Kalle Juntunen

OHJELMISTOTESTAUKSEN KATTAVUUDEN ARVIOINTI

Opinnäytetyö
Kajaanin ammattikorkeakoulu
Tradenomikoulutus
Syksy 2007

Kajaanin ammattikorkeakoulu

OPINNÄYTETYÖ
TIIVISTELMÄ

Koulutusala Luonnontieteiden ala	Koulutusohjelma Tietojenkäsittelyn koulutusohjelma
Tekijä(t) Kalle Juntunen	
Työn nimi Ohjelmistotestauksen kattavuuden arviointi	
Vaihtoehtoiset ammattiopinnot Ohjelmointi	Ohjaaja(t) Synnöve Hämäläinen
	Toimeksiantaja Ebsolut Oy
Aika 22.11.2007	Sivumäärä ja liitteet 29+1

Opinnäyte kehittää Ebsolut Oy:n käyttämää automaattista testausjärjestelmää. Yrityksen käytössä oleva ABaTS-järjestelmä (Automated Build and Test System) on tradenomi Antti Kemppaisen opinnäytetyön tuotos. ABaTS suorittaa moduulitestausta java-kielisille ohjelmille. Työn tavoitteena oli lisätä järjestelmään toiminnallisuus, joka mittaa ja raportoi testien kattavuuden.

Testauksen kattavuuden arviointi toteutettiin käyttämällä Emma-kirjastoa. Emma on java-pohjainen työkalu testauksen kattavuuden mittaamiseen ja raportointiin. Emman vahvuuksia ovat helppo käyttöönotto, avoin lähdekoodi ja ilmaisuus. Kaikkia Emman toimintoja voidaan käyttää sekä komentokehotteella, että Ant-skripteillä.

Emman kattavuusmittausta käytettäessä käännettävään lähdekoodiin tulee lisätä debug-tiedot. Käännetty tavukoodi instrumentoidaan, eli Emma lisää oman kerroksensa kattavuuden mittausta varten. Samalla luodaan metadata testattavista luokista. Instrumentoidut luokat ajetaan virtuaalikoneessa ja Emma kerää ajonaikaista seurantatietoa. Testiajon päätteeksi luodaan html-raportti vertaamalla metadataa ja ajonaikaisen seurannan tietoja.

Käytännön toteutus alkoi Emmaan ja ABaTS-järjestelmään tutustumisella. Aluksi tarvittavat toiminnot toteutettiin komentorivityökaluilla, jonka jälkeen ne lisättiin ABaTS:in käyttämiin Ant-skripteihin. Ant-skriptien käyttö oli perusteltua, koska koko ABaTS-järjestelmä on toteutettu niillä.

Jatkossa Emma-kirjasto otetaan käyttöön ABaTS-ympäristön ulkopuolella järjestelmä- ja kuormitustestaukseen. Tätä varten tulee laatia ohjeistus yrityksen työntekijöille.

Testauksen kattavuuden arviointi helpottaa ja tehostaa testaajien työtä. Testien jälkeen on nähtävillä mihin osaan koodia testitapaukset vaikuttavat. Tämä auttaa uusien testitapausten suunnittelussa. Testauksen tehostaminen edistää osaltaan ohjelmiston laatua.

Kieli	Suomi
Asiasanat	Testaus, testauksen kattavuus
Säilytyspaikka	☐ Kajaanin ammattikorkeakoulun Kaktus-tietokanta ☐ Kajaanin ammattikorkeakoulun kirjasto

Kajaanin ammattikorkeakoulu
University of Applied Sciences

THESIS
ABSTRACT

School Natural Sciences	Degree Programme Data Processing
Author(s) Kalle Juntunen	
Title Code Coverage in Software Testing	
Optional Professional Studies Programming	Instructor(s) Synnöve Hämäläinen
	Commissioned by Ebsolut Oy
Date 22.11.2007	Total Number of Pages and Appendices 29+1

The goal of this project was to develop the automated software testing environment used in the Ebsolut Oy. The company uses the ABaTS (Automated Build and Test System) for module and integration testing. ABaTS is a product of a thesis project made by one of the former students. ABaTS is used for testing Java software.

The code coveage functions were implemented by java based Emma library. Emma is a toolkit for measuring and reporting code coverage. Emma was chosen because it is free, easy to use and it has open source code. All of the Emma's functions can be used both with the command prompt and the Ant scripts.

To use the Emma library for code coverage, the source code must be compiled with the debug information on. The byte code is instrumented. In this stage, Emma adds a new layer into the byte code for the coverage measurement and metadata is created based on the instrumented classes. The instrumented byte code is executed in the Java virtual machine and Emma gathers runtime data on coverage. Finally, the test report is generated by comparing the instrumentation metadata and runtime data.

The project started by gathering information on Emma and ABaTS. First, all the necessary functions were implemented and tested with the command line tools. Afterwards they were replaced with Ant-commands, because the whole ABaTS is based on the Ant.

In the future, Emma will also be used outside the ABaTS for system and stress testing. Therefore, an instruction manual is required for the employees. The code coverage will make the testers' work easier and more efficient. The testers need to actually see what part of the code the test cases affect. This new information will help them in developing new test cases. By making the testing process more efficient, the quality of the whole software will be better.

Language of Thesis	Finnish
Keywords	Code coverage, testing, software testing, automated testing
Deposited at	☐ Kaktus Database at Kajaani University of Applied Sciences ☐ Library of Kajaani University of Applied Sciences

SYMBOLILUETTELO

ABaTS	Automated Build and Test System
	Automaattinen testausympäristö, joka kääntää ja ajaa määritellyt testit. Järjestelmä automatisoi moduuli- ja regressiotestausta. Testien tulokset kootaan raportiksi. Tällä hetkellä testaa ainoastaan java-kielisiä ohjelmia.
ANT	Another Neat Tool
	Java-kielellä toteutettu työkalu ohjelmien automaattiseen kääntämiseen. Käyttää XML-muotoisia tiedostoja kuvaamaan käännösprosessia ja riippuvuuksia.
class –tiedosto	Käännetty Java-kielinen tiedosto, sisältää tavukoodin ja voidaan suorittaa virtuaalikoneessa.
HTML	Hypertext Markup Language
Java	Sun Microsystemsin kehittämä ohjelmointikieli.
java -tiedosto	Java-kielinen lähdekooditiedosto.
jar –tiedosto	Paketti java-kielisiä luokkia, voi sisältää lisäksi luokkien lähdekoodit.

JUnit	Java-koodin moduulitestaukseen tarkoitettu testiajuri.
JVM	Java virtuaalikone Ohjelma, joka luo keinotekoisen ympäristön java-ohjelmien suoritusta varten. Ajaa lähdekoodeista käännettyä tavukoodia.
XML	Extensible Markup Language Yleiskäyttöinen kuvauskieli.

SISÄLLYS

1 JOHDANTO

Opinnäytetyön toimeksiantaja on kajaanilainen ohjelmistoalan palveluyritys Ebsolut Oy. Yritys oli tuttu työharjoittelusta ja sieltä löytyi helposti mielenkiintoinen aihe opinnäytetyöhön.

Ebsolut Oy:llä on käytössä automaattinen testausympäristö, joka on tradenomi Antti Kemppaisen opinnäytetyön tuotos. ABaTS-järjestelmä toimii Ant-skripteillä. ABaTS hakee testattavan projektin koodit ja JUnit-testitapaukset versionhallinnasta. Lähdekoodi ja testitapaukset käännetään käyttäen projektikohtaista build.xml-tiedostoa. JUnit-testitapaukset ajetaan yksi kerrallaan ja tulokset kerätään html-raporttiin, joka siirretään palvelimelle.

Tämä opinnäytetyö jatkaa ABaTS-testausympäristön kehitystä. Testauksen kattavuuden arviointiin käytetään Emma-ohjelmistoa. Emma on java-koodin testauksen kattavuuden arvioinnin työkalu. Avoimen lähdekoodin lisäksi Emma on houkutteleva vaihtoehto, koska se on ilmainen ja helppokäyttöinen. Emman toimintaan on paneuduttu tarkemmin luvussa 4.

Ohjelmiston testauksen kattavuutta voidaan mitata esimerkiksi testattujen lauseiden, ehtojen, haarojen sekä päätösten ja ehtojen kattavuudella. Testauksen teoriaa on esitelty luvussa 2 ja testauksen kattavuutta luvussa 3.

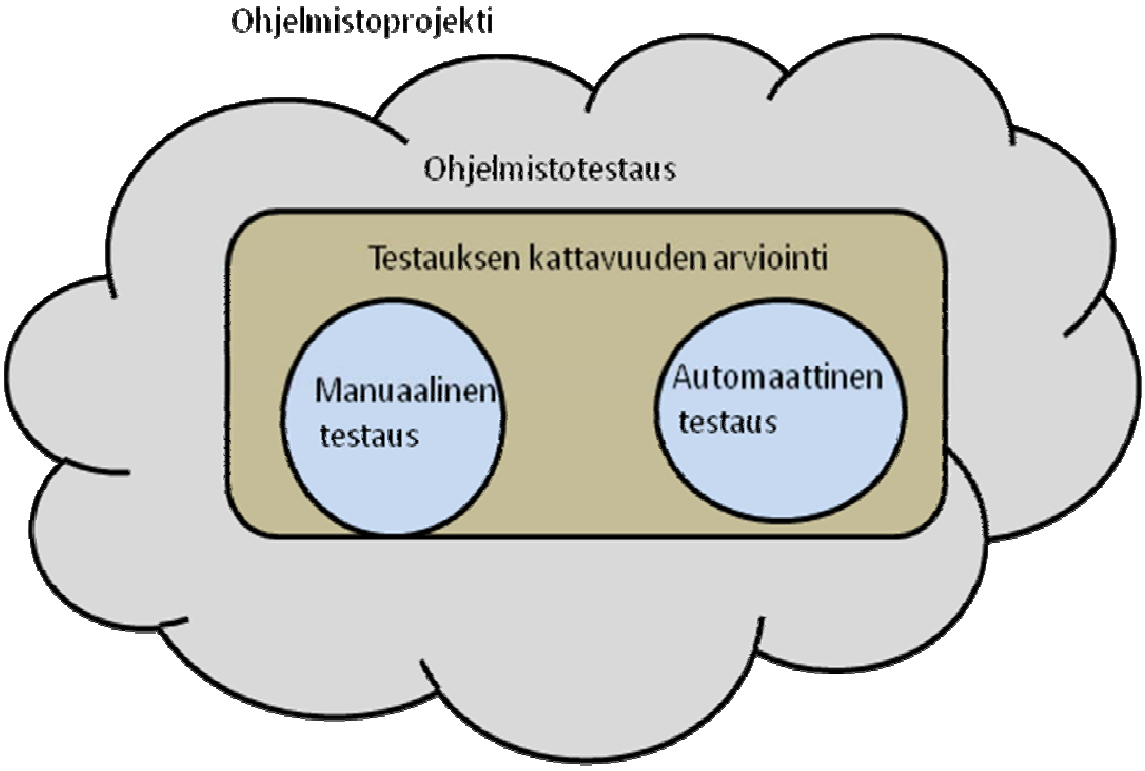
Opinnäytetyön tulisi selvittää vastauksia seuraaviin kysymyksiin. Miten ohjelmiston testauksen kattavuutta voidaan arvioida? Miten kattavasti ohjelmisto tulee testata?

2 OHJELMISTOTESTAUS

Testaus on olennainen osa ohjelmiston kehitysprosessia. Testauksen tavoite on todeta ohjelmiston toimivuus, että se täyttää sille määrittelyssä asetetut vaatimukset, eikä sillä ole määrittelyn ulkopuolisia ominaisuuksia. Testaamalla tuotetta systemaattisesti voidaan löytää siinä mahdollisesti olevat virheet.

Ohjelmiston testauksessa on oleellista selvittää miten kattavaa testaus on. Varsinkin automaattiset testausjärjestelmät voivat käydä ohjelmistoa nopeasti ja useaan kertaan läpi, mutta testauksen kattavuudesta voi olla vaikea saada selkoa. Tähän tarpeeseen on kuitenkin kehitetty työkaluja ja tämän opinnäytetyön tarkoituksena on lisätä Ebsolut Oy:n automaattiseen testausympäristöön testauksen kattavuutta mittaava toiminnallisuus.

Kuviossa 1 on esitelty opinnäytetyön viitekehystä. Ohjelmistotestaus on osa suurempaa kokonaisuutta, ohjelmistoprojektia. Ohjelmistotestaus sisältää manuaalisen- ja automaattisen testauksen, jotka molemmat ovat testauksen kattavuuden arvioinnin alaisena. Ohjelmistotestaus on tietenkin paljon muutakin kuin vain jaottelu manuaaliseen ja automaatomaattiseen testaukseen, mutta tässä kuviossa ei mennä tämän syvemmälle.



Kuvio 1. Viitekehys

2.1 Testauksen tavoite

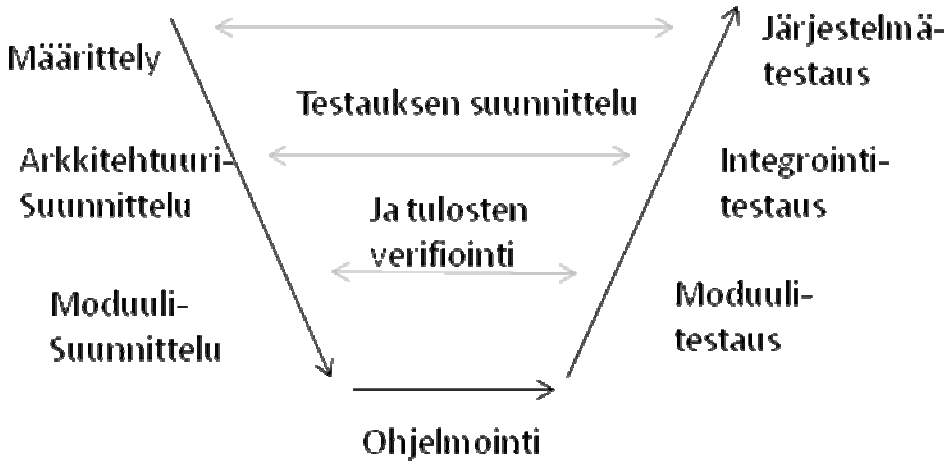
Testaajan tehtävänä ei ole pelkästään löytää virheitä. Testaajan tulisi löytää tuotteen virheet mahdollisimman aikaisessa vaiheessa. Pelkkä virheiden löytäminen ajoissa ei myöskään riitä. Testaajan tulee myös huolehtia siitä, että löydetyt virheet tulevat korjatuksi. Tämä tapahtuu yleisimmin raportoimalla virheestä ohjelmiston kehittäjille. (Patton 2005, 19.)

Testauksella ei voida koskaan todistaa virheiden poissaoloa. Jos virheitä ei löydy, se voi johtua vain siitä, että testitapaukset on valittu siten, ettei virheitä tule esille. Tämän vuoksi testauksen tavoitteeksi olisikin järkevää asettaa virheettömyyden sijasta mahdollisimman monen virheen löytyminen ja niiden korjaaminen. (Kautto 1996.)

2.2 Testausprosessi

Testaus jaetaan yleisesti niin sanotun V-mallin mukaisesti moduuli-, integrointi- ja järjestelmätestaukseen. Joissain tapauksissa järjestelmätestauksen jälkeen voidaan suorittaa erillinen kenttätestaus ja/tai hyväksymistestaus. Muita testaustyyppejä ovat muun muassa alfa- ja betatestaus, regressiotestaus, sekä käytettävyystestaus. (Haikala 2002, 286.)

Kuviossa 2 on kuvattu testauksen V-malli Haikalan mukaan. Mallin mukaan testauksen suunnittelu tulee tehdä testaustasoa vastaavalla suunnittelutasolla. Määrittelyvaiheessa suunnitellaan järjestelmätestaus, arkkitehtuurisuunnittelun yhteydessä integrointitestaus ja moduulisuunnittelun yhteydessä moduulitestaus. Testien tulokset voidaan todistaa oikeiksi vertaamalla niitä vastaaviin suunnitteludokumentteihin. Mahdolliset virheet ovat sitä kalliimpia korjata, mitä korkeammalla ne löytyvät testauksen V-mallissa. (Haikala 2002, 286.)



Kuvio 2. Testauksen V-malli Haikalan mukaan (Haikala, 2002. 287.)

Pöytätestaus

Pöytätestaus on hyvin perinteinen menetelmä, jossa ohjelmoija tarkastelee omaa koodiaan yksin tai ryhmässä. Pöytätestauksessa ohjelmaa käsitellään lähdekoodina, ilman ohjelman ajamista. Pöytätestaus on käyttökelpoinen työväline, koska sen avulla voidaan löytää virheitä jo projektin alkuvaiheessa. Pöytätestaus onkin siksi hyvin kustannustehokas menetelmä. (Patton 2005, 92.)

Koodin tarkastelu ja katselmointi on yksi pöytätestauksen muoto. Katselmointi voi olla kahden ohjelmoijan välinen keskustelu tai yksityiskohtainen ohjelman rakenteen ja koodin läpikäynti. Pattonin mukaan katselmoinnin tulee koostua neljästä perusasiasta:

- **Ongelmien löytäminen**. Katselmoinnin tavoite on löytää ohjelmasta ongelmia, ei pelkästään väärin tehtyjä asioita, vaan myös puuttuvia osia. Kaikki kritiikki tulee kohdistua ohjelman rakenteeseen ja koodiin, eikä osallistujien pidä ottaa sitä henkilökohtaisesti.

- **Sääntöjen noudattaminen.** Katselmoinnilla tulee olla etukäteen asetetut säännöt. Niissä voidaan rajata esimerkiksi katselmoitavan koodin määrää, ajankäyttöä, kommentoitavia asioita. Säännöt auttavat osanottajia tiedostamaan oman roolinsa ja katselmointi etenee sujuvammin.

- **Valmistautuminen**. Jokaisen osallistujan tulee valmistautua osaltaan katselmointiin. Katselmoinnin tyypistä riippuen osallistujan rooli voi vaihdella, mutta yleensä valmistautuminen on koodin läpi käyntiä muistiinpanoja tehden. Osallistujien on tiedettävä oma roolinsa katselmoinnissa ja toimittava aktiivisesti sen mukaisesti. Suurin osa katselmoinnissa löytyvistä ongelmista havaitaan jo valmistautumisvaiheessa, eikä varsinaisessa katselmoinnissa.

- **Tulosten raportointi**. Katselmointiryhmän tulee tuottaa kirjallinen raportti, joka sisältää yhteenvedon katselmoinnissa löytyneistä ongelmista. Tämä raportti tulee julkaista ohjelmistoprojektin kehitysryhmälle. Raportista tulee selvitä löydettyjen ongelmien määrä, niiden sijainti sekä mahdollinen ratkaisumalli.

Katselmoinnin tehokkuus ja hyöty on pidemmän prosessin tuotos. Hyvin toteutettuna katselmointi auttaa löytämään isojakin ongelmia jo projektin alkuvaiheessa. Hätäisesti pidetty palaveri, jossa katsellaan koodin rakennetta, ei ole tehokas, vaan se voi itse asiassa olla jopa haitallinen. Jos katselmoinnin toteutus ei asianmukainen, ongelmia jää huomaamatta ja osanottajien mielestä koko palaveri oli ajan haaskausta. (Patton 2005, 93.)

Ongelmien löytämisen lisäksi katselmoinneilla on muutamia epäsuoria seurauksia, joita Patton luettelee seuraavasti:

- **Kommunikointi**. Katselmointiraportti ei välttämättä sisällä kaikkea mahdollista tietoa, mutta siitä keskustellaan. Tällä tavalla testaajat saavat vihjeitä testitapausten suunnitteluun, kokemattomat ohjelmoijat voivat oppia uusia tekniikoita kokeneemmilta kollegoiltaan ja johto saa tietoa projektin aikataulun pitämisestä.

- **Laatu.** Kun ohjelmoija tietää, että hänen työnsä tulokset tullaan käymään rivi riviltä läpi, mitä luultavimmin hän on varovaisempi. Tämä ei tarkoita, että hän olisi muuten huolimaton, vaan tietoisuus koodin tarkastelusta voi saada hänet tarkkaavaisemmaksi ja varmistamaan, että kaikki toimii oikein.

- **Yhteishenki**. Sujuvasti etenevä katselmointi voi edistää ohjelmoijien ja testaajien kunnioitusta toisiensa ammattitaitoa kohtaan. Samalla eri ohjelmoijat ja testaajat oppivat ymmärtämään toistensa työnkuvaa.

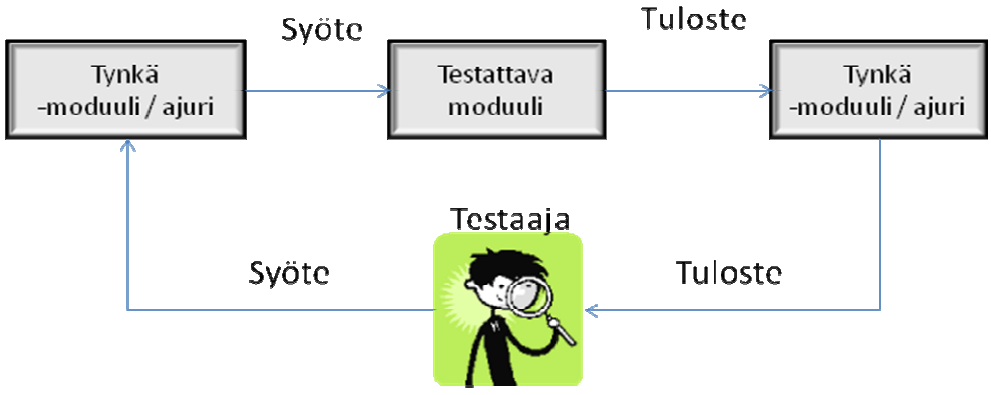
- **Ratkaisut.** Vaikeille ongelmille löytyy usein ratkaisu. Katselmoinnin säännöistä riippuen keskustelu voi olla katselmoinnin yhteydessä tai sen ulkopuolella. Pattonin mukaan ongelmien ratkaisuun liittyvät keskustelut ovat tehokkaampia katselmoinnin ulkopuolella.

Nämä sivuvaikutukset ovat mahdollisia, mutta ne eivät välttämättä toteudu jokaisessa työympäristössä. Usein ohjelmistokehittäjät ajautuvat työskentelemään eristyksissä toisistaan, sen vuoksi katselmoinnit ovat hyvä tapa saada heidät saman pöydän ääreen keskustelemaan projektin ongelmista. (Patton 2005, 93.)

Moduulitestaus

Moduulitestauksen kohteena on yksittäinen ohjelmamoduuli. Yleisimmin yksi moduuli koostuu noin 100 – 1000 koodirivistä. Moduulin toimintaa tarkkaillaan ja tuloksia verrataan suunnitteludokumentteihin, yleisimmin tekniseen määrittelydokumenttiin. Moduulitestauksen suorittaa yleisimmin itse moduulin toteuttaja. (Haikala, 2002.)

Moduulitestausta varten voidaan joutua toteuttamaan erinäisiä testipenkkejä, joiden avulla moduulin toimintaa voidaan kokeilla. Kuviossa 3 on esitetty kaavio testipenkistä. Testaaja syöttää haluamansa syötteen tynkämoduulin kautta testattavaan moduuliin, joka luulee olevansa osa valmista ohjelmistoa. Testattava moduuli tekee tehtävänsä ja antaa tulosteensa toiselle tynkämoduulille, joka näyttää tuloksen testaajalle. Tällainen testipenkki on tarpeen, mikäli muita ohjelmiston moduuleita ei ole vielä kehitetty. (Seppänen, 2005.)



Kuvio 3. Testipenkki Seppäsen mukaan (Seppänen, 2005.)

Integraatiotestaus

Integraatiotestauksessa testataan moduulien yhteistoimintaa. Lähinnä tarkastelussa ovat jokaisen moduulin rajapinnat ja kommunikointi toisten moduulien kanssa. Testien tuloksia verrataan yleisimmin tekniseen määrittelyyn. Integraatiotestaus tehdään yleensä samaan aikaan moduulitestauksen kanssa, ja niitä on usein tarpeetonta tarkastella erillisinä testauksina. Testauksen kattavuuden kannalta moduulitestauksessa on helpompi saavuttaa parempi kattavuus kuin integraatiotestauksessa, sillä testattavan ohjelmiston kasvaessa kaikkea koodia on vaikeampi käydä läpi. Integrointitestaus voidaan suorittaa joko kokoavasti alimman tason moduuleista ylöspäin tai jäsentävästi eli osittaen päinvastaiseen suuntaan. Kokoavasta- (bottom up) ja jäsentävästä- (top-down) –testauksesta löytyy lisää tietoa luvussa 2.3. (Haikala 2002, 287.)

Systeemitestaus

Systeemitestauksen tai järjestelmätestauksen kohteena on koko järjestelmä. Testin tuloksia suhteutetaan määrittelydokumentaatioihin sekä asiakkaan määrittelemiin tarpeisiin. Järjestelmätestaus tulisi suorittaa kehitystyöstä riippumattomien testaajien toimesta, jotta tulokset olisivat mahdollisimman oikeita. Tässä vaiheessa testataan myös järjestelmän ei-toiminnalliset ominaisuudet, kuten kuormitus, käytettävyys, luotettavuus, asennus, jne. (Haikala 2002, 288.)

Regressiotestaus

Regressiotestaus on yleisnimitys uudelleen testaamiselle järjestelmään tehtyjen muutosten jälkeen. Tällä testauksella varmistetaan, etteivät uudet muutokset järjestelmään ole vaikuttaneet aiemmin testattujen osien toimintaan. Regressiotestauksen kustannuksia voidaan vähentää huomattavasti automatisoimalla testaus. (Haikala 2002, 288.)

Regressiotestauksessa tulee valikoida oleelliset testitapaukset, koska esimerkiksi kaikkien integrointitestitapausten ajaminen on yleensä liian kallista ja aikaa vievää. Rajaamalla suoritettavien testitapausten joukkoa oikein voidaan vähentää regressiotestejä jopa puolella. Regressiotestien valintaan voidaan käyttää seuraavanlaista tekniikkaa. Kutsutaan alkuperäistä ohjelmaa nimellä P ja muutettua ohjelmaa P'. Testijoukko T muodostuu alkuperäisen ohjelman ajetuista testeistä. Osa muutetun ohjelman testeistä koostuu alkuperäisten testien joukosta

T'. Lisäksi muutoksen mukanaan tuomia uusia ominaisuuksia ohjelmaan P' varten on luotava uusi testijoukko T". Ohjelman P' testit koostuvat siis testijoukoista T, T' ja T". (Huhtamäki 2001, 10.)

Käytettävyystestaus

Ohjelmistoja tehdään käytettäväksi. Tämä on aika itsestäänselvyys, mutta se voi joskus unohtua monimutkaisen tuotteen suunnittelun, toteutuksen ja testauksen kiireessä. Loppujen lopuksi valmis ohjelmisto tulee "keskustelemaan" loppukäyttäjän kanssa. Käytettävyys on sitä kuinka tarkoituksenmukaista, toiminnallista ja tehokasta käyttäjän vuorovaikutus ohjelmiston kanssa on. (Patton 2006, 169.)

Koska käytettävyys on käyttäjän ja ohjelmiston välistä vuorovaikutusta, käytettävyystestaus on lähinnä käyttöliittymän testausta. Käytettävyystestit suoritetaan valitsemalla pieni joukko ohjelmiston loppukäyttäjiä ja seuraamalla heidän suoriutumista erilaisista tehtävistä valvotussa koetilanteessa. Testaus voidaan suorittaa erillisessä käytettävyyslaboratoriossa, jossa on laitteistoa käyttäjän toimien rekisteröintiin. Tavallisimmin testihenkilön toiminta videoidaan ja testihenkilöä kehotetaan ajattelemaan ääneen. Tällä tavalla saadaan selville millaiseen järkeilyyn testihenkilön toiminta perustuu. Varsinkin ensimmäisillä kerroilla käytettävyystestien tulokset voivat olla hyvin yllättäviä ohjelmiston kehittäjille. (Haikala 2002, 289.)

2.3 Dynaaminen analyysi

Dynaaminen analyysi keskittyy ohjelmiston ja koodin käyttäytymiseen suorituksen aikana. Testauksen edellytyksenä on siis toimiva ohjelma tai ympäristö, jossa voidaan suorittaa yksittäisiä funktioita. (Kautto 1996.)

Tässä luvussa on esitelty yleisimmin käytettyjä dynaamisen analyysin testausmenetelmiä. Testausmenetelmän valinta on riippuvainen testattavasta kohteesta, testausvaiheesta ja kohteen määrittelystä. Yhden menetelmän käyttö ei sulje toisia pois, vaan niitä käytetäänkin usein rinnakkain toisiaan täydentäen. (Kautto 1996.)

Mustalaatikko – Black Box

Mustalaatikkotestauksessa ohjelmistoa, tai sen osaa, testataan ilman tarkkaa tietoa suoritettavasta ohjelmakoodista. Tässä testausmenetelmässä testaaja käyttää ohjelmistoa hyvin pitkälti samoin kuin varsinainen loppukäyttäjä käyttäisi. Ohjelmalle annetaan syötteitä ja tarkastetaan saatujen tulosteiden oikeellisuus. Dynaamista mustalaatikkotestausta kutsutaan myös käyttäytymistestaukseksi (behavioral testing), koska siinä testataan kuinka ohjelmisto käyttäytyy kun sitä käytetään. (Patton 2005, 64.)

Tehokasta mustalaatikkotestausta varten tulee tutustua tarkasti ohjelmiston vaatimusmäärittelyyn ja muihin suunnitteludokumentteihin. Testaajan ei tarvitse tietää mitä ”laatikon” sisällä tapahtuu, vain se, että syötteellä A pitäisi tulostua B tai toiminto C:n suorittaminen johtaa toimintoon D. Hyvin laadittu ohjelmiston määrittely kertoo nämä asiat. (Patton 2005, 64.)

Lasilaatikko – White Box

Nimensä mukaisesti lasilaatikkotestauksessa testin suorittaja näkee testattavan moduulin sisälle ja on tietoinen siitä kuinka ohjelman suoritus etenee. Ajettavat testitapaukset valitaan ja laaditaan testattavan ohjelmakoodin perusteella. Tavoitteena on käydä läpi kaikki kohteen haarat ja ohjelmapolut. Testauksen kattavuuden arviointia on käsitelty luvussa 2.4. (Kautto 1996.)

Harmaalaatikko – Gray Box

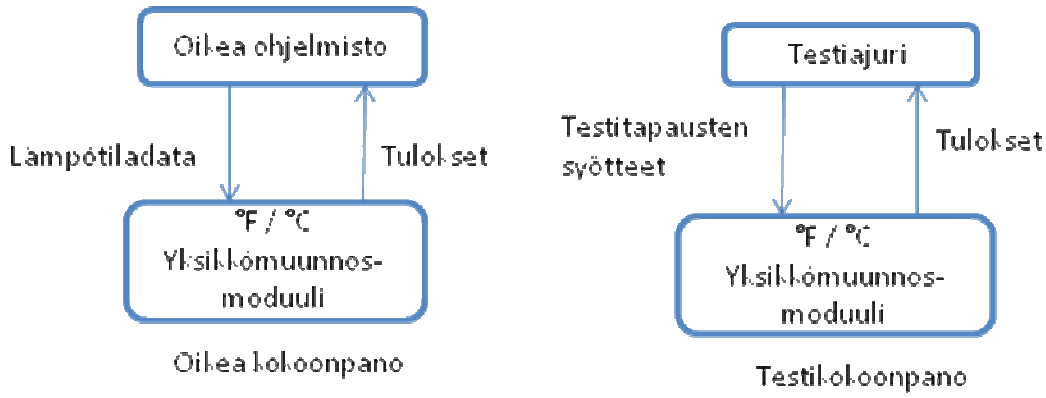
Harmaalaatikkotestaus yhdistää musta- ja lasilaatikkotestauksen ominaisuuksia. Pelkällä mustalaatikkotestauksella voi jäädä löytymättä esimerkiksi lähdekooditason tietovirta- ja rajaarvovirheitä. Lasilaatikkotestaus taas ohittaa käyttöliittymään, yhteensopivuuteen ja käyttöjärjestelmään liittyvät virheet. (Oamk 2006.)

Harmaalaatikkotestaus tarkastelee testattavan kohteen teknistä toteutusta (ohjelmakoodi), käyttäjäpuolen lopputulosta (käyttöliittymä) sekä käyttöjärjestelmää. Testitapaukset valitaan ja laaditaan lähdekoodia tarkastelemalla. Harmaalaatikkotestaus evaluoi järjestelmän eri osien yhteensopivuutta järjestelmän suunnitelman mukaan. Tämä testaustapa sopii hyvin websovellusten testaamiseen, koska ne koostuvat monista eri laitteisto- ja ohjelmistopuolen

komponenteista, joiden yhteensopivuus on varmistettava. Harmaalaatikkotestaus paljastaa yleensä mahdolliset ongelmat liittyen tiedon kulkuun järjestelmän päästä päähän sekä viallisiin ohjelmiston tai laitteiston määrityksiin. (Oamk 2006.)

Bottom-up

Bottom-up –strategian mukaan testaus aloitetaan alimman tason moduuleista. Alin taso määritellään tasoksi, jonka moduulit eivät kutsu tai käytä muita moduuleja. Testaus etenee tasoittain ylöspäin, kunnes koko ohjelma on käyty läpi. Testausta varten joudutaan tekemään testiajureita, joiden avulla alemman tason moduuleja pystytään suorittamaan ja testaamaan. Kuviossa 4 on esitetty malli testiajureista. Oikean ohjelman tilalle vaihdetaan testiajuri, joka syöttää testitapausten syötteet testattavaan moduuliin ja tarkistaa sieltä palautuvien tulosten oikeellisuuden. (Patton 2005, 109.)



Kuvio 4. Testiajuri Pattonin mukaan (Patton 2005, 110.)

Bottom-up –menetelmän eduksi voidaan laskea testiaineiston suhteellisen helppo valinta. Mikäli järjestelmän virhealteimmat tai kriittisimmät moduulit sijoittuvat alimmille tasoille, bottom-up –menetelmä on hyvä valinta. Lisäksi testiajureiden avulla voidaan testitapauksissa käyttää määrältään ja laadultaan vaihtelevia syötteitä. Myös sellaisia, joiden käyttö olisi vaikeampaa korkeammalla tasolla testattuna. Toisaalta tätä menetelmää käyttämällä minkäänlaista valmista järjestelmää ei ole olemassa, ennen kuin kaikki ylimmän tason moduulit on testattu. (Patton 2005, 109.)

Top-down

Nimensä mukaisesti top-down testaus etenee ylimmän tason moduuleista kohti alinta tasoa. Testausta varten alemman tason moduulit korvataan niin sanotuilla ”tyhmillä” moduuleilla tai moduulin pätkillä (module stubs, stubs). Stub-moduulit eivät sisällä monimutkaista toiminnallisuutta ja ovat hyvin yksinkertaisia. Testauksen edetessä seuraavalle tasolle, tynkämoduulit korvataan aidoilla moduuleilla. (Patton 2005, 109.)

Ylhäältä alaspäin etenevän testauksen hyötynä on aikaisessa vaiheessa saatava toimiva ohjelma. Vaikka kaikkea toiminnallisuutta ei ole vielä implementoitu, kehittäjillä on käsitys ohjelman toiminnasta. Lisäksi tällä tavalla voidaan yhdistää mahdollisesti erillisinä suoritettavat testausvaiheet, integraatio- ja systeemitestaus. Suurin yksittäinen haitta on erillisten tynkämoduulien tekemiseen menevä aika. Lisäksi monimutkaisten tynkämoduulien toiminta ei välttämättä täysin vastaa oikean moduulin toimintaa. (Patton 2005, 109.)

3 TESTAUKSEN KATTAVUUS

Testauksen kattavuutta voidaan mitata erilaisilla metodeilla. Lauseiden kattavuus-metodi (statement coverage) laskee suoritettuja ohjelmarivejä tavoitteenaan saada kaikki ohjelman lauseet suoritettua ainakin yhteen kertaan. Käytännössä tämä ei ole kovin käytännöllinen menetelmä, sillä testitapausten määrä kasvaa helposti liian suureksi ohjelmiston monimutkaisuuden ja koon myötä. Lisäksi tulee huomioida, ettei täydellinen lausekattavuus tarkoita täydellistä testausta, sillä se ei takaa kaikkien ohjelman haarojen kattavuutta. (Patton 2005, 118; Haikala 2002, 292.)

Haarojen kattavuus-menetelmä (branch / path coverage) käy läpi ohjelmiston haarojen kaikki vaihtoehdot. Haaralla tarkoitetaan ohjelmiston kohtaa, jossa ohjelman suorituksella on kaksi tai useampia vaihtoehtoisia suoritettavia lauseita. Käytännössä nämä ovat ohjelmassa olevia if-else, switch-case ja muita ehtorakenteita. Samoin kuin lausekattavuus, haarakattavuus sopii lähinnä yksittäisten metodien testaukseen, kun mahdollisia suorituspolkuja on vielä järkevä määrä. Laajemmissa ohjelmarakenteissa mahdollisten haarojen määrä kasvaa kohti ääretöntä. (Haikala 2002, 293.)

Ehtokattavuus (condition coverage) tarkoittaa kaikkien päätösten tekemiseen vaadittavien arvojen testaamista kaikilla vaihtoehdoilla. Tavallisimmin tämä tarkoittaa if-rakenteiden läpikäyntiä true- ja false-arvoilla. (Haikala 2002, 293.)

Päätösten kattavuus-metodin (decision coverage) mukaan toimittaessa testitapausten valinta tulee tehdä siten, että kaikki ehtorakenteet saavat kaikki mahdolliset tulokset. Usein ehdolliset lauseet peittävät tai sisältävät toisia ehtorakenteita ja tämän vuoksi kaikkia vaihtoehtoisia tuloksia ei saada testattua. Ehtokombinaatioiden kattavuus (multiple-condition coverage) – metodi paikkaa tämän vajavuuden, sillä sen mukaan valittujen testitapausten tulee kattaa kaikki mahdolliset tulosyhdistelmät. Tämänkin metodin käytössä mahdolliset tulosyhdistelmät kasvavat usein liian suuriksi, jotta menetelmä voitaisiin suorittaa täydellisesti. (Haikala 2002, 293.)

Päätös/ehtokattavuus on nimensä mukaisesti yhdistelmä päätös- ja ehtokattavuusmetodeista. Sen etuna on yksinkertaisuus ilman osiensa heikkouksia. (Cornett 2007.)

3.1 Testausstrategia

Cornett esittelee artikkelissaan yksinkertaisen strategian testauksen kattavuuden lisäämiseksi. Testauksen perimmäinen tavoite on löytää mahdollisimman monia virheitä ohjelmasta mahdollisimman pienellä vaivalla. Nopea tapa nostaa testauksen kattavuutta on käydä kaikki sovelluksen osat läpi muutamalla testitapauksella. Tässä vaiheessa ei ole oleellista keskittyä yhden osion läpikäymiseen, vaan tarkoituksena on löytää ns. ”helpot” virheet nopeasti mahdollisimman vähällä testauksella. (Cornett 2007.)

Cornettin mukaan yksi tapa toteuttaa tätä testausstrategiaa on seuraava:

- Kutsu ainakin yhtä metodia 90 %:sta testattavia luokkia.
- Kutsu 90 % kaikista metodeista.
- Saavuta 90 % päätös/ehtokattavuus jokaisessa metodissa.
- Saavuta 100 % päätös/ehtokattavuus.

Tässä tulee huomioida, että välitavoitteissa ei tarvitse saavuttaa 100 %:n kattavuutta. Koska täydellisen kattavuuden saavuttaminen voi olla hyvinkin vaikeaa ja aikaa vievää, on testauksen kannalta tehokkaampaa testata ohjelman muita osia. (Cornett 2007.)

3.2 Mikä on riittävä testaus?

Kuten kaikkien näiden kattavuusmetodien kohdalla tuli ilmi, täysin kattava testaus on usein mahdotonta. Tavoiteltava kattavuus ja muut hyväksymiskriteerit testauksen lopettamiselle, tulee määritellä testaussuunnitelmassa. Toiminnaltaan kriittisissä sovelluksissa, kuten esim. sairaanhoidossa ja turvallisuusalalla, kattavuuden tulisi tietysti olla korkeampi kuin muissa sovelluksissa. Yksikkötestauksessa kattavuustavoite voidaan asettaa korkeammalle kuin järjestelmätestauksessa, sillä moduulitasolla tehty virhe voi aiheuttaa useita virhetilanteita järjestelmätestauksessa. (Haikala 2002, 291; Cornett 2007.)

Joidenkin asiantuntijoiden mukaan täytyy aina pyrkiä 100 %:n kattavuuteen testauksessa, koska vain se takaa ohjelmiston laadun. Tässä tulisi kuitenkin huomioida käytetty työmäärä

lähestyttäessä 100 %:n kattavuutta. Usein voitaisiin löytää useampia virheitä lopettamalla yksikkötestaus 85 – 90 %:n tienoilla ja käyttämällä säästetty aika järjestelmätestaukseen, tai järjestelmän tekniseen arviointiin. (Cornett 2007.)

Cornettin mukaan ohjelmiston kehitysvaiheessa ja lopputestauksessa tulee käyttää samaa kattavuusmetodia. Tällä tavoin vaikean testitapauksen kohdalla voi helpommin arvioida voisiko sen toteutusta viivyttää ja lisätä testauksen tehokkuutta siirtymällä seuraavaan testitapaukseen. (Cornett 2007.)

Riittävää testausta määritettäessä on myös huomioitava käytettävä kattavuusmitta. Julkaisuversion testauksessa tavoiteltava kattavuus voi olla esim. 80 – 90% käytettäessä lause- tai päätöskattavuutta. (Cornett 2007.)

4 EMMA

Emma on työkalu, jolla voidaan helposti mitata ja raportoida Java –kielisten ohjelmien testauksen kattavuutta. Emma perustuu avoimeen lähdekoodiin ja sen käyttö on ilmaista kaupallisessa ja yksityisessä käytössä. (Roubtsov 2006.)

Emman vahvuus on sen helppokäyttöisyys. Lisäksi Emma on puhtaasti java-kielinen, eikä se tarvitse mitään ulkoisia kirjastoja toimiakseen. Emmaa voidaan käyttää millä tahansa Java 2 –virtuaalikoneella versiosta 1.2 eteenpäin. (Roubtsov 2006.)

Emma-kirjaston toiminta perustuu java-virtuaalikoneen ja ajettavan ohjelman väliin lisättävään kerrokseen. Lisätyn kerroksen avulla Emma kuuntelee virtuaalikonetta ja kerää tietoa suoritetuista ohjelman osista. Tätä kerroksen lisäämistä kutsutaan instrumentoinniksi. Emma voi instrumentoida tarvittavat luokat ”lennosta” testattavaa ohjelmaa ajettaessa (online) tai instrumentointi voidaan tehdä valmiiksi halutuista luokista (offline). ABaTS –ympäristössä valmiiksi tehty instrumentointi oli sopiva vaihtoehto, koska yöllisessä testiajossa suoritusajalla ei ole suurta merkitystä. (Roubtsov 2006.)

Emma-kirjaston kaikkia toimintoja voidaan käyttää suoraan komentoriviltä tai Ant-skripteistä. Komentorivityökaluilla voidaan pystyttää testiympäristö nopeasti satunnaista käyttöä varten, kun taas Ant-skriptit ovat sopivampia ABaTS:in kaltaisiin järjestelmiin. Komentorivityökaluilla oli myös helppo testata Emman toimivuus ABaTS –järjestelmässä ennen varsinaista Ant-skriptausta. (Roubtsov 2006.)

Ennen ohjelman suorittamista lähdekoodi tulee kääntää class-tiedostoiksi. Kääntämisen jälkeen on suoritettava koodin instrumentointi, joka lisää tavukoodiin tarvittavat muutokset ohjelman suorituksen seuraamiseksi sekä kirjoittaa levylle tarvitsemansa metadatatiedot. Instrumentointi on tavallaan toinen kääntäminen, sen tuloksena on instrumentoidut class-tiedostot. Ohjelma suoritetaan normaalisti instrumentoiduista class-tiedostoista ja Emma kirjoittaa ajonaikaisen seurantansa tulokset tiedostoon. Lopuksi instrumentoinnissa luotu metadata ja ajonaikainen seurantatieto yhdistetään helppolukuiseksi raportiksi. (Roubtsov 2006.)

4.1 Emman asennus ja käyttö

Moduulin asennukseksi riittää emma.jar-paketin kopiointi virtuaalikoneen asennuskansion alle /lib/ext –kansioon. Mikäli kirjastoa tullaan käyttämään Ant-skriptillä, skriptin alussa täytyy kertoa kansio, josta emma.jar ja emma_ant.jar voidaan ladata. Näiden toimenpiteiden jälkeen Emma-kirjasto on käytettävissä.

Emman käyttö voidaan jakaa vaiheisiin seuraavasti:

- lähdekoodin kääntäminen
- käännettyjen luokkien instrumentointi
- ohjelman ajaminen instrumentoiduista luokista
- raportin luonti.

Seuraavaksi käydään läpi nämä vaiheet sekä komentorivityökaluilla, että Ant-skriptien kannalta.

4.1.1 Lähdekoodin kääntäminen

Java kielinen lähdekoodi tallennetaan java–tiedostopäätteellä. Lähdekoodi on kokoelma selväkielisiä käskyjä, jotka noudattavat ohjelmointikielen sisäistä syntaksia. Lähdekoodi voidaan kirjoittaa esim. Muistion kaltaisella tekstieditorilla tai erillisessä ohjelmointikehitysympäristössä. Lähdekoodi on käännettävä virtuaalikoneen ymmärtämään muotoon, jota kutsutaan tavukoodiksi. Tavukoodi on kokoelma virtuaalikoneen ymmärtämiä konekäskyjä.

Kääntäminen tapahtuu muuten normaalisti, mutta kääntäjälle on annettava parametrina tieto koodin debug-ominaisuuksien käyttämisestä. Debug tarkoittaa tässä yhteydessä virheenkorjausta. Debug–ominaisuuksia tarvitaan, jotta virtuaalikone osaa antaa kaikki tarvittavat tiedot Emman käyttöön raportointia varten.

Komentorivillä kääntäminen tapahtuu javac–komennolla. Esimerkkilauseen ensimmäinen parametri (–d out) asettaa kansion, jonne käännetyt luokat tallennetaan. Toinen parametri (-

g) lisää tarvittavat debug-tiedot luokkiin. Lopuksi annetaan käännettävien java-tiedostojen sijainti, tässä tapauksessa käännetään alikansioissa src ja src\search olevat java-päätteiset tiedostot.

```
javac -d out -g src/*.java src/search/*.java
```

Vastaava toiminto Ant-skriptillä tapahtuu seuraavasti:

```
<target name="compile">
    <javac debug="on" srcdir="${src.dir}" destdir="${out.dir}" />
</target>
```

Esimerkissä Ant-kohteen nimi on compile. ”Metodi” sisältää käskyn javac, jolle annetaan tieto debug-toimintojen käytöstä sekä lähde- ja kohdekansiot. Lähdekoodien sijainti on asetettu muuttujaan src.dir ja kohdekansio vastaavasti muuttujaan out.dir. Muuttujat voidaan asettaa seuraavasti:

```
<property name="src.dir" value="${basedir}/src" />
<property name="out.dir" value="${basedir}/out" />
```

4.1.2 Instrumentointi

Lähdekoodien kääntämisen jälkeen seuraava vaihe on käännettyjen luokkien instrumentointi. Instrumentointiin on kaksi vaihtoehtoa, etukäteen tehtävä tai suorituksen aikainen. Tässä luvussa on käsitelty molempien toteutus.

Online instrumentation

Suorituksen aikainen instrumentointi on joustava ja nopea tapa Emman käyttöön. Ohjelmaa ajettaessa lisätään käännetyt luokkatiedostot luokkapolkuun ja komennetaan Emma online instrumentointiin. Muuten ohjelma käynnistetään normaalisti. Komentoriviltä tämä tapahtuu seuraavasti:

```
java emmarun -cp out Main
```

Java-komento käynnistää virtuaalikoneen. Emmarun kutsuu emma-kirjaston toimintoa, joka käynnistää ajonaikaisen instrumentoinnin. Seuraava parametri on luokkapolkumääritys, jonka mukaan out-kansiossa sijaitsevat luokkatiedostot ajetaan. Lopuksi kerrotaan luokka, jossa on ajettavan ohjelman käynnistävä static void main(String []args) –metodi.

Ant-skriptissä vastaava toiminto määritellään seuraavasti:

```
<target name="emma" description="turns on EMMA's on-the-fly instrumen-
tation mode" >
    <property name="emma.enabled" value="true" />
  </target>


<target name="run">
    <emmajava enabled="${emma.enabled}" libclasspathref="emma.lib"
              classname="Main"
              classpathref="run.classpath"
    >
    </emmajava>
</target>
```

Esimerkissä ensimmäinen kohde on <emma>, joka asettaa muuttujan emma.enabled ja mahdollistaa ajonaikaisen instrumentoinnin. Seuraavassa kohteessa ohjelman ajamiseen tarkoitettu tavallinen java-komento on korvattu Emman emmarun-komennolla. Emmarun tarkistaa emma.enabled muuttujan ja lisää emma-kirjaston ajettavaan luokkapolkuun. Sovelluksen käynnistävä luokka asetetaan samoin kuin komentorivillä. Lopuksi käännetyt luokkatiedostot lisätään ajettavaan luokkapolkuun.

Offline instrumentation

Etukäteen tehtävää instrumentointia kutsutaan offline instrumentoinniksi. Vaikka ajonaikainen instrumentointi on kätevä työkalu, se ei ole aina riittävä. Varsinkin jos testiympäristön luontiin käytetään erilaisia testiajureita, etukäteen tehtävä instrumentointi on suositeltava vaihtoehto.

Nimensä mukaisesti offline instrumentointi tehdään erikseen ennen ohjelman suoritusta. Toisaalta tämä lisää yhden työvaiheen Emman käyttöön, mutta se myös selkeyttää testauksen suoritusta.

Komentorivillä erillinen instrumentointi tehdään seuraavasti:

```
java emma instr -d outinstr -ip out
```

Komento käynnistää virtuaalikoneen ja kutsuu Emma-kirjaston instr-toiminnallisuutta. Perustoiminnallisuuteen tarvitaan kaksi parametria, -d asettaa kansion, jonne instrumentoidut luokkatiedostot tallennetaan ja –ip määrittää kansion, josta instrumentoitavat tiedostot löytyvät.

Perustoiminnallisuus luo raportointia varten tarvittavan metadata-tiedoston nykyiseen kansioon nimellä coverage.em. Tiedoston polku ja nimi voidaan asettaa lisäämällä parametri

```
–out kansio\tiedosto.nimi
```

Oletuksena vain instrumentoidut luokat kopioidaan kohdekansioon. Tämä voi aiheuttaa ongelmia ohjelmaa ajettaessa. Kopioinnin sijasta instrumentointi voidaan toteuttaa ylikirjoitustilassa, joka nimensä mukaisesti ylikirjoittaa vanhat luokkatiedostot instrumentoiduilla versioilla. Kolmas vaihtoehto on kopioida kaikki luokat ja kirjastot instrumentoinnista välittämättä. Fullcopy-metodia käytettäessä kohdekansio jaetaan kahtia classes- ja lib-kansioihin. Instrumentoidut luokat kopioidaan classes– ja jar-paketteina olevat kirjastot lib-kansioon. Instrumentoinnin toiminto valitaan seuraavasti:

```
–m copy/overwrite/fullcopy
```

Ant-skriptillä toteutettuna instrumentointi näyttää seuraavalta:

```
<emma enabled="${emma.enabled}" >
      <instr instrpathref="${out.dir}/classes"
             mode="fullcopy"
             outdir="${out.instr.dir}">
      </instr>
</emma>
```

Esimerkissä instrumentoitavat luokat löytyvät muuttujaan out.dir tallennetun kansion alikansiosta \classes. Instrumentointi toteutetaan kopioimalla kaikki luokat out.instr.dir-muuttujaan tallennettuun kansioon.

4.1.3 Testin ajaminen

Offline instrumentointia käytettäessä testattava ohjelma pitää ajaa erikseen. Ohjelma ajetaan normaaliin tapaan virtuaalikoneessa, mutta instrumentoidut luokat tulee lisätä luokkapolkuun ensimmäisiksi ennen instrumentoimattomia luokkia. Komentoriviä käytettäessä komento on:

```
java -cp outinstr;out Main
```

Esimerkissä java-komento käynnistää virtuaalikoneen ja parametri –cp asettaa luokkapoluksi kansiot outinstr ja out. Instrumentoidut luokat ovat ennen instrumentoimattomia luokkia. Lopuksi kerrotaan luokka, joka sisältää ohjelman käynnistävän metodin static void main(String []args).

Ant-skriptillä vastaava toiminto toteutetaan näin:

```
<java classname="Main" fork="true" >
      <classpath>
        <pathelement location="${out.instr.dir}" />
        <path refid="run.classpath" />
        <path refid="emma.lib" />
      </classpath>
      <jvmarg value=
        "-Demma.coverage.out.file=${coverage.dir}/coverage.emma" />
      <jvmarg value="-Demma.coverage.out.merge=true" />
</java>
```

Virtuaalikone käynnistetään <java> -merkinnällä ja samalla annetaan ohjelman käynnistävän luokan nimi classname-parametrilla. Luokkapolku määritetään <classpath> -elementissä ja siinä ensimmäisenä on määritetty instrumentoitujen luokkien sijainti. Lisäksi on annettu instrumentoimattomien luokkien sijainti run.classpath-muuttujasta ja emma-kirjaston sijainti emam.lib–muuttujasta. Lopuksi kaksi parametria välitetään virtuaalikoneelle. Ensimmäisenä määritetään minne Emma tallentaa ajonaikaisen seurantansa tulokset. Tässä tapauksessa se tapahtuu coverage.dir-muuttujassa määriteltyyn kansioon coverage.emma–tiedostoon. Toinen virtuaalikoneen parametri asettaa Emman yhdistämään uudet ajonaikaisen seurannan tulokset aikaisempiin, eikä vanhoja ylikirjoiteta.

4.1.4 Raportointi

Raportointi on oleellinen osa Emman toimintaa. Emma koostaa raporttinsa yhdistämällä instrumentoinnissa luodun luokkien metadatan ajonaikaisen seurannan tuloksiin. Raportit voidaan luoda yksinkertaisen tekstitiedoston lisäksi html- tai xml-muodossa. Html-raportti on näistä helppolukuisin, sitä voidaan tarkastella tavallisella web-selaimella.

Raporteissa eritellään testauksen kattavuus luokkien, metodien, lohkojen ja rivien osalta. Nämä kattavuusmitat lasketaan sekä kaikkien pakettien osalta, että paketeittain eriteltynä. Liitteessä 1 on esimerkki Emman tuottamasta tekstiraportista. Kaikki tässä esitellyt raportit ovat esimerkkiraportteja Emman kotisivuilta ja ne ovat tarkasteltavissa www-osoitteessa http://emma.sourceforge.net/samples.html.

Kuviossa 5 on nähtävissä Emman luoman HTML-raportin etusivu. Etusivulla on aluksi yhteenveto kaikkien pakettien luokka-, metodi-, lohko- ja rivikattavuudesta. Lisäksi samat tiedot on eritelty jokaisen paketin osalta.

EMMA Coverage Report (generated Tue May 18 22:13:27 CDT 2004)

[all classes]

OVERALL COVERAGE SUMMARY

name	class, %	method, %	block, %	line, %
all classes	86% (132/153)	62% (884/1429)	56% (24654/44008)	58% (5260.5/9141)

OVERALL STATS SUMMARY

total packages:	26
total executable files:	136
total classes:	153
total methods:	1429
total executable lines:	9141

COVERAGE BREAKDOWN BY PACKAGE

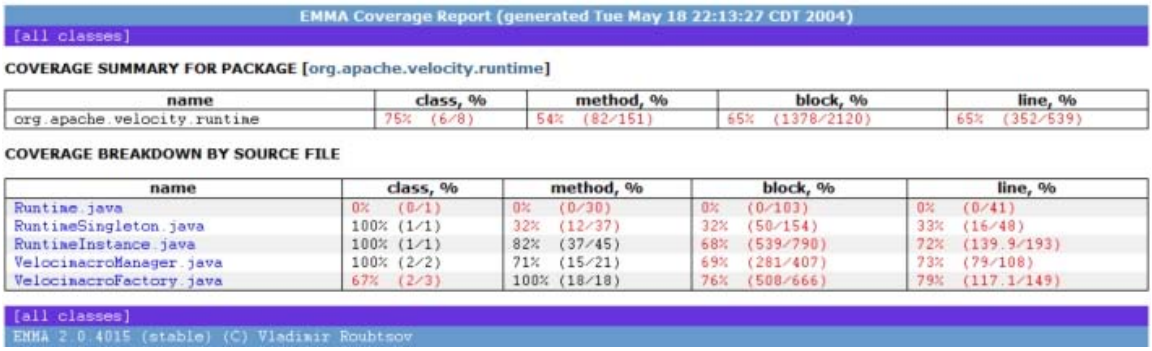
name	class, %	method, %	block, %	line, %
org.apache.velocity.app.tools	0% (0/3)	0% (0/23)	0% (0/338)	0% (0/66)
org.apache.velocity.convert	0% (0/1)	0% (0/10)	0% (0/447)	0% (0/65)
org.apache.velocity.io	0% (0/1)	0% (0/20)	0% (0/348)	0% (0/97)
org.apache.velocity.runtime.compiler	0% (0/1)	0% (0/2)	0% (0/330)	0% (0/50)
org.apache.velocity.servlet	100% (1/1)	33% (6/18)	16% (65/399)	18% (18.5/102)
org.apache.velocity.runtime.visitor	67% (2/3)	26% (20/77)	28% (152/551)	32% (44/137)
org.apache.velocity.runtime.log	57% (4/7)	32% (10/31)	31% (288/919)	29% (64.2/225)
org.apache.velocity.app	100% (3/3)	39% (22/56)	32% (252/798)	35% (70/203)
org.apache.velocity.util	100% (4/4)	47% (16/34)	38% (315/832)	43% (83.4/192)
org.apache.velocity.anakia	91% (10/11)	42% (35/83)	43% (588/1382)	44% (138.9/319)
org.apache.velocity.runtime.configuration	100% (3/3)	64% (37/58)	45% (748/1666)	51% (186.5/368)
org.apache.velocity.runtime.exception	50% (1/2)	50% (1/2)	50% (26/52)	50% (2/4)
org.apache.velocity.runtime.parser	88% (7/8)	82% (241/293)	58% (12411/21359)	58% (2421.5/4144)
org.apache.velocity.runtime.directive	91% (10/11)	84% (56/67)	60% (1186/1988)	68% (278.4/411)
org.apache.velocity.runtime.resource	100% (5/5)	74% (25/34)	60% (494/824)	63% (126.4/200)
org.apache.velocity	100% (2/2)	71% (10/14)	62% (172/279)	65% (46.1/71)
org.apache.velocity.texen	100% (1/1)	83% (20/24)	62% (287/462)	66% (70.2/107)
org.apache.velocity.runtime	75% (6/8)	54% (82/151)	65% (1378/2120)	65% (352/539)
org.apache.velocity.app.event	100% (1/1)	86% (6/7)	65% (85/130)	59% (26/44)
org.apache.velocity.runtime.resource.loader	100% (6/6)	79% (27/34)	67% (573/860)	69% (133/192)
org.apache.velocity.texen.util	100% (2/2)	62% (5/8)	68% (98/144)	69% (27/39)
org.apache.velocity.runtime.parser.node	91% (42/46)	65% (155/238)	70% (3412/4869)	72% (651.8/904)
org.apache.velocity.util.introspection	93% (13/14)	83% (52/63)	72% (1344/1873)	81% (308.6/381)
org.apache.velocity.texen.ant	100% (1/1)	61% (11/18)	72% (368/510)	73% (91.3/125)
org.apache.velocity.context	100% (4/4)	70% (39/56)	76% (368/484)	74% (102.7/138)
org.apache.velocity.exception	100% (4/4)	100% (8/8)	100% (44/44)	100% (18/18)

[all classes]

EMMA 2.0.4015 (stable) (C) Vladimir Roubtsov

Kuvio 5. Emman luoman HTML-raportin etusivu.

Pakettikohtainen raportti (Kuvio 6) erittelee kattavuusmitat luokkakohtaisesti. Paketin luokkakattavuudessa huomioidaan myös mahdolliset aliluokat, esimerkkiraportin paketissa on siis kahdeksan luokkaa, vaikka siinä on vain viisi java-tiedostoa.



EMMA Coverage Report (generated Tue May 18 22:13:27 CDT 2004)

[all classes]

COVERAGE SUMMARY FOR PACKAGE [org.apache.velocity.runtime]

name	class, %	method, %	block, %	line, %
org.apache.velocity.runtime	75% (6/8)	54% (82/151)	65% (1378/2120)	65% (352/539)

COVERAGE BREAKDOWN BY SOURCE FILE

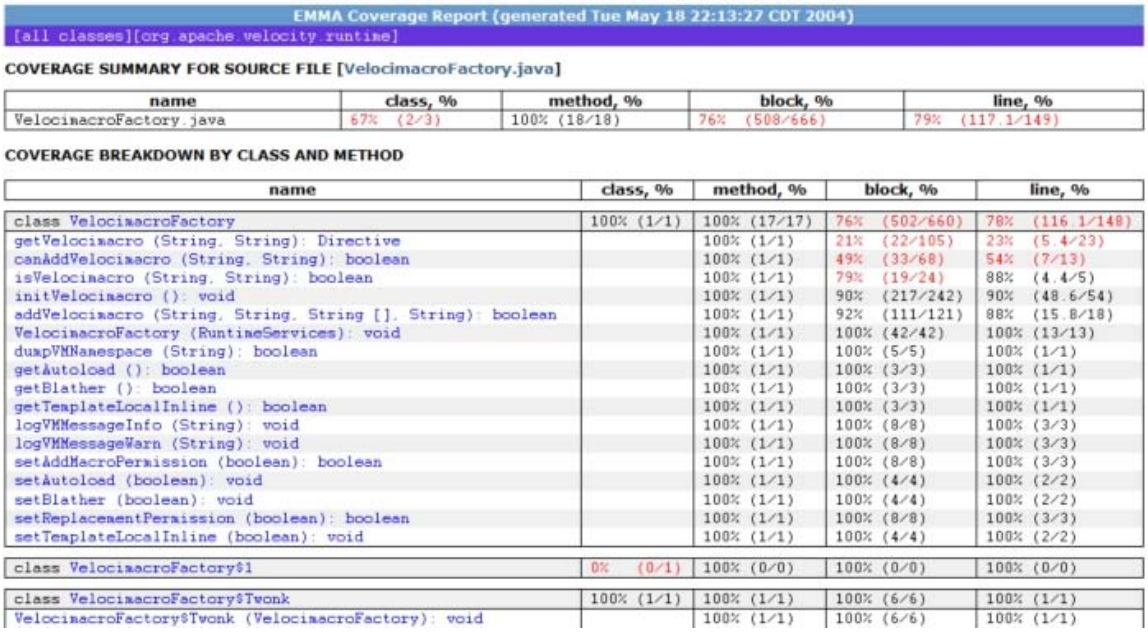
name	class, %	method, %	block, %	line, %
Runtime.java	0% (0/1)	0% (0/30)	0% (0/103)	0% (0/41)
RuntimeSingleton.java	100% (1/1)	32% (12/37)	32% (50/154)	33% (16/48)
RuntimeInstance.java	100% (1/1)	82% (37/45)	68% (539/790)	72% (139.9/193)
VelocimacroManager.java	100% (2/2)	71% (15/21)	69% (281/407)	73% (79/108)
VelocimacroFactory.java	67% (2/3)	100% (18/18)	76% (508/666)	79% (117.1/149)

[all classes]

EMMA 2.0.4015 (stable) (C) Vladimir Roubtsov

Kuvio 6. HTML-raportin pakettinäkymä

Kuten paketeistakin, jokaisesta java-tiedostosta on eritelty luokka-, metodi-, lohko- ja rivikattavuus. Kuviossa 7 on nähtävillä luokkakohtaisen raportin alkutiedot, jossa kattavuudet on vielä eritelty aliluokittain



EMMA Coverage Report (generated Tue May 18 22:13:27 CDT 2004)

[all classes][org.apache.velocity.runtime]

COVERAGE SUMMARY FOR SOURCE FILE [VelocimacroFactory.java]

name	class, %	method, %	block, %	line, %
VelocimacroFactory.java	67% (2/3)	100% (18/18)	76% (508/666)	79% (117.1/149)

COVERAGE BREAKDOWN BY CLASS AND METHOD

name	class, %	method, %	block, %	line, %
class VelocimacroFactory	100% (1/1)	100% (17/17)	76% (502/660)	78% (116.1/148)
getVelocimacro (String, String): Directive		100% (1/1)	21% (22/105)	23% (5.4/23)
canAddVelocimacro (String, String): boolean		100% (1/1)	49% (33/68)	54% (7/13)
isVelocimacro (String, String): boolean		100% (1/1)	79% (19/24)	88% (4.4/5)
initVelocimacro (): void		100% (1/1)	90% (217/242)	90% (48.6/54)
addVelocimacro (String, String, String [], String): boolean		100% (1/1)	92% (111/121)	88% (15.8/18)
VelocimacroFactory (RuntimeServices): void		100% (1/1)	100% (42/42)	100% (13/13)
dumpVMNamespace (String): boolean		100% (1/1)	100% (5/5)	100% (1/1)
getAutoload (): boolean		100% (1/1)	100% (3/3)	100% (1/1)
getBlather (): boolean		100% (1/1)	100% (3/3)	100% (1/1)
getTemplateLocalInline (): boolean		100% (1/1)	100% (3/3)	100% (1/1)
logVMMessageInfo (String): void		100% (1/1)	100% (8/8)	100% (3/3)
logVMMessageWarn (String): void		100% (1/1)	100% (8/8)	100% (3/3)
setAddMacroPermission (boolean): boolean		100% (1/1)	100% (8/8)	100% (3/3)
setAutoload (boolean): void		100% (1/1)	100% (4/4)	100% (2/2)
setBlather (boolean): void		100% (1/1)	100% (4/4)	100% (2/2)
setReplacementPermission (boolean): boolean		100% (1/1)	100% (8/8)	100% (3/3)
setTemplateLocalInline (boolean): void		100% (1/1)	100% (4/4)	100% (2/2)
class VelocimacroFactory$1	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
class VelocimacroFactory$Twonk	100% (1/1)	100% (1/1)	100% (6/6)	100% (1/1)
VelocimacroFactory$Twonk (VelocimacroFactory): void		100% (1/1)	100% (6/6)	100% (1/1)

Kuvio 7. Luokkakohtaisen raportin alkutiedot.

Emman raportteihin voidaan sisällyttää testattujen luokkien täydelliset lähdekoodit. Tämä on erittäin hyödyllistä testitapausten suunnittelemisen kannalta, testitapaukset on osattava kohdistaa tiettyyn osaan lähdekoodia. Kuvio 8 esittää lähdekoodiin tehtävän värikoodauksen, vihreällä korostetut rivit on ajettu ja punaisella värjätyt ovat jääneet suorittamatta. Kyseisessä esimerkissä kaksi ensimmäistä if-lausetta on ohitettu ja vasta kolmanteen on päästy sisälle.

```
private boolean canAddVelocimacro( String name, String sourceTemplate)
{
    /*
     *  short circuit and do it if autoloader is on, and the
     *  template is one of the library templates
     */

    if ( getAutoload() )
    {
        /*
         *  see if this is a library template
         */

        for( int i = 0; i < macroLibVec.size(); i++)
        {
            String lib = (String) macroLibVec.elementAt(i);

            if (lib.equals( sourceTemplate ) )
            {
                return true;
            }
        }
    }


    /*
     * maybe the rules should be in manager?  I dunno. It's to manage
     * the namespace issues first, are we allowed to add VMs at all?
     * This trumps all.
     */
    if (!addNewAllowed)
    {
        logVMMessageWarn("Velocimacro : VM addition rejected : " + name +
            " : inline VMs not allowed."  );

        return false;
    }

    /*
     *  are they local in scope?  Then it is ok to add.
     */
    if (!templateLocal)
    {
        /*
         * otherwise, if we have it already in global namespace, and they can't replace
         * since local templates are not allowed, the global namespace is implied.
         *  remember, we don't know anything about namespace managment here, so lets
         *  note do anything fancy like trying to give it the global namespace here
         *
         *  so if we have it, and we aren't allowed to replace, bail
         */
        if ( isVelocimacro( name, sourceTemplate ) && !replaceAllowed )
        {
            logVMMessageWarn("Velocimacro : VM addition rejected : "
                + name + " : inline not allowed to replace existing VM"  );
            return false;
        }
    }

    return true;
}
```

Kuvio 8. Luokkakohtaisen raportin värikoodaus.

Raportin muodostaminen

Raporttien muodostus komentorivillä muistuttaa offline instrumentointia ja se tapahtuu komennolla:

```
java emma report -r txt,html -sp src/ -in coverage.emma -in coverage.ec
```

Emma-kirjaston report-toiminallisuus tarvitsee peruskäytössä kolme parametria. Ensimmäisenä annettu –r asettaa halutut raporttityypit. Lähdekoodien polku asetetaan –sp parametrilla, näin Emma osaa näyttää kuviossa 8 esitellyn värikoodauksen. Instrumentoinnin tuottama metadata ja ajonaikainen seurantadata on annettava raportin luontia varten –in parametreilla. Oletuksena raportit tallennetaan nykyiseen kansioon luotavaan coverage-kansioon.

Ant-skriptillä vastaavan raportin luonti näyttää seuraavalta:

```
<emma enabled="${emma.enabled}" >
      <report sourcepath="${src.dir} >
        <infileset dir="${coverage.dir}" includes="*.em, *.ec" />
        <txt />
        <html />
      </report>
</emma>
```

Raportin luonnissa kerrotaan lähdekoodien sijainti sourcepath-parametrilla, johon annetaan muuttujan src.dir arvo. Raportissa huomioidaan kaikki coverage.dir –kansiossa olevat em- ja ec-päätteiset tiedostot. Lopuksi vielä asetetaan halutut raporttityypit <txt/> ja <html/> -elementeillä.

4.2 ABaTS ja Emma

Emma-moduulin asennus Ebsolut Oy:n ABaTS-järjestelmään eteni seuraavien vaiheiden kautta:

- tutustuminen olemassa olevaan ABATS –järjestelmään
- tutustuminen Emma-projektin dokumentaatioon http://emma.sourceforge.net –sivustolla
- Emma-moduulin asennus ja testaus ABATS –järjestelmässä
- ohjeistuksen laatiminen testiympäristön käyttäjille

ABaTS -järjestelmä on käytännössä tavallinen Windows-työasema, jossa ajetaan ajastetusti Ant-skriptejä. Skriptit ovat projektikohtaisia ja ne hakevat testattavan sovelluksen lähdekoodit versionhallinnasta. Sovelluksen kääntämiseen voidaan käyttää erillistä xml-tiedostoa, joka haetaan versionhallinnasta lähdekoodien mukana. Testattavan ohjelman lähdekoodien lisäksi versionhallinnasta haetaan JUnit-testitapausten lähdekoodit. Kääntämisen jälkeen testitapaukset ajetaan JUnit-testiajurin avulla. Mikäli jokin testi epäonnistuu tai palauttaa odottamattoman virheilmoituksen, järjestelmä lähettää sähköpostiviestin projektin vastuuhenkilölle. Lopuksi testien tuloksista muodostetaan html-muotoinen raportti ja se kopioidaan www-palvelimelle.

Emmaan tutustuminen alkoi komentorivityökalulla, joiden avulla oli yksinkertaisinta rakentaa ensimmäinen versio halutusta järjestelmästä. Aluksi komentorivin avulla suoritettiin lähdekoodin kääntäminen, jonka jälkeen oli instrumentoinnin vuoro. Instrumentoinnin yhteydessä oli hieman ongelmia, mutta ne ratkesivat fullcopy-parametrilla. Instrumentoinnin onnistuttua testien ajaminen sujui ongelmitta, samoin raportin koostaminen.

Komentorivityökalut olivat hyödyllisiä koko asennusprosessin ajan, sillä Ant-skiptistä on mahdollista suorittaa normaaleja komentokehotteen käskyjä. Ant-skriptien muokkaaminen etenikin siten, että aluksi uusi toiminto ajettiin komentorivin kautta. Kun uusi toiminto oli saatu toimimaan luotettavasti komentorivin kautta, se oli helppoa muuttaa Ant-komennoksi. Käytännössä komennot muutettiin Ant-muotoon viimeistään siinä vaiheessa, kun niille piti

antaa vaihtelevia parametreja. Parametrit oli yksinkertaisinta asettaa Ant-muuttujiin ja käyttää niitä.

Emman käyttöönottamiseksi skripteihin lisättiin käännettyjen koodien instrumentointi, virtuaalikoneen parametrit sekä raporttien muodostaminen. ABaTS luo testiajonsa lopuksi html-muotoisen pääsivun, jossa on linkit jokaisen projektin testiraporttiin. Pääsivu luodaan bat-tiedostolla. Bat-tiedosto sisältää normaaleja komentorivikomentoja koottuna yhteen tiedostoon. Pääsivun generointia muokattiin lisäämällä linkit Emman luomiin raportteihin.

5 POHDINTA

Opinnäytetyön teko sujui ilman suurempia ongelmia. Opinnäytetyösuunnitelmassa laaditut aikataulut eivät loppujen lopuksi pitäneet, se johtui lähinnä tekijästä itsestään. Ohjelmistotestauksesta oli monia kattavia lähteitä, mutta Emma-kirjastoa käsitteleviä lähteitä oli hieman vaikeampi löytää. Emma-projektin kotisivut ja keskustelupalsta tarjosivat kuitenkin käytännönläheisiä esimerkkejä.

ABaTS-järjestelmä oli entuudestaan melko tuntematon, mutta työn edetessä se tuli hyvinkin tutuksi. Suurimmat ongelmat liittyivät Ant-skripteihin ja niiden käyttämään syntaksiin, joka oli aluksi hieman vaikeaselkoinen. Ongelmatilanteissa saatoin kääntyä yrityksen työntekijöiden puoleen, heiltä saatujen vinkkien avulla oli helppo päästä asiassa eteenpäin.

Opinnäytetyön ensisijainen tavoite oli kehittää Ebsolut Oy:n automaattista testausympäristöä. Tässä onnistuttiin ja ABaTS-järjestelmän tehokkuutta saatiin lisätyksi. Nyt JUnit-testejä ei tarvitse tehdä sokkona, vaan testaaja voi tarkistaa, mihin osaan lähdekoodia testitapaukset vaikuttavat ja suunnitella seuraavat testit tehokkaammin.

Opinnäytteen tuotosta on jo jatkokehitetty ulottamalla testien kattavuusmittaus ABaTS-järjestelmän ulkopuolelle. Emma on käytössä koneellisesti ajettavissa kestotesteissä sekä käsin tehtävissä käytettävyystesteissä. Käytännössä tämä tapahtuu instrumentoimalla lähdekoodeista käännetty jar-tiedosto. Sovellus käynnistetään instrumentoidusta jar-tiedostosta ja varsinaiset testit suoritetaan normaalisti. Testien päätteeksi voidaan koostaa raportti ja siirtää se www-palvelimelle. Tällaista ABaTS-järjestelmän ulkopuolista käyttöä varten on laadittu Ant-skripti, joka ajetaan komentokehotteesta. Skriptille annetaan parametrina haluttu toiminto: instrumentointi, raportin koostaminen tai raportin julkaiseminen www-palvelimelle.

Yrityksen työntekijöille on tarkoitus laatia ohjeistus Emma-kirjaston hyödyntämisestä ohjelmistokehityksessä. Ohjeistuksessa esitellään Emman toimintoja ja käyttöä. Emman käyttö manuaalisessa testauksessa vaatii muutamia huomioitavia asioita, jotka tulee saattaa ohjelmoijien tietoon. Kuten instrumentoinnin yhteydessä on kerrottu käännettäessä lähdekoodeja on muistettava lisätä debug-informaatio käännettäviin luokkiin. Lisäksi testaajalla pitää olla tiedossa versionhallinnan haara, jossa testattavaa sovellusta vastaavat lähdekodit sijaitsevat. Lähdekoodit tarvitaan testiraportin luomiseen.

Testauksen kattavuus oli aiheena kaiken kaikkiaan mielenkiintoinen. Työtä tehdessä Ant-skriptaus tuli tutuksi käytännössä ja siitä on varmasti hyötyä jatkossakin. Työelämässä tulee varmasti monia tilanteita, joissa on itsenäisesti opiskeltava uusia asioita. Opinnäytetyön tekeminen olikin hyvää harjoitusta juuri itsenäisestä tiedon hausta, ja sen soveltamisesta käytäntöön.

Testauksen kattavuus tulisi ottaa huomioon jo ohjelmistokehityksen alusta asti. Testaus on yksi osa ohjelmoijan ammattitaitoa. Testitapausten kirjoittaminen on helpointa ohjelmoinnin ohessa. Koulu- ja työelämässä ohjelmoijat tuntuvat suhtautuvan testaukseen vastahakoisesti ja omien tuotosten testaus on lähinnä summittaista kokeilua. Tämä on enimmäkseen asennekysymys ja oman ammatillisen kehityksen kannalta testaukseen tulee kiinnittää huomiota.

LÄHTEET

Cornett, S. 2007. Code Coverage Analysis. Tallennettu 24.9.2007. Saatavilla www-muodossa osoitteessa http://www.bullseye.com/coverage.html

Haikala, I. Märijärvi, J. 2002. Ohjelmistotuotanto. Pieksämäki, Satku.fi, 2002

Huhtamäki, M. 2001. Oliopohjainen testaus. Tutkielma. Helsingin yliopisto. Tallennettu 27.5.2005. Saatavilla www-muodossa osoitteessa www.cs.helsinki.fi/u/mphuhtam/download/olio.pdf

Kautto, T. 1996. Ohjelmistotestaus ja siinä käytettävät työkalut. Tulostettu 18.5.2007. Saatavilla www-muodossa osoitteessa http://www.mit.jyu.fi/opiskelu/seminaarit/ohjelmistotekniikka/testaus/

Oulun seudun ammattikorkeakoulu. 2006. Testausstrategiat. Tulostettu 27.5.2007. Saatavilla www-muodossa osoitteessa http://www.oamk.fi/sbc/testaus/testausstrategiat.htm

Patton, R. 2005. Software Testing - Second Edition. Indianapolis, SAMS, 2005

Roubtsov, V. 2006. Emma – a free code coverage tool. Tallennettu 31.10.2007. Saatavilla www-muodossa osoitteessa http://emma.sourceforge.net/index.html

Seppänen, K. 2005. Ohjelmistotestaus-opintojakson materiaalit. Tallennettu 24.5.2007. Saatavilla www-muodossa osoitteessa http://data.hamk.fi/~lseppane/courses/ohjelmistotestaus/Videot_ohjelmistotestaus.htm

LIITTEIDEN LUETTELO

EMMAN TEKSTIRAPORTTI

```
[EMMA v2.0.4015 (stable) report, generated Sat May 15 12:02:28 CDT 2004]
-------------------------------------------------------------------------------
OVERALL COVERAGE SUMMARY:

[class, %]     [method, %]    [block, %]     [line, %]      [name]
85%  (157/185)!65%  (1345/2061)!      60%  (44997/74846)!    64%  (8346.3/13135)!   all classes

OVERALL STATS SUMMARY:

total packages:8
total classes: 185
total methods: 2061
total executable files:       62
total executable lines:       13135

COVERAGE BREAKDOWN BY PACKAGE:

[class, %]     [method, %]    [block, %]     [line, %]      [name]
25%  (1/4)!    25%  (3/12)!   40%  (3012/7446)!      25%  (3/12)!   com.sun.tools.javac.v8.resources
94%  (16/17)!  49%  (41/83)!  48%  (1111/2292)!      45%  (201.1/450)!      com.sun.tools.javac.v8
88%  (45/51)!  61%  (242/397)!54%  (3070/5729)!      52%  (809.6/1563)!     com.sun.tools.javac.v8.tree
83%  (19/23)!  60%  (134/224)!54%  (2746/5063)!      56%  (580.1/1041)!     com.sun.tools.javac.v8.util
100% (1/1)     40%  (2/5)!    58%  (25/43)!  49%  (5.9/12)! com.sun.tools.javac
77%  (33/43)!  59%  (310/529)!60%  (10584/17674)!    61%  (2077.2/3396)!    com.sun.tools.javac.v8.code
91%  (39/43)!  75%  (521/698) 66%  (19701/29863)!    70%  (3606.9/5138)!    com.sun.tools.javac.v8.comp
100% (3/3)     81%  (92/113)  70%  (4748/6736)!      70%  (1062.4/1523)!    com.sun.tools.javac.v8.parser
-------------------------------------------------------------------------------
```