessa tapahtuvasta tietokannan oletustilaansa palauttamisesta.

Kuten aiemmin tuli ilmi, testauksen voi suorittaa kahdella eri tavalla. Käydään läpi ensin esimerkki puhtaasti yksikkötestien kautta tapahtuvasta hallinnasta ja lopuksi Maveniä laajemmin hyödyntävä esimerkki.

Ensimmäisenä tulee pom.xml:n riippuvuudet ottaa kuitenkin huomioon. DbUnitin riippuvuus on seuraavanlainen:

```
<dependency>
    <groupId>org.dbunit</groupId>
    <artifactId>dbunit</artifactId>
    <version>2.4.3</version>
</dependency>
```

Nyt voidaan kirjoittaa yksikkötesti, jossa otetaan yhteyttä tietokantaan. Oletetaan, että käytetään DAO-luokkaa, joka käsittelee aiemmin XML-muodossa esiteltyä HENKILO-taulua. Tietokanta halutaan tyhjentää, lisätä XML-tiedoston henkilöt tauluun ja testata DAO-luokan hakutoimintoa.

Pitää luoda *DatabaseTestCase,* missä toteutuu vähintään metodit getConnection(), joka luo yhteyden tietokantaan ja getDataSet(), joka lisää dataa tietokantaan.

```
public class DAOTest extends DatabaseTestCase{
  protected IDatabaseConnection getConnection()
  throws Exception {
    Class driverClass =
    Class.forName("com.mysql.jdbc.Driver");
    Connection yhteys =
    DriverManager.getConnection(
     "jdbc:mysql://1.0.0.2:3306/hlokanta",
     "ohtu",
     "salasana"
    );
           return new DatabaseConnection(yhteys);
  }

  protected IDataSet getDataSet() throws Exception
  {
    return new FlatXmlDataSet(
      new FileInputStream("henkilot.xml"));
  }
}
```

Esimerkissä luotiin MySQL-tietokantayhteyden- ja henkilot.xml-tiedoston hyödyntämisen mahdollistava tietokantatestiluokka. Nyt voidaan kirjoittaa ensimmäinen testi, sillä DbUnitin oletuskäyttäytyminen on *CLEAN_INSERT*, eli se tyhjentää taulun ennen kuin testejä ajetaan ja siihen lisätään tietoa. Tätä ja jälkipuhdistustoimintoja voi muokata toteuttamalla myös metodit getSetUpOperation() ja getTearDownOperation() [31]. Kirjoitetaan testiesimerkki:

```
public void testaaHenkiloHaku() throws Exception{
  DAO dao = new DAO();
  int henkilonId = 1;
  String henkilonNimi = ”Pekka”;
```

```
  Henkilo saatuHenkilo = dao.haeHenkilo(henkilonId);
  TestCase.assertEquals(
   ”Pitäisi olla Pekka”,
   henkilonNimi,
   saatuHenkilo.getEtunimi()
  );
}
```

Testissä tarkistetaan, onko DAO-luokan palauttaman henkilön etunimi oletettu. Huomaataan, että testien kirjoittaminen itsessään ei juuri poikkea normaalista yksikkötestauksesta.

Testit on mahdollista kirjoittaa normaaleina yksikkötesteinä hyödyntämättä *DatabaseTestCase*-laajennusta. Tällöin aiemmin toteutetun rajapinnan vaatimat tapahtumat tulee määritellä koontityökalujen kautta. Mavenillä tämä tapahtuu aktivoimalla DbUnit-liitännäinen *build*-osiossa. Liitännäisen käyttöönotto on hieman pidempi kuin aiemmissa esimerkeissä, joten käydään määritykset läpi osissa.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>dbunit-maven-plugin</artifactId>
  <version>1.0-beta-3</version>

  <!-- jdbc ajuri -->
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.0.5</version>
    </dependency>
  </dependencies>
```

Yllä asetettiin MySQL-riippuvuus liitännäiselle. Tähän mennessä riippuvuudet on asetettu aina *dependencies*-osioon.

```
  <!-- yhteysasetukset -->
  <configuration>
    <driver>com.mysql.jdbc.Driver</driver>
    <url> jdbc:mysql://1.0.0.2:3306/hlokanta </url>
```

```
    <username>ohtu</username>
    <password>salasana<password>
  </configuration>
```

Nämä asetukset vastaavat edellisen esimerkin getConnection()-metodia. Siinä määritellään tietokannan tyyppi, kohde ja käyttäjätunnus.

```
  <executions>
    <execution>
      <phase>test-compile</phase>
      <goals>
        <goal>operation</goal>
      </goals>
      <configuration>
        <type>CLEAN_INSERT</type>
        <src>henkilot.xml</src>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Liitännäisille on tavanomaista, että niille määritellään suoritteita (*executions*). Tämän esimerkin suoritusvaiheeksi on valittu *test-compile*, joka edeltää vaihetta *test*. Tietokanta on siis testausvalmiudessa ennen varsinaista testien ajoa (taulukko 1 esittelee kaikki vaiheet). Henkilot.xml:n tiedot lisätään jälleen tyhjennettyyn tietokantaan CLEAN_INSERT:illä.

4.7 Hyväksymistestaus

Tähän mennessä teemat (penetraatiotestaus mahdollisena poikkeuksena) ovat käsitelleet lasilaatikkotestausta (white box testing), missä testaaja tuntee ohjelmiston rakenteen. Testien tarkoituksena on selvittää ”onko koodi ohjelmoitu oikein”. Hyväksymistestauksessa otetaan selville ”onko oikea koodi ohjelmoitu”. Mustalaatikkotestauksen (black box testing) piiriin kuuluva hyväksymistestaus on mahdollista toteuttaa ilman aiempaa ohjelman tuntemusta. Testit perustuvat usein asiakkaan vaatimuksiin, jolloin viimeiset hyväksymistestit toimivat takuuna, että asiakas saa sen mitä on pyytänyt. Jois-

sakin projekteissa asiakas itse toteuttaa hyväksymistestauksen, ja antaa tulosten perusteella viimeiset korjausvaatimukset.

Testit tapahtuvat käyttäjälle näkyvässä rajapinnassa eli käyttöliittymässä. Tarkoitus on toteuttaa ajateltu skenaario ja tarkistaa, että lopputulos on hyväksyttävissä. Esimerkki testistä voisi perustua verolaskuriohjelmaan, missä 3000 euron kuukausitulot tuottavat veroprosentiksi 29 %. Kun ohjelmaan syötetään ”tulot”-tekstikenttään 3000 ja painetaan ”laske”-painiketta, ”veroprosentti”-tekstikentässä tulee näkyä tulos 29. Muulloin testi on hylätty, koska ohjelma ei tuota hyväksymistä edellyttävää tulosta.

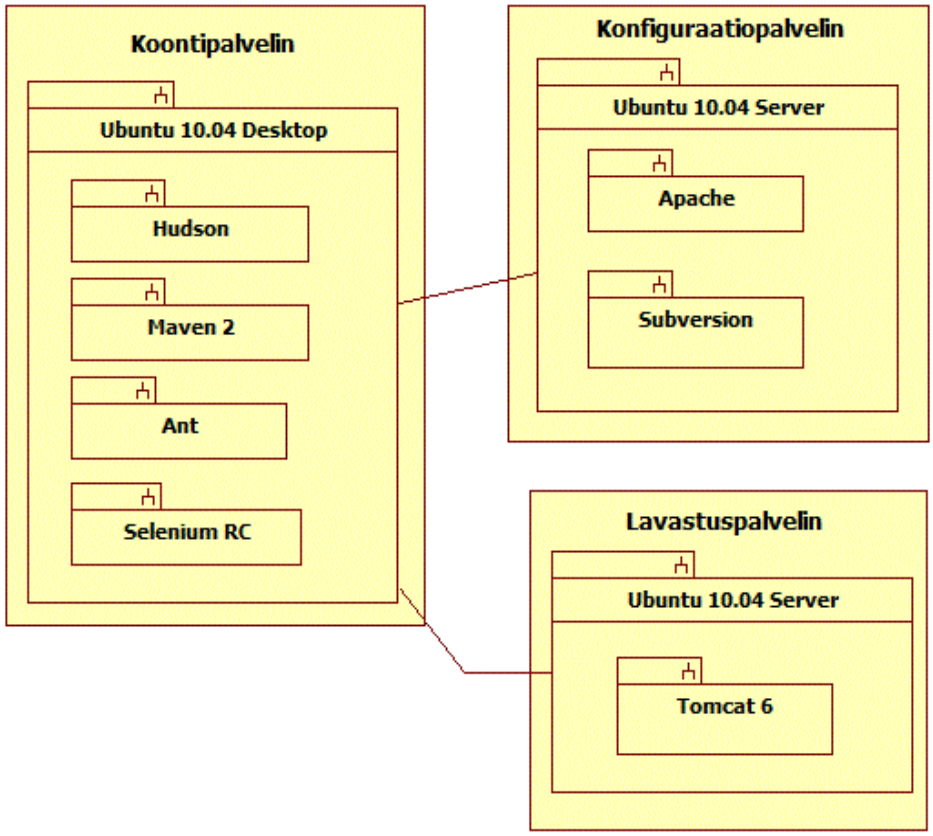
Hyväksymistestaus ei välttämättä ole helpointa automatisoida virhealttiuden takia. Päänvaivaa voi tuottaa myös koonti- ja testausympäristön graafisen käyttöliittymän puute, jota pyrittiin Metropolian järjestelmässä paikkaamaan asentamalla koontipalvelimelle Ubuntu 10.04 Desktop –versio. Hyväksymistestauksen manuaalinen toteutus ei ole lainkaan tavanomaisuudesta poikkeavaa vielä tänäkään päivänä, vaikka siihen kuluu paljon työaikaa. Toisaalta automatisoitukin testikokoelma voi osoittautua koontiaikaa venyttäväksi resurssirohmuksi.

Java-ohjelmissa yleisimmät käyttöliittymät ovat Swing- ja verkkosovelluskäyttöliittymät. Tarkastellaan molempien automatisoinnin mahdollisuuksia ja sudenkuoppia.

4.7.1 Verkkosovellukset

Verkkosovellusten hyväksymistestaus on huomattavasti paremmassa asemassa kuin swing-käyttöliittymät. Työkaluja on sekä kaupallisena että avoimella lähdekoodilla. Näistä erityisesti Selenium [32] on saavuttanut suosiota. Selenium on moneen suuntaan taipuva järjestelmä. Ydin on Selenium Core, jonka avulla Selenium-testejä voi ajaa sisäänrakennetulla Mozilla Firefox [33] -tuella. Siitä on laajennettu erityistarpeisiin soveltuvia versioita. Selenium IDE on Mozilla Firefox –liitännäinen, joka tarjoaa mm. helpon tavan nauhoittaa testejä suoraan selaimella. Selenium RC (Remote Control) auttaa Selenium-testien ajamiseen muilta internetselaimilta, kuten Internet Explorel tai Safari. Selenium Grid hallinnoi Selenium-testejä hajautetusti monelta koneelta, jolloin testeihin käytettyä aikaa voidaan vähentää ja saadaan parempi testikattavuus eri järjestelmien suhteen.

Metropolian automatisoidun koonti- ja testausjärjestelmän suunnittelussa otettiin huomioon Selenium-testaus. Monista mahdollisuuksista aikataulun puitteissa valittiin Selenium RC:hen pohjautuva koonti- ja testauspalvelimen yhteistyöhön perustuva ratkaisu. Testauspalvelimeksi alkuperäisessä suunnitelmassa (kuva 1) nimetty palvelin voisi lopullisessa toteutuksessa olla kuvaavammin lavastuspalvelin (staging), koska siellä ei ajeta testejä vaan lavastetaan sovelluksen toimintaa. Palvelin toimii väliaikaisena lavana, ennen todellista käyttöönottoa. Lavastuspalvelimelle asennettiin Ubuntu 10.04 Server ja Tomcat 6 –palvelinohjelmisto. Päivitetään koonti- ja testiympäristön palvelinkuvaus toiseen välivaiheeseensa, jota esittää kuva 9.



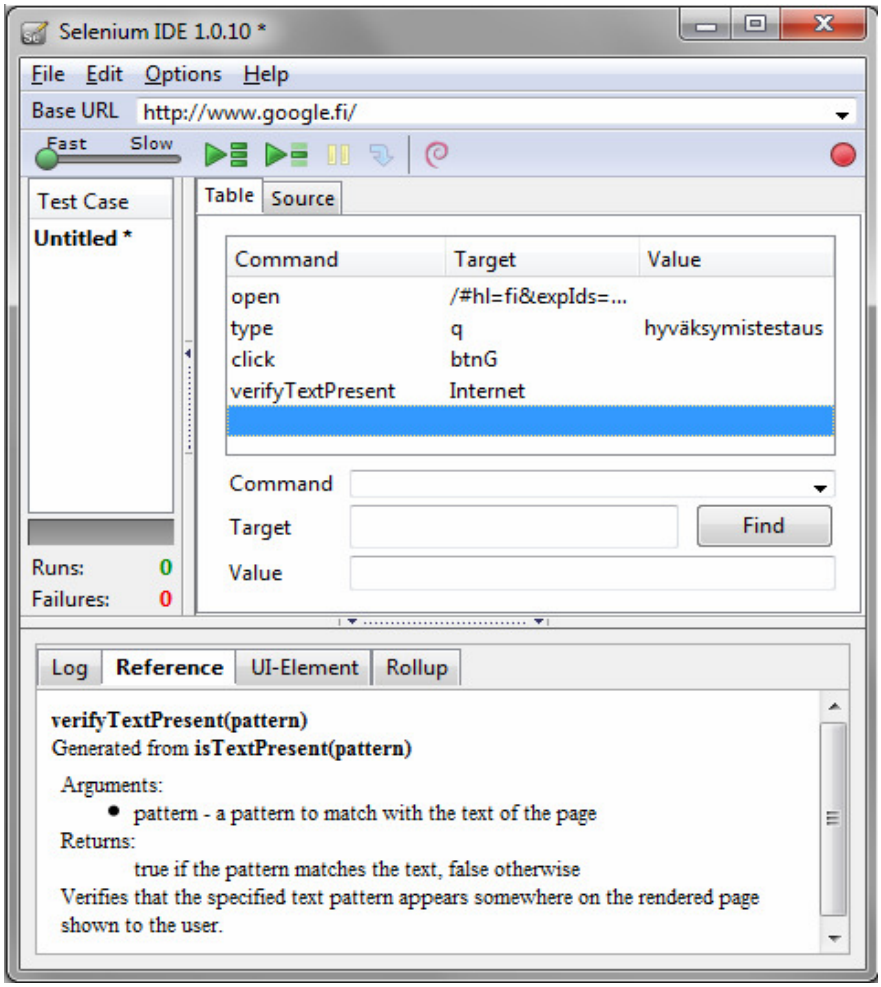
Kuva 9. Palvelinratkaisun toinen välimuoto

Kuvasta huomataan, että koontipalvelimelle on lisätty Selenium RC, ja lavastuspalvelin on kokonaan uusi. Tämä on jo lähes valmis kokonaisuus, sillä kolmas välivaihe on lopullinen. Tämä mahdollistaa verkkopohjaisen Java-sovelluksen ajamisen lavastuspalvelimella ja hyväksymistestien ajamisen koontipalvelimella hyödyntäen eri selaimia. Selainvalikoima rajoittuu niihin internetselaimiin, mitä Ubuntulle pystyy asentamaan. Selkein puute on siis

Internet Explorel –selaimen pois jääminen. Tämä vaatisi Selenium Grid –toteutuksen, missä vähintään yhdelle Selenium-testikoneista on asennettu Windows-käyttöjärjestelmä.

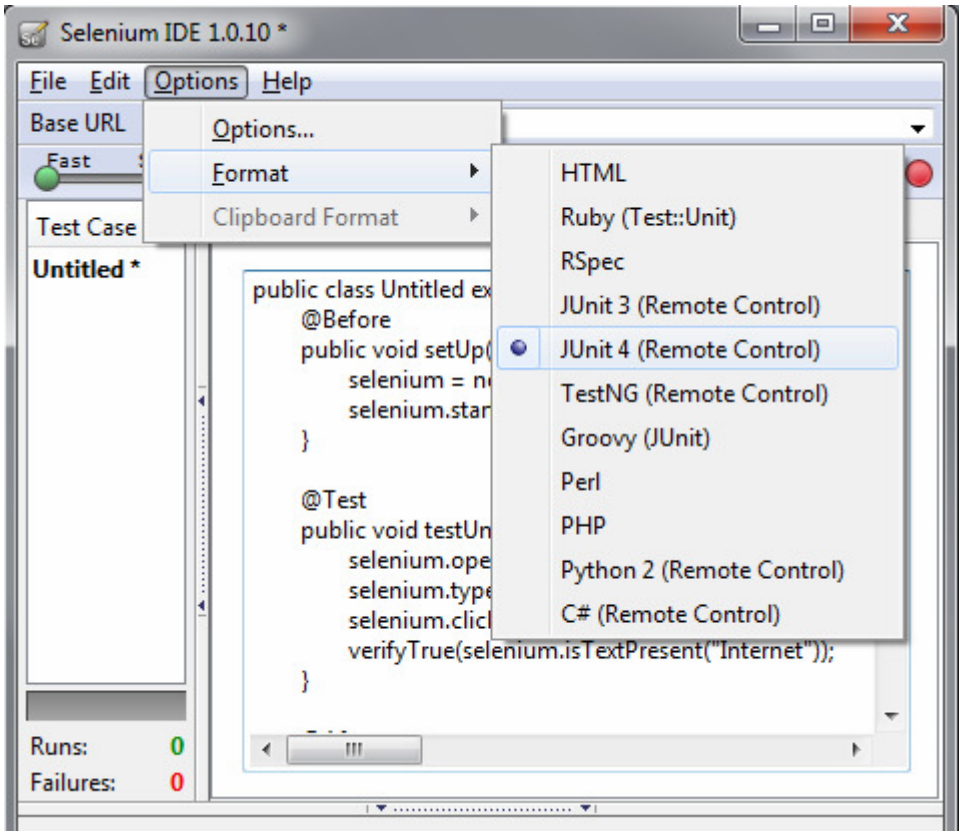
Testauksen käyttöönotto vaatii pom.xml:n muokkausta siten, että koottu ohjelma sijoitetaan (deploy) lavastuspalvelimelle, ajetaan testit ja lopuksi puretaan sijoittaminen. Selenium-testit tulee tietenkin myös kirjoittaa, joten aloitetaan siitä.

Yksi vaihtoehto on kirjoittaa testit alusta loppuun itse, mutta alkuun pääsee helpommin hyödyntämällä Selenium IDE:tä. Tätä varten opiskelijan tulisi asentaa kehityskoneelleen Mozilla Firefox, mikäli sitä ei ennestään ole. Selaimelle asennetaan tämän jälkeen Selenium IDE –liitännäinen [32]. Selaimeen avataan testattava sivusto, eli tässä tapauksessa esimerkiksi tuotantokoneella väliaikaisesti käyttöönotettu verkkosovellus, ja käynnistetään Selenium IDE ”Tools”-valikosta.



Kuva 10. Selenium IDE:n käyttöliittymä

Käyttöliittymä on perustoimintojen puitteissa selkeä (kuva 10). Selenium IDE menee oletusarvoisesti nauhoitustilaan, jota voi hallita oikean yläkulman nauhoituspainikkeella. Verkkosivun selaaminen tallentuu komentohistoriaan myöhempää toistoa varten testien muodossa. Testin voi päättää tarkistukseen, esim. verifyTextPresent, joka tarkistaa tietyn tekstin esiintymisen sivulla. Tämän voi lisätä helposti maalaamalla halutun tekstin ja valitsemalla toiminto oikean hiirenpainikkeen takaa löytyvästä valikosta. Testi rakentuu nauhoituksen yhteydessä ja vaihtamalla ”Source”-välilehteen, saa näkyviin testin HTML-muotoisen variaation, jota Selenium Core osaa tulkita. Halutaan kuitenkin hyödyntää tutuksi tullutta JUnitia, joten ”Option”-valikosta voi vaihtaa formaattia (format) JUnit 4:ään (kuva 11).



Kuva 11. Selenium IDE:n testiformaattivaihtoehdot

Tästä testikoodista saa hyvän pohjan, jonka voi kopioida ohjelmointiympäristöönsä. Sitä pitää kuitenkin muokata vastaamaan koonti- ja testausjärjestelmän IP-asetuksia ja yhteistyötä Mavenin kanssa. Otetaan esimerkiksi eräs Selenium IDE:n luoma testikoodi.

```
public class selenium01Test extends SeleneseTestCase {
  @Before
  public void setUp() throws Exception {
     selenium = new DefaultSelenium("localhost",
     4444, "*chrome", "localhost");
    selenium.start();
  }

  @Test
  public void testSoktest01() throws Exception {
    selenium.open("/Sok/login.jsp");
    selenium.type("usernameField", "admin");
    selenium.type("passwordField", "pass");
    selenium.click("//input[@value='Login']");
    selenium.waitForPageToLoad("30000");
     verifyTrue(selenium.isTextPresent("firstn
     secondn admin"));
  }
  @After
  public void tearDown() throws Exception {
    selenium.stop();
  }
}
```

Huomataan, että testiluokka on JUnitia laajentava *SeleneseTestCase.* Testikoodi näyttää hyvältä, ja se voi jopa toimia kehityskoneella tietyissä olosuhteissa, mutta kaikki esimerkissä korostettu tulisi poistaa, eli setUp() –metodin sisältö ja koko tearDown() –metodi. Tämä tehdään sen takia, että Maven käynnistää ja sulkee Seleniumin, jolloin testiluokka yrittäisi tuloksettomasti avata uuden instanssin luultavasti aiheuttaen ristiriidan jo olemassa olevan instanssin kanssa. Lisäksi tiedot eivät viittaa oikeaan palvelimeen. Lisätään kuitenkin setUp() –metodiin seuraava:

```
setUp("http://10.95.250.100", "*firefox");
```

Tässä oletetaan, että lavastuspalvelimen IP-osoite on 10.95.250.100. Toinen parametri on yksi monista vaihtoehdoista, joka määrittelee sen, millä selaimella testi avataan. Tämä ei kuitenkaan toimi kuiviltaan, vaan selaimen

polku tulee lisätä palvelimella ympäristömuuttujaan PATH. Vaihtoehtoisesti polun voi lisätä osaksi parametria:

```
setUp("http://10.95.250.100","*firefox /usr/bin/firefox");
```

Ilman erityishuomiota, Selenium-testit ajetaan muiden yksikkötestien ohessa Mavenin testivaiheessa. Tämä ei ole mahdollista, koska ohjelman tulee olla julkaisukelpoinen pakkaus. Halutaan siis toimia integraatiotestausvaiheessa. On useampi vaihtoehto, kuinka tilanne ratkaistaan. Nopea, vaikkakin epälooginen, vaihtoehto on lykätä kaikki testaus integraatiotesteihin. Parempi ratkaisu on hyödyntää sisällyttämistä (include) ja pois jättämistä (exclude). Yhdessä näihin perustuvassa ratkaisussa Selenium-testit voidaan tallentaa eri hakemistoon kuin muut testit, jolloin ne jäävät ajamatta testausvaiheessa. Integraatiotestausvaiheeseen kyseinen hakemisto lisätään, ja muut testit jätetään pois. Tämä ja kaikki muut yksityiskohdat huomioiden, työkaluarsenaali tulee olemaan laaja. Tarvitaan Selenium-liitännäinen käynnistämään Selenium-palvelimen, Cargo-liitännäisen sijoittamaan sovelluksen Tomcatin ajettavaksi lavastuspalvelimella, ja Surefire-liitännäinen hallinnoimaan testien ajamista.

Aloitetaan pom.xml-muokkaus lisäämällä tietovarastoja Seleniumia varten. Nämä määritykset tulevat ennen riippuvuuksia, eli *dependencies*-osion yläpuolelle.

```
<pluginRepositories>
  <pluginRepository>
     <id>codehaus snapshot repository</id>
     <url>https://nexus.codehaus.org/content/groups/
          snapshots-group/</url>
    <releases>
      <enabled>true</enabled>
    </releases>
  </pluginRepository>
</pluginRepositories>

<repositories>
  <repository>
    <id>openqa</id>
    <name>OpenQA Repository</name>
```

```
    <url>http://nexus.openqa.org/content
         /repositories/releases/</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <releases>
      <enabled>true</enabled>
    </releases>
  </repository>
</repositories>
```

Jos Maven ei löydä etsimäänsä riippuvuutta tai liitännäistä omasta tietovarastostaan, se etsii sitä muista määritellyistä kohteista. Seleniumin käyttöönotto vaatii myös riippuvuuksien määrittämistä. Yksikkötestausluvun JUnit riippuvuudet tulee myös olla kunnossa.

```
<dependency>
  <groupId>org.openqa.selenium.client-
           drivers</groupId>
  <artifactId>selenium-java-client-
              driver</artifactId>
  <version>0.9.2</version>
  <scope>test</scope>
</dependency>

<dependency>
 <groupId>org.openqa.selenium.server</groupId>
  <artifactId>selenium-server</artifactId>
  <version>1.0-20081010.060147</version>
  <scope>test</scope>
</dependency>
```

Seuraavaksi voidaan määritellä Selenium-liitännäisen käyttöönoton *build*-osiossa. Seuraavissa määrityksissä Selenium-palvelin käynnistetään *pre-integration-test*-vaiheessa ja sammutetaan *post-integration-test*-vaiheessa.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>selenium-maven-plugin</artifactId>
```

```
  <executions>
    <execution>
      <id>start-selenium-server</id>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>start-server</goal>
      </goals>
      <configuration>
        <background>true</background>
        <logOutput>true</logOutput>
        <multiWindow>true</multiWindow>
      </configuration>
    </execution>
    <execution>
      <id>stop-selenium-server</id>
      <phase>post-integration-test</phase>
      <goals>
        <goal>stop-server</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Nyt kun tiedetään, että Selenium-palvelin on päällä integration-test -vaiheessa, voidaan muokata Surefire-liitännäisen asetukset vastaamaan tilannetta.

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <excludes>
     <exclude>seleniumtest/*.java</exclude>
   </excludes>
  </configuration>
  <executions>
    <execution>
```

```
        <phase>integration-test</phase>
        <goals>
          <goal>test</goal>
        </goals>
        <configuration>
         <excludes>
            <exclude>test/*.java</exclude>
          </excludes>
          <includes>
           <include>seleniumtest/*.java</include>
         </includes>
          <skip>false</skip>
          <testFailureIgnore>true</testFailureIgnore>
        </configuration>
      </execution>
    </executions>
</plugin>
```

Normaalissa testivaiheessa oletussisällytykset ovat **/Test*.java, **/*Test.java sekä **/*TestCase.java [34]. Ylläolevasta jätettiin pois hakemistossa ”seleniumtest” sijaitsevat testit, missä esimerkin Selenium-testit sijaitsevat. Myöhemmin aktivoidaan suorite *integration-test*-vaiheessa, mistä on jätetty pois hakemisto ”test” ja sisällytetty ”seleniumtest”. Näin jaettuna testit ajetaan aina oikeassa vaiheessa. Integraatiotestauksen vaiheessa on myös hyvä asettaa *testFailureIgnore*, eli Maven jatkaa eteenpäin, vaikka osa testeistä ei menisikään läpi. Tämä tehdään siksi, että jos Maven pysähtyy tässä vaiheessa, jää Selenium-palvelin ja testattava ohjelma turhaan päälle odottamaan manuaalista siivousta.

Verkkosovellus pitää olla ajokunnossa testausta varten, joten lisätään Cargo käyttöön. Tässä esimerkissä hyödynnetään Tomcat 6:n ominaisuuksiin kuuluvaa ajonaikaista sijoitusta (hot deploy). Palvelin on jo ennestään päällä, ja verkkosovellus sijoitetaan ajettavaksi suoraan lennosta, ja testien jälkeen se otetaan pois käytöstä. Käydään asetukset läpi osissa.

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <executions>
    <execution>
      <id>deploy</id>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>deployer-deploy</goal>
      </goals>
    </execution>
    <execution>
      <id>undeploy</id>
      <phase>post-integration-test</phase>
      <goals>
        <goal>deployer-undeploy </goal>
      </goals>
    </execution>
  </executions>
```

Esimerkissä määritellään kaksi suoritetta. Ennen integraatiotestausvaihetta annetaan käsky liittää ja vastaavasti testauksen jälkeen käsketään purkaa liitos.

```
<configuration>
  <container>
    <containerId>tomcat6x</containerId>
    <type>remote</type>
  </container>
```

Kerrotaan kyseessä olevan etäinen Tomcat 6 –palvelin, jolle seuraavaksi määritellään tarvittavat asetukset.

```
  <configuration>
    <type>runtime</type>
      <properties>
        <cargo.tomcat.manager.url>
```

```
          http://10.95.250.112/manager
        </cargo.tomcat.manager.url>
        <cargo.remote.username>
          Admin
        </cargo.remote.username>
        <cargo.remote.password>
          Salasana
        </cargo.remote.password>
      </properties>
    </configuration>
```

Ajon aikaiseen sijoittamiseen tarvitaan Tomcat-manager –niminen verkkosovellus, joka asentuu oletuksena Tomcatin mukana. Näissä asetuksissa kerrotaan kyseisen ohjelman URL ja käyttäjätunnukset kirjautumista varten.

```
    <deployer>
      <type>remote</type>
      <deployables>
        <deployable>
          <groupId>ohtu.k2010.ryhma3</groupId>
          <artifactId>uusi-projekti</artifactId>
          <type>war</type>
        </deployable>
      </deployables>
    </deployer>
  </configuration>
</plugin>
```

Lopuksi kerrotaan, mitä sijoitetaan, eli tässä tapauksessa juuri koottu esimerkki verkkosovellus. Nyt ohjelma ajetaan integraatiotestien ajan lavastuspalvelimella.

Tomcat voi kärsiä resurssipulasta toistamalla tämä prosessi tarpeeksi monta kertaa. Vaikka kaikki verkkosovellukset eivät ongelmia aiheuttaisikaan, voi Tomcatin ajoittainen alasajo olla hyvä keino välttää ongelmia.

4.7.2 Swing-käyttöliittymä

Swing-hyväksymistestauksen automatisoinnin lähtökohdat eivät ole parhaimmat mahdolliset. Kattavimmat työkalut ovat kaupallisia, monet avoimen lähdekoodin ratkaisut ovat keskeneräisiä raakileita tai tuki nykypäivän automatisointiratkaisuille on olematon. Käyttökelpoisimmat työkalut ovat kallistuneet testien ohjelmointiin, mikä Metropolian ohjelmistotuotantoprojektien kannalta on huonompi vaihtoehto.

Hyväksymistestaukselle on olemassa kaksi selkeää lähtökohtaa: testien ohjelmoiminen ja testien nauhoittaminen. Kummallakin on hyvät ja huonot puolensa. Testien nauhoittaminen tarkoittaa sitä, että testissä simuloidaan käyttäjän toimet aiemmin tehdyn manuaalisen nauhoituksen perusteella ja lopuksi tarkistetaan lopputulos. Samoja nauhoitteita voidaan usein hyödyntää myös ohjelmaa esiteltäessä. Nauhoitteiden heikkous piilee siinä, että käyttöliittymässä tapahtuvat muutokset rikkovat nauhoitteen helposti, jolloin saadaan väärä hälytys ja testitapaus tulee nauhoittaa uudestaan. [35] Tämä kuitenkin olisi Metropolian opiskelijoille ehdottomasti tehokkaampi vaihtoehto, koska testien tekeminen on nopeuden ja helppouden myötä tehokasta.

Testien ohjelmoiminen on sitä, että käyttöliittymän käyttöä simuloidaan sisältä käsin. Tämä on mahdollista toteuttaa pelkällä JUnitillakin, mutta avuksi on myös olemassa muita työkaluja. Niiden tarkoitus on virtaviivaistaa testien ohjelmointia. Vahvuutena on testien kestävyys kosmeettisista muutoksista huolimatta sekä niiden luettavuus ja siirrettävyys [35]. Ohjelmoitavat testit kuitenkin tarkoittaisivat jälleen uuden API:n opiskelua kaiken muun ohella. Tästä johtuen silloin tällöin toteutettavat manuaaliset testit olisivat ohjelmistotuotantoprojektikurssin puitteissa vähemmän aikaa ja resursseja tuhlaava vaihtoehto.

Tarpeeksi yksinkertaista tapaa automatisoida Swing-käyttöliittymätestausta ohjelmistotuotantoprojektin puitteissa ei kirjoitushetkellä ollut saatavilla esiteltäväksi. Tämä ei kuitenkaan ole kurssin kannalta erityisen rampauttavaa, sillä Swing-sovelluksia ei ohjelmistotuotantoprojekteina juuri toteuteta, vaan painopiste on verkkosovelluksilla.

5 DOKUMENTOINTI

Automatisoidun käännös- ja testausprosessin lopputuotteena syntyy ajokelpoisen ja testatun ohjelmiston lisäksi dokumentaatiota. Tämä ei korvaa käsin tuotettua dokumenttikokonaisuutta, mutta toimii sen jatkeena helpottamaan kehitystyötä.

Maven-projektissa dokumentaation hyödyntää sivustoelinkaarta, joka aktivoidaan komennolla *site*. Tuloksena on verkkosivusto, johon koonnin ja testauksen aikana luotu dokumentaatio on koottu. Dokumentaation määrä riippuu käytetyistä liitännäisistä. Osa näistä aktivoidaan pom.xml:n *reporting*-osiossa. Javadoc-liitännäinen on yksi näistä. Sen saa käyttöön oletusasetuksin lisäämällä seuraavanlaiset määritykset:

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-javadoc-plugin</artifactId>
    <version>2.7</version>
    <configuration>
      <!-- Ei asetuksia -->
    </configuration>
  </plugin>
</plugins>
```

Testauksen ohessa luodaan testausraportti, joka on myös saatavissa sivustolta. Luvussa 4.1 käytiin läpi staattiseen testaukseen liittyviä liitännäisiä ja kuva 7 esitti CheckStyle-raporttia. Vastaavanlainen raportti tulee osaksi sivustoa automaattisesti.

Sivuston rakennetta voi muokata site.xml-tiedostolla. [36] Helpoin tapa aloittaa muokkaaminen, on luoda sivusto ensin oletusasetuksin. Tämän jälkeen koontipalvelimelta voidaan hakea site.xml ja muokata sitä tarpeen mukaan.

6 YHTEENVETO

Hudsonia ja Maveniä hyödyntävän automatisoidun jatkuvan integroinnin projektin luominen onnistuu varsin vaivattomasti. Vaatimuksena tälle on Hudsonin käyttöliittymän ja Mavenin pom.xml-editoinnin perusteiden ymmärtäminen.

Eri testauslajeja hyödyntämällä saa aikaiseksi hyvin kattavasti testatun projektin, mutta kattavuus jää loppujen lopuksi opiskelijoiden harkinnanvaraisuuden varaan. Tämän työn esimerkit auttavat pääsemään alkuun mahdollisesti paremmin, kuin verkon pirstoutuneet ja monesti vanhentuneet dokumentaatiot.

Helpoimmalla pääsee normaalilla yksikkötestauksella ja staattisella analyysillä. Yksikkötestauksen laajentaminen XML- ja tietokantatestaukseen, voi olla hyvin opettavaista tuleville Java EE –osaajille. Seleniumilla toteutettujen hyväksymistestien automatisoitu ajaminen vaatii muuta testausta enemmän valmistelua, mutta itse testien tekeminen on helppoa. Selenium on kuitenkin varsin toimiva ratkaisu verkkosovellusten testaamiseen.

Mutaatiotestaus, penetraatiotestaus ja Swing-käyttöliittymän hyväksymistestaus ovat hieman ongelmallisia ohjelmistotuotantoprojektin aikataulun vuoksi. Penetraatiotestauksen ohessa kuitenkin todettiin, että Maven joustaa tarpeen vaatiessa rajojensa ulkopuolelle, jos kiinnostusta luovaan pom.xml-editointiin löytyy.

VIITELUETTELO

[1] Fowler, Martin, *Continuous Integration.* [Verkkodokumentti] 2006 [*Viitattu:* 11.9.2010] Saatavilla: http://www.martinfowler.com/articles/continuousIntegration.html.

[2] Hudson, Saatavilla: http://hudson-ci.org/.

[3] Ant, Saatavilla: http://ant.apache.org/.

[4] Maven, Saatavilla: http://maven.apache.org/.

[5] Williams, Stuart, *TomcatExpert Q/A*. [Verkkodokumentti] [Viitattu: 15.11.2010] Saatavilla: http://www.tomcatexpert.com/ask-the-experts/known-cases-system-compromise-due-running-tomcat-root.

[6] Moore, Dan, *Running Tomcat on port 80.* [Verkkodokumentti] [Viitattu: 15.11.2010] Saatavilla: http://www.mooreds.com/wordpress/archives/295.

[7] Subversion, Saatavilla: http://subversion.tigris.org/.

[8] Mercurial, Saatavilla: http://mercurial.selenic.com/.

[9] Spolsky, Joel, *Subversion Re-education*. [Verkkodokumentti] [Viitattu: 13.11.2010] Saatavilla: http://hginit.com/00.html.

[10] *Wikipedia, Software Metric.* [Verkkodokumentti] [*Viitattu: 15.12.2010*] Saatavilla: http://en.wikipedia.org/wiki/Software_metric.

[11] CheckStyle, Saatavilla: http://checkstyle.sourceforge.net

[12] *Using a Custom Checkstyle Checker Configuration.* [Verkkodokumentti] [Viitattu: 16.12.2010] Saatavilla: http://maven.apache.org/plugins/maven-checkstyle-plugin/examples/custom-checker-config.html.

[13] Gaudin, Olivier, *What makes Checkstyle, PMD, Findbugs and Macker complementary?* [Verkkodokumentti] [Viitattu: 17.12.2010] Saatavilla: http://www.sonarsource.org/what-makes-checkstyle-pmd-findbugs-and-macker-complementary/.

[14] Cobertura, Saatavilla: http://cobertura.sourceforge.net/.

[15] JUnit, Saatavilla: http://junit.sourceforge.net/.

[16] TestNG, Saatavilla: http://testng.org/doc/index.html.

[17] Mook, Kim Yong, *JUnit 4 Vs TestNG – Comparison*. [Verkkodokumentti] [Viitattu: 19.12.2010] Saatavilla: http://www.mkyong.com/unittest/junit-4-vs-testng-comparison/.

[18] Glover, Andrew, *In pursuit of code quality: JUnit 4 vs. TestNG*. [Verkkodokumentti] [Viitattu: 19.12.2010] Saatavilla: http://www.ibm.com/developerworks/java/library/j-cq08296/.

[19] jMock, Saatavilla: http://www.jmock.org.

[20] *Wikipedia, Mutation testing.* [Verkkodokumentti] [*Viitattu: 12.9.2010*] Saatavilla: http://en.wikipedia.org/wiki/Mutation_testing.

[21] Jumble, Saatavilla: http://jumble.sourceforge.net.

[22] Javalanche, Saatavilla: http://www.st.cs.uni-saarland.de/~schuler/javalanche/index.html.

[23] Javalanche, Jaxen example. [Verkkodokumentti] Saatavilla: http://www.st.cs.uni-saarland.de/~schuler/javalanche/example-jaxen.html.

[24] Rainsberger, J.B., *JUnit Recipes: Practical Methods for Programmer Testinh. Greenwich: Manning Publications Co. 2005.*

[25] XMLUnit, Saatavilla: http://xmlunit.sourceforge.net/.

[26] Corsaire, *Penetration testing guide.* [Verkkodokumentti] [Viitattu: 23.12.2010] Saatavilla: http://www.penetration-testing.com/.

[27] Penetration Tests – Tools & Software. Saatavilla: http://www.penetrationtests.com/Tools-Software/.

[28] Codenomicon, *For Penetration Testers.* [Verkkodokumentti] [Viitattu: 21.12.2010] Saatavilla: http://www.codenomicon.com/solutions/penetration-testers.shtml.

[29] Dallaway, Richard, *Unit testing database code.* [Verkkodokumentti] [Viitattu: 28.12.2010] Saatavilla: http://www.dallaway.com/acad/dbunit.html.

[30] Glover, Andrew, *Effective Unit Testing with DbUnit.* [Verkkodokumentti] [Viitattu: 28.12.2010] Saatavilla: http://onjava.com/pub/a/onjava/2004/01/21/dbunit.html.

[31] DbUnit, *Getting Started. [Verkkodokumentti] [Viitattu: 3.1.2010] Saatavilla: http://www.dbunit.org/howto.html#createtest.*

[32] Selenium, Saatavilla: http://seleniumhq.org/.

[33] Mozilla Firefox, Saatavilla: http://www.mozilla-europe.org/fi/firefox/.

[34] Surefire plugin, *Inclusions and exlusions of tests.* [Verkkodokumentti] [Viitattu: 7.1.2011] Saatavilla: http://maven.apache.org/plugins/maven-surefire-plugin/examples/inclusion-exclusion.html.

[35] Iline, Alexandre, *Test recording or test coding.* [Verkkodokumentti] [Viitattu: 4.1.2011] Saatavilla: https://jemmy.dev.java.net/RecordingVSCoding.html.

[36] Apache Maven Project, *Creating a site.* [Verkkodokumentti] Saatavilla: http://maven.apache.org/guides/mini/guide-site.html.

pom.xml esimerkki

```
<?xml version="1.0" encoding="UTF-8"?>
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">

<modelVersion>4.0.0</modelVersion>
<groupId>ohtu.k2010.ryhma3</groupId>
<artifactId>uusi-projekti</artifactId>
<version>1.0</version>
<packaging>war</packaging>
<name>Uusi Projekti</name>

<pluginRepositories>
  <pluginRepository>
    <id>codehaus snapshot repository</id>
    <url>https://nexus.codehaus.org/content/groups/snapshots-group/</url>
    <releases>
      <enabled>true</enabled>
    </releases>
  </pluginRepository>
</pluginRepositories>

<repositories>
  <repository>
    <id>openqa</id>
    <name>OpenQA Repository</name>
    <url>http://nexus.openqa.org/content/repositories/releases/</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <releases>
      <enabled>true</enabled>
    </releases>
  </repository>
</repositories>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.jmock</groupId>
    <artifactId>jmock-junit4</artifactId>
    <version>2.5.1</version>
  </dependency>
  <dependency>
    <groupId>xmlunit</groupId>
    <artifactId>xmlunit</artifactId>
    <version>1.3</version>
  </dependency>
  <dependency>
    <groupId>org.dbunit</groupId>
    <artifactId>dbunit</artifactId>
    <version>2.4.3</version>
```

```
  </dependency>
  <dependency>
    <groupId>org.openqa.selenium.client-drivers</groupId>
    <artifactId>selenium-java-client-driver</artifactId>
    <version>0.9.2</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.openqa.selenium.server</groupId>
    <artifactId>selenium-server</artifactId>
    <version>1.0-20081010.060147</version>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.0.2</version>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <excludes>
          <exclude>seleniumtest/*.java</exclude>
        </excludes>
      </configuration>
      <executions>
        <execution>
          <phase>integration-test</phase>
          <goals>
            <goal>test</goal>
          </goals>
          <configuration>
            <excludes>
              <exclude>test/*.java</exclude>
            </excludes>
            <includes>
              <include>seleniumtest/*.java</include>
            </includes>
            <skip>false</skip>
            <testFailureIgnore>true</testFailureIgnore>
          </configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-antrun-plugin</artifactId>
      <version>1.6</version>
      <executions>
        <execution>
          <phase>integration-test</phase>
          <configuration>
```

```
            <tasks name="Penetraatio testi">
              <exec
               dir="${basedir}"
               executable="${basedir}/ptestprogram "
               output="${basedir}/t/tulos.txt">
               <arg value="/scan"/>
               <arg value="http://1.0.0.1/index.html"/>
              </exec>
            </tasks>
          </configuration>
          <goals>
            <goal>run</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
<plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>dbunit-maven-plugin</artifactId>
      <version>1.0-beta-3</version>
      <!-- jdbc ajuri -->
      <dependencies>
        <dependency>
          <groupId>mysql</groupId>
          <artifactId>mysql-connector-java</artifactId>
          <version>5.0.5</version>
        </dependency>
      </dependencies>
      <!-- yhteysasetukset -->
      <configuration>
        <driver>com.mysql.jdbc.Driver</driver>
        <url> jdbc:mysql://1.0.0.2:3306/hlokanta </url>
        <username>ohtu</username>
        <password>salasana<password>
      </configuration>
      <executions>
        <execution>
          <phase>test-compile</phase>
          <goals>
            <goal>operation</goal>
          </goals>
          <configuration>
            <type>CLEAN_INSERT</type>
            <src>henkilot.xml</src>
          </configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>selenium-maven-plugin</artifactId>
      <executions>
        <execution>
          <id>start-selenium-server</id>
          <phase>pre-integration-test</phase>
          <goals>
            <goal>start-server</goal>
          </goals>
          <configuration>
            <background>true</background>
            <logOutput>true</logOutput>
```

```
        <multiWindow>true</multiWindow>
      </configuration>
    </execution>
    <execution>
      <id>stop-selenium-server</id>
      <phase>post-integration-test</phase>
      <goals>
        <goal>stop-server</goal>
      </goals>
    </execution>
  </executions>
</plugin>
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <executions>
    <execution>
      <id>deploy</id>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>deployer-deploy</goal>
      </goals>
    </execution>
    <execution>
      <id>undeploy</id>
      <phase>post-integration-test</phase>
      <goals>
        <goal>deployer-undeploy </goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <container>
      <containerId>tomcat6x</containerId>
      <type>remote</type>
    </container>
    <configuration>
      <type>runtime</type>
      <properties>
        <cargo.tomcat.manager.url>
          http://10.95.250.112/manager
        </cargo.tomcat.manager.url>
        <cargo.remote.username>
          Admin
        </cargo.remote.username>
        <cargo.remote.password>
          Salasana
        </cargo.remote.password>
      </properties>
    </configuration>
    <deployer>
      <type>remote</type>
      <deployables>
        <deployable>
          <groupId>ohtu.k2010.ryhma3</groupId>
          <artifactId>uusi-projekti</artifactId>
          <type>war</type>
        </deployable>
      </deployables>
    </deployer>
  </configuration>
```

```
    </plugin>
  </plugins>
</build>

<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>2.7</version>
      <configuration>
        <!-- Ei asetuksia -->
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>
        maven-checkstyle-plugin
      </artifactId>
      <version>2.5</version>
      <configuration>
        <configLocation>checkstyle.xml</configLocation>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</artifactId>
      <version>2.4</version>
    </plugin>
  </plugins>
</reporting>
</project>
```