WWW.JAMK.FI

OHJELMISTOTESTAUS

Opas ohjelmistokehittäjille

Antti Kolehmainen

Opinnäytetyö
Marraskuu 2009

Tietojenkäsittely
Luonnontieteiden ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN KUVAILULEHTI

Tekijä(t) KOLEHMAINEN, Antti	Julkaisun laji Opinnäytetyö	Päivämäärä 13.11.2009
	Sivumäärä 46	Julkaisun kieli suomi
	Luottamuksellisuus ( )	Verkkojulkaisulupa myönnetty (X)
Työn nimi OHJELMISTOTESTAUS – OPAS OHJELMISTOKEHITTÄJILLE		
Koulutusohjelma Tietojenkäsittely		
Työn ohjaaja(t) KIVIAHO, Niko		
Toimeksiantaja(t) RAEHALME, Thomas, Abakus Ohjelmistot Oy		

Tiivistelmä

Työn tarkoituksena oli toteuttaa Abakus Ohjelmistot Oy:lle kirjallinen ohjelmistotestauksen ohjeistus Java EE -teknologioilla. Lisäksi työn tavoitteena oli osoittaa raportin lukijalle testauksen tarpeellisuus teoreettisesta näkökulmasta sekä esitellä toimiviksi havaitut testausstrategiat ja -tekniikat.

Raporttiin sisällytettiin teoriaosuus ohjelmistoprosessista ja ohjelmistotestauksesta. Teoriaosuus on yleistettävissä myös muiden ohjelmointiteknologioiden saralla, eikä sitä ole sidottu mihinkään varsinaiseen prosessimalliin. Testauksen työkalujen opas on toteutettu Java EE -teknologioilla, ja se vaatii kyseisen teknologian pohjatuntemusta.

Työn tuloksena saatiin aikaan raportti, joka soveltuu testausstrategioiden ja -tekniikoiden opiskeluun. Lisäksi raportti sisältää ohjeet sekä valittujen testaustyökalujen käyttöön että käyttöönottoon. Raporttia voidaan käyttää oppaana sekä työn ohessa että perehdyttämisessä.

Avainsanat (asiasanat)

Ohjelmistotestaus, Java EE, JUnit, Selenium, Cobertura, JMeter, Eclipse

Muut tiedot

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s)	Type of publication	Date
KOLEHMAINEN, Antti	Bachelor's Thesis	13112009
	Pages 46	Language Finnish
	Confidential ( ) Until	Permission for web publication (X)

Title
SOFTWARE TESTING – GUIDE FOR SOFTWARE DEVELOPERS

Degree Programme
Business Information Systems

Tutor(s)
KIVIAHO, Niko

Assigned by
RAEHALME, Thomas, Abakus Ohjelmistot Oy

Abstract

The goal of this assignment was to produce a written guide of software testing for Abakus Ohjelmistot Oy using Java EE technologies. In addition, the goal was to show the reader the importance of software testing from a theoretical standpoint and introduce a proven set of testing strategies and methods.

Theory of software process and software testing was included in the report. The report's theory section is generalizable for other software technologies as well – it is not tied to any process model. The guide is dependent on Java EE technologies and requires some experience with this technology.

The result of this assignment was a report that can be used when studying testing strategies and methods. In addition, the report includes instructions for deployment and usage of selected tools. The report can be used as a reference guide alongside work and when briefing workers.

Keywords
Software testing, Java EE, JUnit, Selenium, Cobertura, JMeter, Eclipse

Miscellaneous

SISÄLTÖ

KUVIOT

1 JOHDANTO

1.1 Opinnäytetyön toimeksiantaja ja tarkoitus

Opinnäytetyön toimeksiantajana toimi Abakus Ohjelmistot Oy. Yritys on suomalainen mobiiliajan ohjelmistotalo, joka on erikoistunut palvelutoimialojen asiakkuudenhallinta- ja toiminnanohjausratkaisuihin Java EE- ja Java ME -teknologioita käyttäen.

Toimeksiantajayrityksen vähäisestä henkilöstömäärästä johtuen kehittäjät joutuvat usein työskentelemään projekteissa yhden tai kahden henkilön voimin. Osittain tämän vuoksi kehityksessä käytetty prosessi muodostuu kehittäjän henkilökohtaisista työtavoista, mikä aiheuttaa monenlaisia ongelmia. Useimmiten tämä johtaa siihen, että tuotantoprosessissa toteutuksen osuus on maksimoitu ja testauksen osuus minimoitu. Tämä aiheuttaa pahimmassa tapauksessa ohjelmistoihin jääneiden virheiden siirtymisen suoraan tuotantokäyttöön.

Tämän opinnäytetyön tarkoituksena oli tuottaa toimeksiantajayrityksen ohjelmistokehittäjille hyödyllisten testaustyökalujen käyttöopas Java EE -web-sovellusten testaukseen. Lisäksi työn tehtävänä oli osoittaa ohjelmistokehittäjälle testauksen tarpeellisuus teoreettisesta näkökulmasta ja esitellä toimiviksi havaitut testaustekniikat. Raportti sisältää teoriaosuuden sekä ohjelmistoprosessista että testauksesta. Lopullisena yhteenvetona toimii ohjeistus ohjelmistojen testaukseen Java EE -teknologioilla. Tutkimus toteutettiin kehitysprojektina.

1.2 Lähtökohdat

National Institute of Standards and Technologyn vuonna 2002 tekemän tutkimuksen mukaan ohjelmistovirheet aiheuttavat vuosittain pelkästään Yhdysvalloissa 59,5 miljardin dollarin kustannukset. Tutkimuksessa todetaan myös, että vaikka kaikkia vikoja ei koskaan voida korjata, yli kolmannes niistä aiheutuneista kustannuksista voitaisiin välttää parantamalla testausinfrastruktuuria. (Software Errors Cost U.S. Economy $59.5 Billion Annually 2002.)

Ohjelmistot ovat kasvavasta monimutkaisuudestaan johtuen yhä alttiimpia virheille. Ohjelmistotuotteiden kokoa ei enää nykypäivänä mitata tuhansilla vaan pikemminkin miljoonilla riveillä ohjelmakoodia. Ohjelmistokehittäjät käyttävät arviolta 80 prosenttia kehityskustannuksista virheiden löytämiseen ja korjaamiseen. Tästä huolimatta tuskin mikään muu teollisuudenala toimittaa yhtä virheellisiä tuotteita loppukäyttäjille.

Testaus tulisi kokea tärkeäksi ohjelmistoprosessin osa-alueeksi, sillä ohjelmistot elävät jatkuvassa kasvupaineessa. Ohjelmistotuotteet kasvavat ja monimutkaistuvat, mutta organisaatioilla ei kuitenkaan ole varaa sijoittaa kaikkia resurssejaan huonon testausprosessin aiheuttamiin kustannuksiin. Ohjelmistotestaus ei ole pelkkää ohjelmiston toimivuuden silmämääräistä toteamista vaan ohjelmistoprosessin tärkeä ja laaja osa-alue, jonka toimivuus tulisi ottaa huomioon prosessin jokaisessa vaiheessa.

2 OHJELMISTOPROSESSI

Ohjelmistoprosessi on joukko työtoimintoja, jotka johtavat ohjelmiston tuottamiseen. Tämä voi tarkoittaa ohjelmiston tuottamisen standardilla ohjelmointikielellä tyhjästä tai olemassa olevia järjestelmiä ja komponentteja käyttäen. (Sommerville 2007, 64.)

Minkä tahansa tuotteen tai järjestelmän rakentamisessa on tärkeää kulkea ennalta määriteltyä polkua, joka on apuna laadukkaan tuloksen saavuttamiseksi. Etenemissuunnitelmaa kutsutaan ohjelmistoprosessiksi, jonka yksi tärkeä osa-alue testaus on. Vuosikymmenten saatossa on kehitetty tiettyjä hyväksi havaittuja ohjelmistoprosessin vaiheita, joiden tuntemus ja soveltaminen on myös onnistuneen testauksen edellytyksenä. (Pressman 2005, 52.)

2.1 Prosessi

Sommervillen (2007) mukaan ohjelmistoprosessin tulee jossain muodossa sisältää ainakin seuraavat peruselementit:

- ohjelmiston määrittely
- ohjelmiston suunnittelu ja toteutus
- ohjelmiston validointi (testaus)
- ohjelmiston kehittyminen (ylläpito)

Suurin osa nykypäivänä käytössä olevista ohjelmistoprosesseista perustuu seuraavissa luvuissa esiteltyihin vaiheisiin. Tämä koskee sekä ketteriä että perinteisiä ohjelmistoprosesseja.

2.1.1 Määrittelyvaihe

Ohjelmiston määrittely tarkoittaa vaihetta, jossa pyritään hahmottamaan ja dokumentoimaan, mitä palveluita ja toimintoja tuotettavalta järjestelmältä odotetaan. Vaihe, jota usein vaatimusmäärittelyksikin kutsutaan, on erityisen kriittinen ohjelmistoprosessin vaihe, koska siinä tehdyt virheet johtavat yleensä ongelmiin myöhemmin ohjelmiston suunnittelussa ja toteutuksessa. Vaatimusmäärittelyn tulos on dokumentti, joka toimii ohjelmiston spesifikaationa. Määrit-

tely on oleellinen vaihe myös testauksen kannalta, sillä yhtenä testauksen tärkeimmistä tehtävistä on todeta, että tuotettava ohjelmisto vastaa sille asetettuja vaatimuksia. (Sommerville 2007, 75.)

Vaatimusmäärittelyn tuloksena syntyvää dokumenttia kutsutaan *toiminnalliseksi määrittelyksi*. Dokumentissa kuvataan ohjelmiston toiminnalliset ja ei-toiminnalliset vaatimukset sekä rajoitukset. Toiminnalliset vaatimukset sisältävät ohjelmistolla toteutettavat ominaisuudet, käyttöliittymän ja mahdollisen integroinnin muihin järjestelmiin. Ei-toiminnallisia vaatimuksia ovat muun muassa suoritusteho, vasteaika ja käytettävyys. Rajoituksia ovat esimerkiksi palvelimen tekniset rajoitukset sekä käytettävä ohjelmointikieli. (Haikala & Märijärvi 2004, 38 - 39.)

2.1.1.1 Toiminnalliset vaatimukset

Toiminnalliset vaatimukset määrittelevät sen, mitä järjestelmän tulisi tehdä. Vaatimusten rakenne riippuu tuotettavan ohjelmiston tyypistä, tulevista käyttäjistä sekä organisaation lähestymistavasta vaatimusten kirjaamiseen. Vaatimukset voivat olla muun muassa seuraavanlaisia:

- Käyttäjän tulee pystyä etsimään kirjoja sähköisen kirjakaupan koko tietokannasta.
- Järjestelmän tulee tarjota sopivat katselutoiminnot, joilla käyttäjät voivat lukea kohteena olevia dokumentteja.

Epätarkkuus toiminnallisissa vaatimuksissa johtaa usein ongelmiin ohjelmiston kehityksessä. Kehittäjälle on luontaista tulkita epäselviä vaatimuksia yksinkertaistetusti helpottaakseen toteutusta, mikä ei useinkaan vastaa asiakkaan odotuksia. Tämä taas johtaa uusiin vaatimuksiin ja vaatimusmuutoksiin, jotka hidastavat ohjelmiston valmistumista ja testausta sekä lisäävät tuotantokustannuksia. (Sommerville 2007, 120 - 121.)

Toiminnallisten vaatimusten tulisi olla sekä *perinpohjaisia* että *johdonmukaisia*. Perinpohjaisuus tarkoittaa sitä, että kaikki käyttäjän vaatimat palvelut tulisi määritellä. Johdonmukaisuus tarkoittaa sitä, että järjestelmälle ei tulisi määritellä ristiriidassa olevia vaatimuksia. Mitä suurempi järjestelmä, sen vaikeampi tätä periaatetta on toteuttaa täydellisesti. (Sommerville 2007, 121.)

2.1.1.2 Ei-toiminnalliset vaatimukset

Ei-toiminnalliset vaatimukset ovat vaatimuksia, jotka eivät suoraan koske ohjelmiston toiminnallisuutta. Ne voivat liittyä ominaisuuksiin, kuten luotettavuus, turvallisuus, palvelun saatavuus ja vasteaika. Toisaalta ne voivat myös määritellä järjestelmälle rajoituksia, kuten palvelimen kapasiteetti ja suorituskyky. (Sommerville 2007, 122 - 123.)

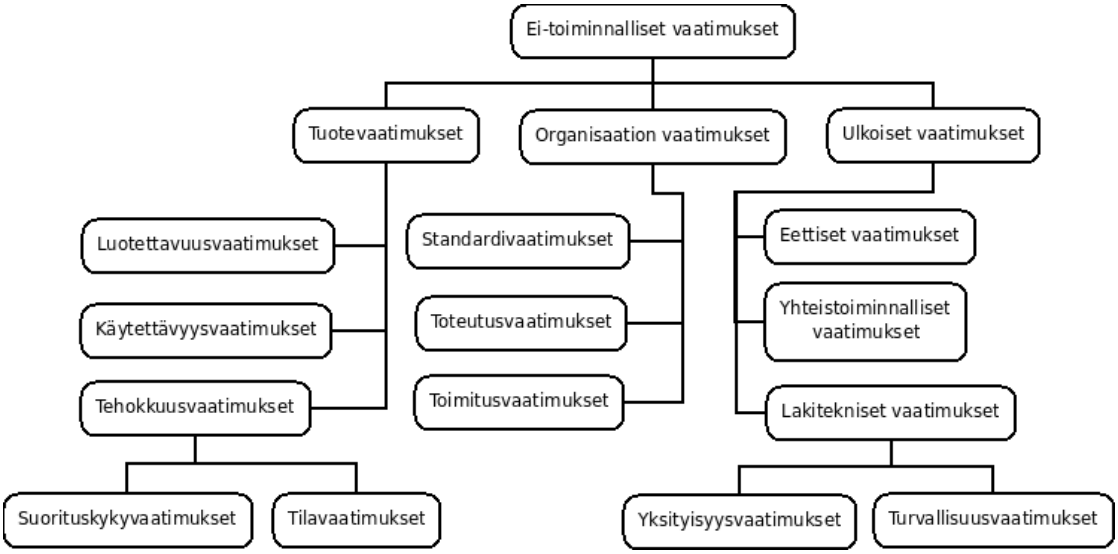
Ei-toiminnalliset vaatimukset ovat usein kriittisempiä kuin toiminnalliset vaatimukset. Käyttäjät löytävät yleensä tapoja kiertää ohjelmiston yksittäisten toimintojen omituisuuksia, mutta ei-toiminnallisen vaatimuksen pettäessä koko ohjelmiston toiminta voi olla vaakalaudalla. Tyypillisiä ei-toiminnallisia vaatimuksia ovat seuraavat vaatimukset:

- *Tuotevaatimukset*, jotka määrittelevät ohjelmiston käyttäytymistä, luotettavuutta, toiminnallisuuden siirrettävyyttä ja käyttäjävaatimuksia.
- *Organisaatiotason vaatimukset*, jotka johdetaan asiakas- ja kehittäjäorganisaatioiden toimintaperiaatteista ja menettelytavoista. Niitä voivat olla käytettävä prosessi, ohjelmointikieli ja ohjelmiston toimitukseen liittyvät vaatimukset.
- *Ulkoiset vaatimukset*, jotka sisältävät kaikki ne vaatimukset, jotka johdetaan järjestelmän ulkoisista tekijöistä ja kehitysprosessista. Ne voivat sisältää tekijöitä, jotka määrittelevät, miten ohjelmisto integroituu organisaation muihin järjestelmiin, laillisia vaatimuksia, jotka varmistavat, että

ohjelmisto on laillinen, sekä eettisiä vaatimuksia.
(Sommerville 2007, 122 - 123.)

Ei-toiminnallisten vaatimusten yleinen ongelma on se, että ne ovat usein hankalasti todennettavissa. Käyttäjät tai asiakkaat asettavat ne usein vaikeasti määriteltäviksi tavoitteiksi, kuten helppokäyttöisyys, nopea käyttäjäpalaute ja kyky selviytyä järjestelmävirheestä. Tällaiset vaatimukset ovat haastavia kehittäjille, koska ne jättävät tilaa tulkinnoille ja erimielisyyksille. Tämän vuoksi ei-toiminnallisten vaatimusten pitäisi olla kvantitatiivisia, jotta ne pystytään mittaamaan objektiivisesti. (Sommerville 2007, 124.)

Kaikkien vaatimusten tekeminen mitattaviksi on käytännössä asiakkaalle mahdotonta. Joillekin tavoitteille, kuten ylläpidettävyydelle, ei ole mittareita. Toisaalta vaikka vaatimus olisikin mitattavissa, asiakkaat eivät välttämättä osaa hahmottaa tarpeitaan vaaditulla tavalla. Lisäksi ei-toiminnallisten vaatimusten mittaaminen voi olla toisinaan kallista, eivätkä asiakkaat ole välttämättä valmiita sijoittamaan siihen suuria summia. Tämän vuoksi vaatimusmäärittely usein sisältää tarkkojen ja mitattavissa olevien vaatimusten lisäksi yleisiä tavoitteita, jotka auttavat kehittäjiä hahmottamaan asiakkaan asettamia prioriteetteja. Seuraavassa kuviossa (kuvio 1) esitellään esimerkkejä ei-toiminnallisten vaatimusten tyypeistä. (Sommerville 2007, 124.)



KUVIO 1. Ei-toiminnallisten vaatimusten tyyppejä.

2.1.2 Suunnitteluvaihe

Suunnitteluvaiheessa suunnitellaan määrittelyn kuvaamien toimintojen toteutus. Usein mielletään, että määrittelyvaiheessa kuvataan ”mitä” ohjelmisto tekee ja suunnitteluvaiheessa kuvataan ”miten” se suorittaa tehtävänsä. Yleensä suunnitteluvaihe on jaettu kahteen tai useampaan vaiheeseen. (Haikala & Märijärvi 2004, 40.)

2.1.2.1 Arkkitehtuurisuunnittelu

Laajat järjestelmät koostuvat aina pienemmistä komponenteista, jotka tarjoavat tiettyjä palveluita. Arkkitehtuurisuunnittelu on kriittinen vaihe suunnittelun ja vaatimusmäärittelyn välillä. Se on vaihe, jossa pyritään saamaan aikaan järjestelmä, joka vastaa toiminnallisia ja ei-toiminnallisia vaatimuksia. Arkkitehtuurisuunnittelun aikana pyritään vastaamaan muun muassa seuraaviin kysymyksiin:

- Onko olemassa yleistä ohjelmistoarkkitehtuuria, jota voitaisiin käyttää pohjana tuotettavalle järjestelmälle?

- Millainen arkkitehtuuri sopii tuotettavalle järjestelmälle?
- Millainen lähestymistapa otetaan järjestelmän rakentamiseen?
- Kuinka järjestelmän osat jaetaan komponentteihin?
- Miten arkkitehtuurisuunnitelma arvioidaan?
- Kuinka järjestelmän arkkitehtuuri dokumentoidaan?

(Sommerville 2007, 243 - 245.)

2.1.2.2 Komponenttisuunnittelu

Komponenttisuunnittelun tarkoituksena on arkkitehtuurisuunnitelman perusteella suunnitella sekä järjestelmäkohtaisten että uudelleenkäytettävien komponenttien käyttö ja toteutus. Komponentin uudelleenkäyttö tarkoittaa sitä, että samaa komponenttia voidaan käyttää joko uudelleen saman järjestelmän sisällä tai kokonaan toisessa järjestelmässä. (OPEN Process Framework, 2005)

Usein tavoitteena on suunnitella komponentit uudelleenkäytettävyyden näkökulmasta, koska uudelleenkäyttö vähentää kehittäjien työmäärää huomattavasti. Komponenttisuunnittelun tuloksena saadaan arkkitehtuurisuunnitelma, jota on tarkennettu kunkin komponentin osalta. Komponenttisuunnittelussa voidaan myös suunnitella testit kullekin komponentille. (Mt.)

2.1.2.3 Käyttöliittymäsuunnittelu

Käyttöliittymäsuunnittelun tarkoituksena on suunnitella mahdollisimman tehokas kommunikointiväylä tietokoneen ja ihmisen välille. Jos ohjelmisto on vaikeakäyttöinen, se voi aiheuttaa käyttäjälle virheitä ja turhautumista riippumatta siitä, onko ohjelmisto itsessään tehokas tai monipuolinen. (Pressman 2005, 356.)

Käyttöliittymäsuunnittelussa tunnistetaan käyttäjät, tehtävät ja ympäristön vaatimukset, joista luodaan käyttötapaukset. Käyttötapaus analysoidaan ja ana-

lyysin perusteella määritellään joukko toimintoja, joilla tehtävä suoritetaan. Toimintojen perusteella luodaan käyttöliittymän ulkoasu, joka sisältää toimintojen suorittamiseen vaadittavat käyttöliittymäkomponentit. (Pressman 2005, 356.)

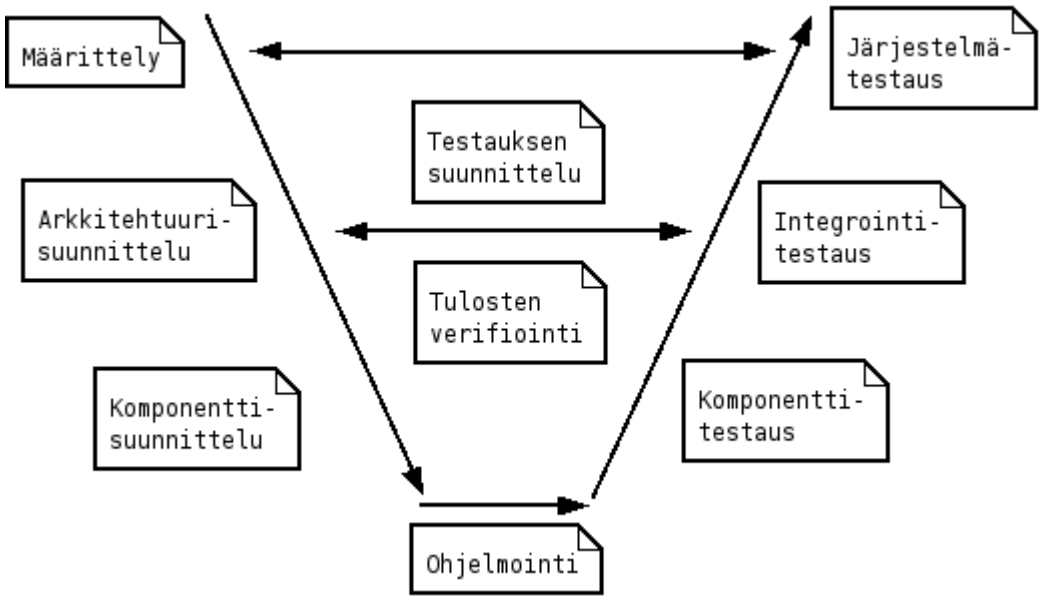
2.1.3 Ohjelmointivaihe

Ohjelmointivaihe sisältää suunnitelmien perusteella suoritettavan ohjelmakoodin kirjoituksen. Nykyään on myös tyypillistä kehittää ohjelmistoja Test Driven Development -periaatteen mukaisesti. Tämä tarkoittaa sitä, että osa testausvaihetta sisällytetään heti alusta asti osaksi ohjelmointivaihetta, kehittämällä ohjelmakoodia testitapausten avulla.

Ohjelmakoodi kirjoitetaan yleensä organisaation valitsemilla ohjelmointikielillä, kehyksillä ja työkaluilla. Nykyään on kuitenkin yleistymässä käytäntö, jossa kehittäjät voivat valita omat kehitystyökalunsa. Henkilökohtainen työkalujen valinta vaikuttaa positiivisesti motivaatioon, sekä lisää joissain tapauksissa jopa työtehoa.

2.1.4 Testausvaihe

Testausvaiheessa pyritään löytämään ohjelmistosta ennalta havaitsemattomia virheitä. Testaus tapahtuu yleensä usealla tasolla niin sanotun V-mallin mukaisesti. Seuraavassa kuviossa (kuvio 2) esitelty V-malli jakaa testauksen komponenttitestaukseen, integraatiotestaukseen ja järjestelmätestaukseen. Komponenttitestauksessa etsitään vikoja yksittäisistä komponenteista, integraatiotestauksessa komponenttien yhteistoiminnasta ja järjestelmätestauksessa koko järjestelmän toiminnoista ja suorituskyvystä. Tämän oppaan luku 3 keskittyy kokonaan ohjelmistotestaukseen. (Haikala & Märijärvi 2004, 40.)



KUVIO 2. Testauksen V-malli (Haikala & Märijärvi 2004, 289, muokattu)

2.1.5 Ylläpitovaihe

Ylläpitovaihe on pääosin virheiden korjausta, ohjelmiston muuttamista uusien tai muuttuneiden vaatimusten perusteella sekä asiakkaan ongelmien ratkomista. Yleensä ylläpitovaihe suoritetaan kaikkien muiden vaiheiden jälkeen.

Ylläpito voidaan karkeasti jakaa korjaavaan, mukautuvaan ja täydentävään ylläpitoon. Korjaavassa ylläpidossa korjataan ohjelmiston virheitä, mukautuvassa ylläpidossa muutetaan ohjelmistoa muuttuneiden vaatimusten perusteella ja täydentävässä ylläpidossa parannellaan ohjelmistoa lisäämällä sen toiminnallisuutta. (Haikala & Märijärvi 2004, 41.)

2.2 Prosessin arviointi

Ohjelmistoprosessin olemassaolo ei ole tae sille, että ohjelmisto valmistuu ajallaan tai edes vastaa asiakkaan tarpeita. Tämän saavuttamiseksi prosessissa tulee käyttää luotettavia ohjelmistotuotannon työtapoja. Tämän lisäksi prosessia itsessään tulisi arvioida, millä varmistetaan, että se vastaa sellaisia kriteereitä, joiden on huomattu olevan edellytyksenä onnistuneelle ohjelmistotuotannolle. Jos prosessi on kehno, työn lopputulos epäilemättä kärsii. Pakonomainen prosessin noudattaminen on toisaalta myös vaarallista. (Pressman 2005, 66, 72.)

Vaikka täydellistä prosessia ei ohjelmistotuotannossa varsinaisesti ole, monessa organisaatiossa käytetyissä prosesseissa on parantamisen varaa. Käytössä olevat prosessit voivat sisältää vanhoja tekniikoita, tai ne eivät välttämättä käytä hyödyksi ohjelmistotuotannon parhaita toimintatapoja. (Sommerville 2007, 64 – 65.)

3 OHJELMISTOTESTAUS

Nykypäivän liiketoimintaympäristössä tietokoneet eivät enää ole pelkästään työkaluja, joilla organisaatiot saavuttavat lisää tehokkuutta ja tuottavuutta. Tietokoneiden käyttö on saavuttanut sen pisteen, etteivät yritykset, niin pienet kuin suuretkaan, *voi selvitä ilman tietotekniikkaa*. Tämän vuoksi ohjelmistovirheet ovat yhä pelätympi uhka liiketoiminnalle. (Loveland, Miller, Prewitt & Shannon 2005, 1.)

Ohjelmistojen testaus on suunnitelmallista virheiden etsimistä ohjelmaa tai sen osaa suorittamalla. Nykyään testauksen määritelmä käsittää myös kaikki ne menetelmät, joilla pyritään mittaamaan ja parantamaan ohjelman laatua. Tällä määritelmällä korostetaan testauksen tuottamien mittareiden tärkeyttä tuoteke-

hityksessä. (Haikala & Märijärvi 2004, 284 – 285.)

3.1 Ohjelmistovirheet

Virhe (error, mistake, bug) on poikkeus tuotettavan ohjelmiston spesifikaatiosta. Käytännössä tämä tarkoittaa sitä, että ohjelma käyttäytyy epätoivotulla tavalla, vaatimusten vastaisesti. Ohjelmavirheiden vakavuus voi vaihdella pienistä, käyttäjää ärsyttävistä yksityiskohdista jopa järjestelmän käytön kokonaan estäviin virheisiin. (Haikala & Märijärvi 2004, 287.)

Arvioidaan, että ohjelmoinnin jälkeen ohjelma sisältää yleensä noin yhden virheen muutamaa kymmentä koodiriviä kohden. Pitkään käytössä olleissa ohjelmissa arvioidaan olevan yksi virhe tuhatta koodiriviä kohden. Lisäksi on arvioitu, että noin 5 % virheistä jää kokonaan havaitsematta. Tästä voidaan päätellä, että ohjelmistotestaus on sekä erittäin tärkeä että vaikea ohjelmistoprosessin vaihe. (Haikala & Märijärvi 2004, 287 - 288.)

3.2 Testauksen perusperiaatteet

Ohjelmistotestauksella on kaksi selkeää perusperiaatetta:

1. Osoittaa kehittäjille ja asiakkaille, että tuotettu ohjelmisto täyttää sille asetetut vaatimukset.
2. Löytää ohjelmistosta viat ja ongelmat, jotka aiheuttavat sille epätoivottua, virheellistä tai ylipäätään määrittelystä poikkeavaa käytöstä.

Pressman (Pressman 2005, 146.) määrittelee testauksen seuraavanlaisesti:

- Testauksen pääasiallinen tarkoitus on löytää ohjelmavirheitä.
- Hyvä testi on sellainen, jolla on suuri todennäköisyys löytää virheitä.

- Onnistuneeksi testiksi voidaan sanoa testiä, jolla testaaja löytää aiemmin löytämättömän virheen.

Testauksen hyöty tulee parhaiten esille silloin, kun pystytään suunnittelemaan ja toteuttamaan testejä, joilla järjestelmällisesti onnistutaan löytämään virheitä mahdollisimman pienellä ponnistelulla.

Pressman (2005) esittelee ohjelmistotestaukselle viisi toimintaperiaatetta:

Tulisi pitää huoli siitä, että *kaikki testit voidaan jäljittää asiakkaan vaatimusmäärittelyyn*. Suurimmat ohjelmavirheet asiakkaan kannalta ovat ne, jotka estävät ohjelmaa toteuttamasta vaatimuksiaan. (Mts. 147.)

Testit tulisi suunnitella kauan ennen kuin testaus alkaa. Testaussuunnitelmat voidaan tehdä yhdessä toteutussuunnitelmien kanssa. Kaikki testit voidaan siis suunnitella ennen kuin yhtään ohjelmakoodia on kirjoitettu. (Mts. 147.)

Pareto-periaate toteutuu ohjelmistotestauksessa. Pareto-periaate väittää, että 80 prosenttia kaikista testauksen aikana löydetyistä virheistä voidaan suurella todennäköisyydellä jäljittää 20 prosenttiin kaikista ohjelman komponenteista. (Mts. 147.)

Testauksen tulisi alkaa ”pienimuotoisesti” ja edetä pikku hiljaa ”suurimuotoiseen”. Ensimmäisten suunniteltujen testien tulisi kohdistua erillisiin ohjelmakomponentteihin. Kun testaus etenee, sen fokus siirtyy kohti virheiden löytämistä integroiduista komponenteista ja lopulta koko järjestelmästä. (Mts. 147.)

Perusteellinen testaus on mahdotonta. Tämä siksi, että ohjelmapolkujen määrä jopa keskisuuressa ohjelmassa kasvaa poikkeuksellisen suureksi. Tämän vuoksi on mahdotonta suorittaa kaikkia mahdollisia ohjelmapolkuja testausvaiheessa. On kuitenkin mahdollista testata ohjelmalogiikka riittävän hyvin ja var-

mistaa, että kaikki on kunnossa komponenttitasolla. (Mts. 147.)

3.3 Testausstrategiat

Testaus on sarja toimintoja, jotka voidaan suunnitella etukäteen ja suorittaa systemaattisesti. Tämän vuoksi ohjelmistoprosessiin pitäisi sulauttaa myös strategia testaukselle. Tyypillinen testausstrategia sisältää yleensä seuraavanlaisia piirteitä:

- Kehitystiimin tulisi pitää virallisia teknisiä katselmuksia. Tällä tavoin monet virheet löydetään jopa ennen testausvaihetta.
- Testaus alkaa komponenttitasolta ja etenee kohti valmista integroitua järjestelmää.
- Tietyt testaustekniikat sopivat eri tilanteisiin.
- Kehittäjä suorittaa testit. Suuremmissa projekteissa testauksen hoitavat erilliset testaajat.
- Testaus ja debuggaus ovat kaksi eri toimintoa, mutta debuggaus kuuluu olennaisesti jokaiseen testausstrategiaan.

Testausstrategian tulee sisältää sekä alhaisen tason testejä, jotka varmistavat pienten lähdekoodisegmenttien toiminnan, että korkean tason testejä, joilla validoidaan suuret järjestelmätoiminnot asiakkaan vaatimuksia vasten. Lisäksi testauksen etenemisen tulee olla mitattavissa ja ongelmien tulee nousta esille mahdollisimman varhaisessa vaiheessa. (Pressman 2005, 387.)

3.3.1 Yksikkötestaus

Yksikkötestaus (unit-testing) on vianetsintäprosessi, jonka päämääränä on paljastaa vikoja yksittäisissä komponenteissa. Yksikkötestauksella voidaan testata yksittäisiä funktioita tai metodeja, luokkia, joilla on useita attribuutteja ja

metodeja, sekä komponentteja, jotka koostuvat useista luokista, olioista ja/tai funktioista. Yksikkötestaus on tarkoitettu puhtaasti ohjelmavirheiden etsintään, ja se on useimmiten järjestelmän kehittäjien vastuulla. (Sommerville 2007, 547 - 548.)

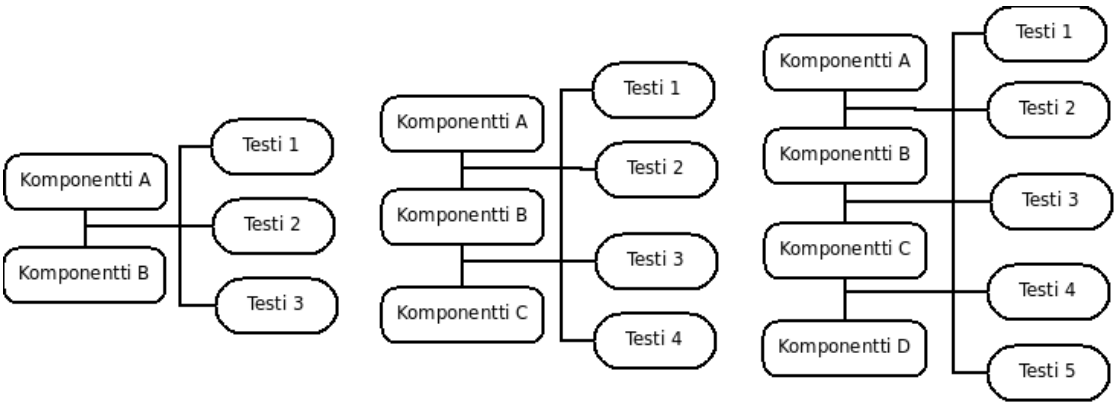
Yksikkötesti on ensimmäinen, alimman tason testausvaihe, joka ohjelmalle suoritetaan. Yksikkötestin aikana kehittäjät testaavat kaikki uudet ja muuttuneet polut ohjelmakoodissa. Tämä sisältää muun muassa syötteiden, silmukoiden, funktioiden paluuarvojen ja kooditason virhekäsittelyn verifiointia. Koska yksikkötestaus kohdistuu suoraan ohjelmointivirheiden etsintään, havaitut virheet voivat olla yksinkertaisia. Jos ne kuitenkin jäävät huomaamatta, ne voivat aiheuttaa suurempia ongelmia myöhemmissä vaiheissa. Tyypilliset yksikkötestauksessa löytyvät virheet koskevat silmukoiden suoritusta, yksinkertaisten sisäisten parametrien käsittelyä, virhetilanteista palautumiseen tarkoitettua logiikkaa sekä virheitä funktioissa ja metodeissa. (Loveland ym. 2005, 29 - 30.)

Koska yksikkötestauksessa fokus on yksittäisen komponentin koodin testaamisessa, se hieman hankaloittaa integroitujen komponenttien yhteistoiminnan testaamista. Lähestymistapa kuitenkin mahdollistaa tiukemman fokuksen uusien ja muuttuneiden ohjelmapolkujen testaamiseen: mitä aiemmassa vaiheessa ongelmat havaitaan, sitä halvempaa ne on myös korjata. Yksikkötestauksen etuna toimii lisäksi testit suorittavan ohjelmistokehittäjän ymmärrys omasta, testattavasta koodistaan. Monesti voidaankin huomata, että yksikkötestaus on kaikkiin muihin testausvaiheisiin nähden kustannustehokkain tapa löytää ja torjua ohjelmavirheitä. (Loveland ym. 2005, 30.)

3.3.2 Integraatiotestaus

Järjestelmäintegrointi tarkoittaa järjestelmän rakentamista sen toiminnallisuuksia toteuttavista komponenteista. Integraatiotestaus on tapa löytää ja selvittää näiden komponenttien keskinäisestä vuorovaikutuksesta muodostuvia ongelmia. Tällä varmistetaan, että komponentit toimivat keskenään oikein ja siirtävät oikeaa tietoa oikeaan aikaan rajapintojensa välityksellä. (Sommerville 2007, 541.)

Integraatiotestauksessa virheiden löytäminen on astetta haastavampaa. Tämän vuoksi sekä integrointi että integraatiotestaus olisi hyvä suorittaa inkrementaalisesti. Integroinnin tulisi lähteä liikkeelle minimaalisella järjestelmäkonfiguraatiolla ja tämän testauksella. Seuraavissa inkrementeissä lisätään komponentteja ja laajempaa kokonaisuutta testataan jälleen. Hyvä nyrkkisääntö on integroida ensin komponentteja, joiden toiminnallisuus on käytössä useimmissa tilanteissa. Tämä varmistaa sen, että käytetyimmät komponentit saavat eniten testausaikaa. Seuraavassa kuviossa (kuvio 3) esitellään yksinkertainen inkrementaalinen integraatiotestaussuunnitelma. (Sommerville 2007, 541 – 542.)



KUVIO 3. Inkrementaalinen integraatiotestaussuunnitelma (Sommerville 2007, 542, muokattu)

Monesti vaaditun toiminnallisuuden testaus voi vaatia usean eri komponentin integroimista. Virheiden korjaus voi osoittautua vaikeaksi usean muutosta tarvitsevan komponentin vuoksi. Lisäksi uusien testattavien komponenttien integrointi voi muuttaa jo aiemmin integroitujen komponenttien vuorovaikutusta, ja yksinkertaisemmalla konfiguraatiolla testatuista toiminnallisuuksista saattaa löytyä uusia virheitä. Jokaisessa inkrementissä on siis tärkeää suorittaa edellisten inkrementtien testit uudelleen. Tätä kutsutaan *regressiotestaukseksi*. Regressiotestaus on kallis prosessi, ja ilman automatisoitua testausarkkitehtuuria se on kovin epäkäytännöllinen. (Sommerville 2007, 543.)

3.3.3 Järjestelmätestaus

Järjestelmätestaus tarkoittaa kahden tai useamman toiminnallisuutta toteuttavan komponentin integroimista ja sen jälkeen integroidun järjestelmän testaamista. Riippuen käytetystä prosessista järjestelmätestaus voi kattaa joko yhden iteraation tai kokonaisen järjestelmän testauksen. (Sommerville 2007, 540 - 541.)

Järjestelmätestauksen päämääränä on varmistaa, että järjestelmä vastaa vaatimuksiaan. Jotta tämä voidaan varmuudella osoittaa, täytyy järjestelmän toteuttaa määritelty toiminnallisuus, tehokkuus ja luotettavuus. Järjestelmätestaus suoritetaan yleensä toiminnallisuustestauksena, koska tässä vaiheessa ollaan kiinnostuneita ainoastaan toiminnallisuudesta, ei ohjelman toteutustavasta. Tämän vuoksi järjestelmätestaus kannattaa hoitaa erillisten testaajien toimesta, joilla ei ole tietoa ohjelman sisäisestä toteutuksesta. (Sommerville 2007, 544.)

Paras tapa varmistaa, että järjestelmä vastaa vaatimuksiaan, on käyttää skenaariopohjaista testausta. Tässä lähestymistavassa kehitetään tietty määrä skenaarioita eli tapahtumakuvauksia, joista testitapaukset juonnetaan. Katta-

vasti suunnitellusta skenaariosta on mahdollista luoda monia hyviä testitapauksia. Tapahtumakuvaukset tulisi myös suunnitella siten, että todennäköisimmät skenaariot testataan ensin ja epätavallisia tai epätodennäköisiä skenaarioita mietitään myöhemmin, jotta testaus kohdistetaan käytetyimpiä järjestelmän osia silmällä pitäen. (Sommerville 2007, 544.)

3.3.4 Suorituskykytestaus

Suorituskykytestauksella tarkoitetaan nimensä mukaisesti järjestelmän suorituskyvyn testaamista. Suorituskykytestauksella pyritään varmistamaan, että järjestelmä kykenee suorittamaan sille tarkoitetun työmäärän. Yleensä suorituskykytestaus koostuu sarjasta testejä, joissa järjestelmän kuormaa lisätään tasaisesti, kunnes järjestelmän toimintakyky alenee merkittävästi. (Sommerville 2007, 546.)

Suorituskykytestauksen tarkoituksena on löytää vahvuudet ja heikkoudet testattavan ohjelmiston suorituskyvystä. Testaustiimi suunnittelee mittareita, jotka kohdistetaan suorituskykykriittisiin alueisiin, jotka on löydetty ohjelmiston kehitysvaiheessa tai asiakkaan toimesta olemassa olevista järjestelmistä. Mittaukset suoritetaan ja tulokset dokumentoidaan, ja näiden perusteella suoritetaan tarkka analyysi. Kohteena ovat niin kutsutut pullonkaulat, jotka rajoittavat järjestelmän vasteaikoja ja tuottavuutta. Nämä ongelmat sijaitsevat yleensä pitkien ohjelmapolkujen sisällä, tärkeimmissä funktioissa. Ne voivat myös liittyä ohjelmiston toimintaan tietyllä laitealustalla. (Loveland ym. 34 - 35.)

Suorituskykyongelmat ovat yleensä kalleimpia virheitä korjattaviksi. Jotkin suorituskykyongelmat voidaan toki ratkaista nopeasti, mutta kriittisimmät niistä vaativat yleensä monimutkaisia ratkaisuja. (Mts. 35 - 36.)

3.3.5 Debuggaus

Debuggaus ei ole testausta, mutta se tapahtuu onnistuneen testauksen seurauksena ja lopulta johtaa virheen korjaamiseen. Kun testitapaus ajetaan ja virheitä löytyy, ne ilmenevät usein ulkoisina oireina, joiden syy on edelleen piilossa. Debuggauksen tarkoituksena on yrittää löytää todellinen syy oireille. Debuggaus on niin oleellinen ohjelmistokehityksen työvaihe, että nykyään kaikki varteenotettavat ohjelmoinnin IDE-työkalut sisältävät debuggerin. (Pressman 2005, 411.)

3.3.5.1 Debuggaustekniikat

Yleisin ja tehottomin tapa selvittää ohjelmavirheen syy on niin sanottu *brute force -lähestymistapa*. Tämä tarkoittaa sitä, että ohjelmaa ajettaessa tulostetaan esimerkiksi muistitietoja ja ajonaikaisia viestejä järjestelmästä. Tästä informaatiotulvasta yritetään löytää vihjeitä, jotka johtavat virheen löytämiseen. Tämä on kuitenkin hidasta ja toimii yleensä huonolla menestyksellä. (Pressman 2005, 414.)

Backtracking (suom. peruutus) on debuggaustapa, jota on helppo käyttää pienissä ohjelmissa. Tämä tarkoittaa sitä, että ohjelmakoodissa ”peruutetaan” manuaalisesti taaksepäin aloittaen oirekohdasta, kunnes ongelman syy löytyy. Valitettavasti ohjelmakoodin lisääntyessä mahdolliset peruutuspolut lisääntyvät ja voivat tulla mahdottomiksi testata. (Pressman 2005, 414.)

Cause elimination (suom. syyn poissulkeminen) toteutetaan suunnittelemalla *syyolettamus*, joka osoitetaan todeksi tai epätodeksi. Vaihtoehtoisesti luodaan lista kaikista mahdollisista syistä ja suoritetaan testejä, joilla poissuljetaan kukin syy listalta. Jos ensimmäiset testit osoittavat, että tietty olettamus näyttää lupaavalta, toimintaa muokataan sen suuntaisesti virheen löytämiseksi. (Pressman 2005, 414.)

3.3.5.2 Virheen korjaaminen

Kun virhe löydetään, se täytyy luonnollisesti korjata. On kuitenkin enemmän kuin mahdollista, että yhden virheen korjaus aiheuttaa uusia virheitä ja täten aiheuttaa enemmän harmia kuin hyötyä. Pressman (2005) ehdottaa kolmea kysymystä, joita kehittäjän tulisi miettiä ennen korjaustoimenpidettä:

- Onko virheen syy jossain muualla ohjelmassa?
- Mitä uusia virheitä suunnittelemani korjaustapa voi mahdollisesti aiheuttaa?
- Mitä olisimme voineet alun perin tehdä estääksemme kyseisen virheen?

(Pressman 2005. 416.)

3.4 Testaustekniikat

Ohjelmistot tulisi suunnitella alusta lähtien *testattavuuden* näkökulmasta. Seuraavat piirteet johtavat testattavaan ohjelmistoon:

- Laatua silmällä pitäen toteutettu ohjelmisto vähentää ohjelmavirheiden määrää.
- Hyvin hallittavissa oleva ohjelmisto helpottaa testauksen automatisointia ja optimointia. Ohjelmiston ajonaikaisten muuttujien tulisi olla suoraan testaajan nähtävissä ja hallittavissa.
- Testauksen jaottelu ja kohdentaminen nopeuttaa ongelmien eriyttämistä ja järkevää uudelleentestausta.
- Ohjelmiston toiminnallinen, rakenteellinen ja koodin yksinkertaisuus nopeuttaa testaamista.
- Ohjelmistomuutosten määrä vaikuttaa suoraan testauksen määrään:

mitä vähemmän muutoksia, sitä vähemmän lisätestausta.

- Ohjelmiston ymmärrettävyys, niin dokumentaation, komponenttiriippuvuuksien kuin lähdekoodinkin näkökulmasta, helpottaa ohjelmiston järkevää testausta.

(Pressman 2005, 422.)

3.4.1 White-box-testaus

White-box-testaus on testitapausten suunnittelutapa, jossa käytetään hyväksi tietoa ohjelman rakenteesta ja toteutuksesta. White-box-testauksella kehittäjä voi luoda testitapauksia, jotka varmistavat, että kaikki yksittäiset ohjelmapolut komponentissa suoritetaan ainakin kerran, kaikkien loogisten valintojen tulosvaihtoehdot käydään läpi, kaikki silmukat suoritetaan rajoitustensa mukaisesti ja ohjelman sisäiset tietorakenteet käydään läpi. (Pressman 2005, 425.)

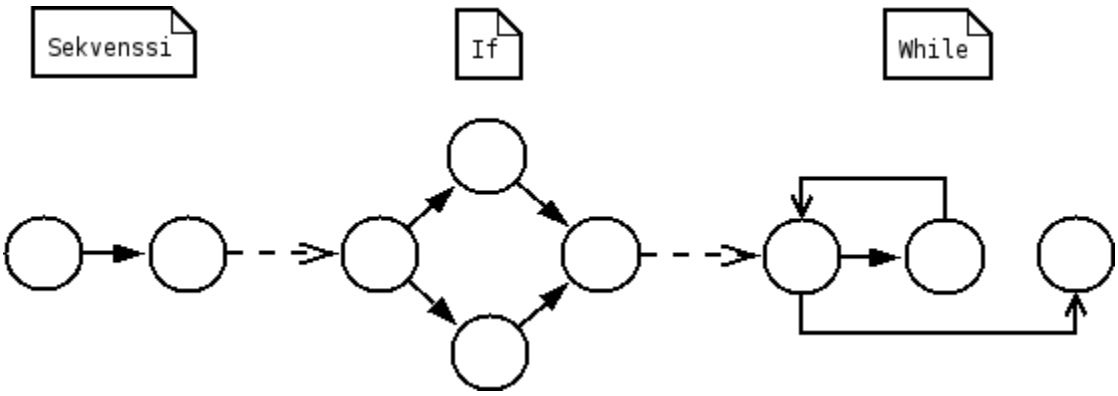
Polkutestaus

Polkutestaus on strukturaalinen testaustekniikka, jonka tarkoituksena on suorittaa kaikki erilliset ohjelmapolut komponentissa tai ohjelmassa ainakin kerran. Polkujen määrä on suoraan verrannollinen ohjelman kokoon. Kun komponenteista integroidaan järjestelmiä, strukturaaliset menetelmät hankaloituvat oleellisesti ohjelmakoon kasvaessa. Tämän vuoksi polkutestausta käytetään usein vain yksikkötestauksen aikana. (Sommerville 2007, 559.)

Polkutestauksella ei suinkaan suoriteta kaikkia mahdollisia polkuyhdistelmiä. Se on käytännössä lähes mahdotonta ohjelman koosta riippumatta, koska esimerkiksi silmukoita sisältävän ohjelman polkuyhdistelmien määrä voi muuttua käytännössä äärettömäksi. Tämä tarkoittaa sitä, että vaikka kaikki ohjelmapolut olisi käyty kerran läpi, virheitä voi esiintyä, kun eri polkuyhdistelmiä suoritetaan. (Sommerville 2007, 559.)

Polkutestaus aloitetaan testattavan ohjelman tai funktion vuokaaviosta. Vuo-

kaavio on ohjelman tai funktion polkujen rakennemalli. Kaavio koostuu noodeista, joista kukin kuvaa yhtä tai useampaa proseduraalista lauseketta, sekä nuolista, jotka esittävät ohjelmapolkujen haarautumista ja etenemistä. Jokainen ehtolausekkeen (if-else tai case) haara esitetään erillisenä polkuna. Nuoli, joka palaa takaisin valintanoodiin, esittää silmukkaa (for, do-while). Seuraavassa kuviossa (kuvio 4) on esimerkki vuokaaviosta. (Pressman 2005, 425 – 426.)



KUVIO 4. Esimerkki polkutestauksen vuokaaviosta

Erillisten polkujen määrän voi selvittää laskemalla vuokaavion syklomaattisen monimutkaisuuden (McCabe's cyclomatic complexity). Ohjelmilla, joissa ei ole goto-lausekkeita, syklomaattisen monimutkaisuuden arvo on yksi enemmän kuin ehtojen määrä ohjelmassa. Yksinkertainen ehto on looginen ehtolauseke ilman ”and”- ja ”or”-yhdistelmiä. Jos ohjelma sisältää komposiittiehtoja, jotka ovat loogisia ehtolausekkeita ”and”- ja ”or”-yhdistelmillä, lasketaan yhteen yksinkertaisten ehtojen määrä komposiittiehdoissa. Kun erillisten polkujen määrä on selvitetty, testitapaukset suunnitellaan siten, että kukin poluista suoritetaan. Kaikkien polkujen suorittamiseen tarvittavien testien minimimäärä on syklomaattisen monimutkaisuuden luku. (Sommerville 2007, 561.)

3.4.2 Black-box-testaus

Black-box-testauksessa tarkoituksena on testata ohjelmiston toiminnallisia vaatimuksia. Siinä ei olla kiinnostuttuja ohjelman sisäisestä rakenteesta vaan testattavan kohteen antamista tulosteista tietyillä syötearvoilla. Testattavan kohteen toimivuutta tarkastellaan haluttujen tai odotettujen tulosten oikeellisuudella. Black-box-testaus ei korvaa white-box-testausta vaan pikemminkin täydentää sitä. Black-box-testauksen hyviä puolia ovat seuraavat:

- Testit voidaan uusia helposti, koska aiempien testien syötteet voidaan yleensä syöttää ohjelmalle sellaisenaan.
- Testattavan kohteen ympäristö tulee myös testatuksi.
- Testaajan ei tarvitse olla välttämättä ohjelmoija, koska testit suoritetaan toimivalla ohjelmalla, speksien perusteella.

Black-box-testauksen huonoja puolia ovat seuraavat:

- Kaikkia ominaisuuksia ei pystytä testaamaan, koska ohjelmaa on työlästä muuttaa kesken testauksen.
- Virheen syytä ei löydetä, koska testaajalla on mahdollista vain havaita virhe, ei sen syytä.
- Testauksen uusimisessa voi ilmetä ongelmia, koska uudelleen ajetut testit löytävät hyvin harvoin täysin uusia virheitä.
- Testaustapa tuottaa lisäkustannuksia.

(Pressman 2005, 434 - 435.)

3.5 Testauksen riittävyys

Tarvittavan testauksen määrää on haastava arvioida. Testausvaihetta, varsinkin järjestelmätestausta, voi jatkaa lähes loputtomiin, jos testauksen riittävyydestä ei ole suunnitelmia. Testauksen mittareille tulisi siis aina määritellä hy-

väksymiskriteerit. (Haikala & Märijärvi 2004, 293 – 294.)

Mutkikkuusmitoilla (complexity measure) pyritään arvioimaan ohjelmamoduulin monimutkaisuutta analysoimalla lähdekoodin rakennetta. Mutkikkuusmitalla voidaan yrittää parantaa ohjelmistosta testausta eniten vaativat komponentit. Tunnetuimpia mutkikkuusmittoja ovat Halsteadin mitta ja McCaben syklomaattinen numero. (Haikala & Märijärvi 2004, 294.)

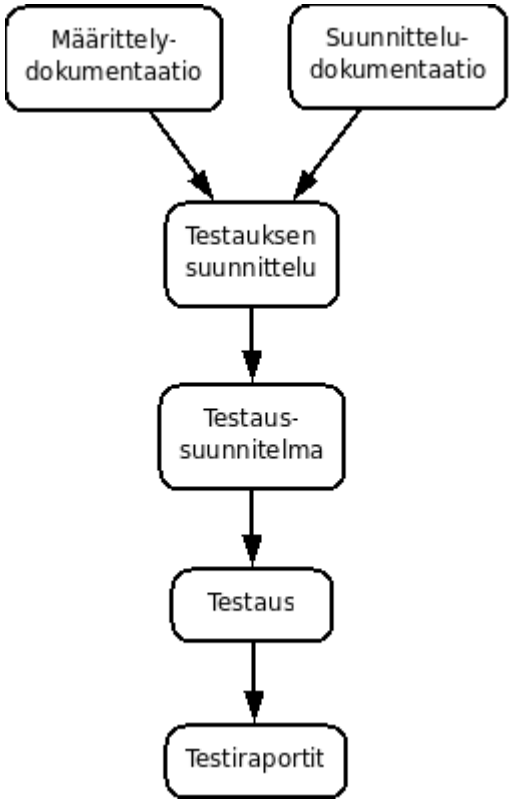
Kattavuusmitat (coverage measure) auttavat testauksen kattavuuden varmistuksessa. Kattavuusmitoilla voidaan siis hankkia todisteita siitä, että testausta on suoritettu riittävästi ja riittävän kattavasti ohjelmiston eri osille. Kattavuusmittoja ovat muun muassa lausekattavuus, päätöskattavuus, ehtokattavuus, päätös/ehtokattavuus ja moniehtokattavuus. Kyseiset esimerkit koskevat vain lähdekoodin kattavuutta (code coverage). Kattavuutta voidaan mitata myös toiminnallisen kattavuuden näkökulmasta (functional coverage). (Haikala & Märijärvi 2004, 294.)

Lausekattavuus (statement coverage) tarkoittaa sitä, että ohjelman jokainen lause suoritetaan vähintään kerran. Pienissäkin ohjelmissa on kuitenkin lähes mahdotonta saavuttaa 100 % lausekattavuutta. Käytännössä todellinen kattavuus ylittää harvoin 90 prosenttia. *Päätöskattavuus* (decision coverage) edellyttää, että ehtorakenteen päätös saa vähintään kerran molemmat arvonsa (tosi tai epätosi). *Ehtokattavuus* (condition coverage) edellyttää, että päätöksen kaikkien ehtojen on saatava kaikki arvonsa. Sekä päätös- että ehtokattavuuden mittaa kutsutaan *päätös/ehtokattavuudeksi*. *Moniehtokattavuus* edellyttää, että testaus suoritetaan kaikkien ehtojen kaikilla yhdistelmillä. (Haikala & Märijärvi 2004, 295.)

3.6 Testauksen dokumentointi

Testaus, kuten muutkin ohjelmistoprosessin vaiheet, voi tuottaa suhteellisen paljon dokumentaatiota. Isomman järjestelmän dokumentaatioksi voi syntyä järjestelmätestaussuunnitelma, integrointitestaussuunnitelma jokaisesta integrointitestistä, komponenttitestaussuunnitelma jokaisesta komponenttitestistä sekä raportointi jokaisesta suoritetusta testistä. Pienemmissä projekteissa usein riittää yksi testaussuunnitelma, joka kattaa integrointitestauksen ja järjestelmätestauksen. (Haikala & Märijärvi 2004, 299.)

Alustava suunnitelma kannattaa laatia määrittelyvaiheessa ja täydentää myöhemmin suunnitteluvaiheessa. Testaussuunnitelmat voidaan sisällyttää projektisuunnitelmaan, toiminnalliseen määrittelyyn ja/tai tekniseen määrittelyyn. Testaussuunnitelmassa määritellään, mitä testejä tehdään ja milloin, testien järjestys sekä odotetut lopputulokset. Lisäksi määritellään testauksen lopettamiskriteerit. Seuraavassa kuviossa (kuvio 5) esitellään yksinkertainen prosessi testaussuunnitelman ja testauksen toteuttamiseen. (Mts. 299.)



KUVIO 5. Testauksen suunnittelu

Eri testaustasojen virheet tulisi raportoida ja analysoida. Raportoitavia tietoja ovat esimerkiksi virheen kuvaus, vakavuus, löytymisajankohta, löytymistapa ja estämismenetelmät tulevaisuuden varalle. Asiakkaalta tuleviin virheilmoituksiin tulisi käyttää erillistä raportointityökalua tai -lomaketta. (Haikala & Märijärvi 2004, 300.)

4 OPAS TESTAUSTYÖKALUIHIN JAVA EE -YMPÄRISTÖSSÄ

Tämä luku käsittelee ohjelmistotestausta Java-ohjelmointikielellä, erityisesti web-sovelluskehityksen näkökulmasta. Esimerkeissä oletetaan, että lukija tuntee Java EE -kehityksen perusteet. Kehitysalustana käytetään Unix-pohjaista käyttöjärjestelmää. Työkaluina käytetään pääasiassa Eclipse IDE:ä (versio 3.4.x) sekä Maven-projektityökalua (versio 2.x). Sovelluspalvelimena toimii

Apache Tomcat (versio 6.x). Käytettyjen Java-kirjastojen versionumerot mainitaan kullekin kirjastolle omistetussa luvussa erikseen.

4.1 Yksikkötestaus Junit-kehyksellä

JUnit on avoimen lähdekoodin testauskehys Java-ohjelmointikielelle ja on ollut Java-kehittäjien yleinen käytäntö yksikkötestien tekemiseen lähes julkaisustaan saakka. JUnitin testausmalli, jota kutsutaan nimellä xUnit, on jopa niin yleisesti hyväksytty, että se on siirtymässä usean ohjelmointikielen standardiksi. JUnit onkin käännetty suurelle osalle nykypäivän ohjelmointikielistä. (Massol & Husted 2004, 5.)

4.1.1 Käyttöönotto

JUnit koostuu yhdestä jar-tiedostosta ja on helppo ottaa käyttöön. Kehittäjän tarvitsee vain lisätä JUnit-kirjasto projektin luokkapolkuun. Mavenia käytettäessä kehittäjien ei tarvitse huolehtia JUnitin erikseen sisällyttämisestä luokkapolkuun, koska JUnit-kirjasto on oletusriippuvuus uusissa Maven-projekteissa.

JUnit-testien ajaminen onnistuu Java-ohjelmassa seuraavalla kutsulla:

```
org.junit.runner.JUnitCore.runClasses(TestClass1.class, ...);
```

Jos testit haluaa ajaa komentoriviltä, täytyy varmistaa, että JUnit sekä testiluokka ovat luokkapolussa ja ajaa seuraava komento:

```
java org.junit.runner.JUnitCore TestClass1.class [...muut
testit...]
```

Kehitystyökalut, kuten Maven ja Eclipse, tukevat JUnit -testejä suoraan, ja niissä on omat toimintonsa testien suorittamista varten. (Beck & Gamma 2009.)

4.1.2 JUnit-testien automatisointi Mavenilla ja Eclipsellä

Maven sisällyttää aina testauksen projektin elinkaareen. Tämän vuoksi testit ajetaan lähes kaikkien kääntämis- ja paketoimiskomentojen yhteydessä, ellei sitä erikseen estetä. Testiluokkien tulee vain sijaita Maven-projektin testeille varatussa kansiossa. Jos kuitenkin haluat ajaa vain testit, voit suorittaa seuraavan komennon:

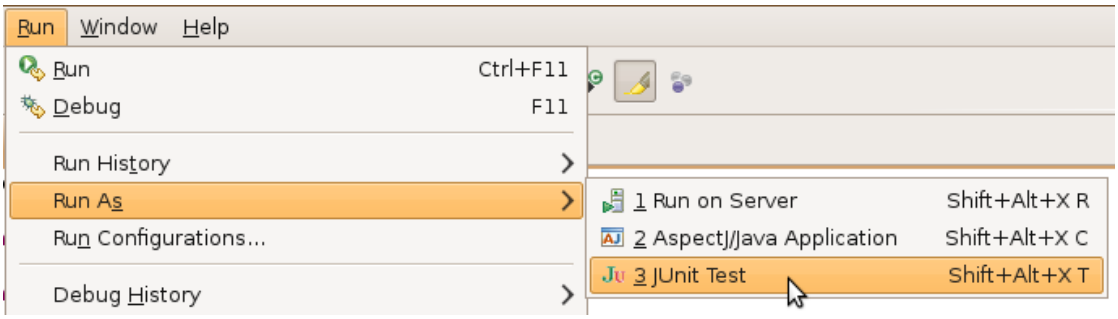
```
mvn test
```

Yksittäisen tai testijoukon suorittaminen onnistuu seuraavalla komennolla:

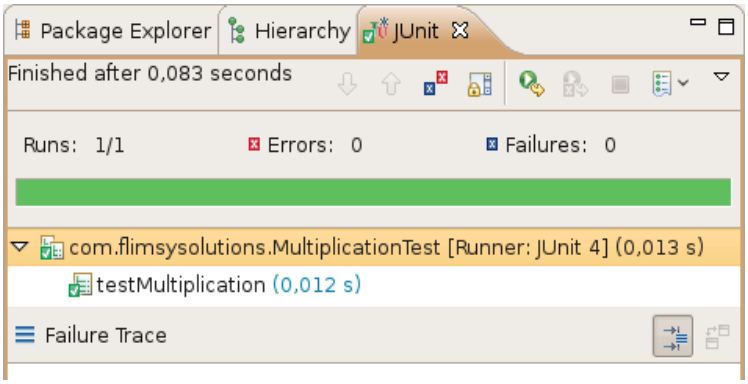
```
mvn -Dtest=MyTest,MySecondTest test
```

Testien tulokset löytyvät oletuksena Mavenin generoimasta target/surefire-reports-kansiosta sekä tekstimuodossa että xml-muodossa.

Eclipsessä testien ajaminen on helppoa graafisen käyttöliittymän siivittämänä. Kun testiluokka on valittuna, valitaan Run- tai pikavalikosta Run As -alivalikko ja sen alta vaihtoehto JUnit Test. Toiminto avaa erillisen JUnit-näkymän, josta voidaan tarkastella ajetun testin tuloksia. Suoritusesimerkki nähdään seuraavista kuvioista (kuviot 6-7).



KUVIO 6. JUnit-testin suorittaminen (Kuvankaappaus Eclipse IDE:stä)



KUVIO 7. JUnit-testin tulokset suorituksen jälkeen (Kuvankaappaus Eclipse IDE:stä)

4.1.3 Testien kirjoittaminen

Yksikkötestien kirjoittaminen JUnitilla on helppoa. JUnitilla (versio 4.x) tehty kertolaskutesti näyttäisi seuraavalta:

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class MultiplicationTest {
    @Test
    public void testMultiplication() throws Exception {
          assertEquals("Test that 3 * 2 = 6", 6, 3 * 2);
    }
}
```

MultiplicationTest-luokan metodi testMultiplication löytyy automaattisesti *Test-annotaation* avulla. JUnit-kehyksen staattisesti linkitettyä metodia assertEquals käytetään testituloksen varmistamiseen. Tämä yksinkertainen testi ei tee oikeutta JUnit-kehyksen laajuudelle, mutta periaate on sama kaikissa sillä toteutettavissa testeissä.

Joskus testitapaus vaatii taustalleen kiinteitä tietorakenteita, joiden tulisi olla valmiiksi alustettuja testin suoritusvaiheessa. Usein tietorakenteiden alustus vie jopa enemmän ohjelmointiaikaa kuin testit itsessään. Alustuksen voi toki suorittaa testiluokan konstruktorissa, mutta tämä vaikeuttaa rakenteiden uudelleenkäyttöä. Tätä toimenpidettä varten JUnit sisältää *Before*- ja *After*-meto-

diannotaatiot, joiden avulla voidaan kätevästi alustaa testitapausten tietorakenteet. Before-metodia kutsutaan ennen testien suorittamista, ja sillä alustetaan muuttujia. After-metodia kutsutaan kaikkien testien suorittamisen jälkeen ja sillä voidaan esimerkiksi vapauttaa kaikki pysyvät resurssit, jotka allokoitiin Before-metodilla. (Beck & Gamma 2009.)

```
public class MoneyTest {
    private Money money;
    @Before public void setUp() {
          money= new Money(12, "EUR");
    }
    @After public void tearDown() {
          // unallocate
    }
}
```

4.1.4 Mock-oliot EasyMockilla

Kun puhutaan yksikkötestauksesta olio-ohjelmoinnin näkökulmasta, ohjelmakoodi on harvoin riippuvaista vain itsestään. Mock-oliot ovat simuloituja olioita, jotka matkivat oikeiden olioiden toimintaa hallitusti. Niiden tarkoitus on erottaa testattava ohjelmakoodi ympäristöstään, jolloin tiettyjen loogisten yksiköiden, kuten yksittäisten metodien, testaus on mahdollista ilman riippuvuutta ohjelmakoodiin, jota ei ole välttämättä vielä toteutettu (esimerkiksi muut komponentit). Parhaimmillaan niin sanottu Mock-strategia mahdollistaa metodikohtaisten testien tekemisen. (Massol & Husted 2004, 140.)

EasyMock on Java-kehys, jonka tarkoituksena on helpottaa Mock-olioiden käyttöä yksikkötestauksessa. EasyMockin käyttöönottamiseksi lisätään kirjasto projektin luokkapolkuun, kuten toimittiin JUnitinkin kanssa. Seuraavassa esimerkissä luodaan Mock-olio JUnit-testin sisällä, jotta voidaan testata ServiceImpl-toteutusta ilman, että DAO-rajapintaa tarvitsee alustaa tai toteuttaa.

```
public class ServiceTest{

     private ServiceImpl service;
     private DAO mockDao;
```

```
    @Before public void setUp() {
        service = new ServiceImpl();
        mockDao = createMock(DAO.class);
        service.setDAO(mockDao);
    }
}
```

EasyMockin createMock-metodi luo DAO-rajapinnasta simuloidun olioinstanssin, joka toteuttaa rajapinnan määrittelemät metodit. Tämän jälkeen mock-olio asetetaan ServiceImpl-oliolle. Testimetodissa määritellään mock-oliolle kutsuttavat metodit sekä niiden paluuarvot.

```
@Test public void testSimpleScenario() {
    String question = "testQuestion";
    expect(mockDao.loadResults(eq(question))).andReturn(true);
    replay(mockDao);
    assertTrue(service.loadResults(query));
    verify(mockDao);
}
```

EasyMockin expect-metodilla määritellään mock-oliolle odotettava metodikutsu sekä paluuarvo. Replay-metodi kertoo EasyMockille, että kaikki halutut asiat on määritelty ja testausvaiheeseen voidaan siirtyä. AssertTrue-metodi suorittaa toteutetun koodin ServiceImpl-oliossa ja varmistaa metodin paluuarvon. Verify-metodilla EasyMock tarkastaa, että kaikki halutut metodikutsut on kutsuttu. Esimerkkikoodissa metodikutsujen järjestyksellä ei ole väliä, mutta mock-olio voidaan luoda myös EasyMockin createStrictMock-metodilla, jolloin järjestys saadaan pakolliseksi.

4.2 Integraatiotestaus Junit-kehyksellä

JUnit mahdollistaa myös komponenttien integraatiotestauksen. Java-komponenttien integraatiotesti ohjelmakooditasolla eroaa yksikkötestistä vain vähän. Suurin ero löytyykin testausympäristön alustuksesta. Toisin kuin yksikkötesteissä, integraatiotestissä käytetään mock-olioiden sijaan toisten komponenttien tarjoamia, toteutettuja rajapintoja. Lisäksi komponenttien DAO-kerros vaatii yleensä todellisen yhteyden tietokantaan, joko oikeaan tai muistinvaraiseen.

Tietokantayhteys ja sen alustus DbUnitilla

DbUnit on JUnit-laajennos, joka on suunnattu tietokantaa käyttävien projektien testaukseen. Sen pääasiallinen tarkoitus on alustaa integraatiotesteille tietokanta, jonka tila voidaan säilyttää testien välillä. Jos testeihin käytettäisiin niin sanottua manuaalista tietokantayhteyttä, jokin testi saattaisi korruptoida tietokannan sisältämän tiedon, ja myöhemmät testit voisivat tämän vuoksi epäonnistua. Eräs DbUnitin kätevä ominaisuus on myös tietokantataulujen ja datan (dataset) lataaminen XML-tiedostoista.

IDatabaseTester on DbUnitin versiossa 2.2 mukaan tullut apuluokka, jonka avulla on helppo toteuttaa uudelleenkäytettävä tietokanta-alustus testejä varten. Seuraavassa esimerkissä alustetaan IDatabaseTester muistinvaraiselle hsqldb-tietokannalle.

```
public class AbstractDatabaseTest {

    private IDatabaseTester databaseTester;
    private DataSource dataSource;

    // ...

    @Before
    public void setUp() throws Exception {
          // käytä hsqldb-alustettua datasourcea
          DatabaseDataSourceConnection connection = new DatabaseDa-
taSourceConnection(dataSource);

          // aseta yhteyden asetukset
          DatabaseConfig config = connection.getConfig();
          config.setProperty(DatabaseConfig.PROPERTY_DATATYPE_FAC-
TORY, new HsqldbDataTypeFactory());
          this.databaseTester = new DefaultDatabaseTester(connec-
tion);
          this.databaseTester.setTearDownOperation(DatabaseOpera-
tion.DELETE_ALL);

          // aseta dataSet
          IDataSet dataSet = getDataSet();
          databaseTester.setDataSet(dataSet);

          // setUp-kutsu (oletus)
```

```
            databaseTester.onSetup();
        }

        @After
        public void tearDown() throws Exception {
            // tearDown-kutsu (oletus)
            databaseTester.onTearDown();
        }

        // ...

    }
```

Edellisessä koodiesimerkissä kutsuttu getDataset-metodi hakee IDataSet-rajapintaa toteuttavan tietokantakuvauksen. FlatXMLDataSet-tyyppisen dataset-olion alustus XML-tiedostosta voidaan toteuttaa esimerkiksi seuraavalla tavalla:

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

Dataset.xml-tiedoston sisällön tulee noudattaa seuraavanlaista skeemaa:

```
<dataset>
    <henkilot id=”1” name=”Henkilo A” />
    <henkilot id=”2” name=”Henkilo B” />
</dataset>
```

Edellämainittujen toimenpiteiden jälkeen käytettävissä on XML-tiedostosta alustettu tietokantayhteys, jonka tila säilyy alkuperäisenä jokaiselle erilliselle testille.

4.3 Järjestelmätestaus

4.3.1 Web-sovellusten testaus Seleniumilla

Selenium on joukko työkaluja, joiden tarkoituksena on automatisoida web-sovellusten testausta. Se tukee useita käyttöjärjestelmiä, ja sitä voidaan hallita useilla ohjelmointikielillä ja testauskehyksillä. Java ja JUnit-kehys ovat suoraan

tuettuja. Seleniumin kolme pääkomponettia ovat

- Selenium IDE, työkalu testien luomiseen
- Selenium RC, työkalu testien automatisointiin eri ohjelmointikielille
- Selenium Grid, mahdollistaa skaalautuvan testijoukkojen ajamisen useissa eri ympäristöissä Selenium RC:llä

Selenium tarjoaa suuren määrän komentoja, joilla voi testata periaatteessa mitä tahansa web-selaimella suoritettavaa toimintoa tai ominaisuutta. Komentokirjasto on nimeltään Selenese, ja se on niin kattava, että sitä voisi sanoa omaksi testauskielekseen. Selenese-komennoilla on mahdollista testata muun muassa seuraavia asioita:

- HTML-käyttöliittymäelementit ja niiden olemassaolo
- halutun sisällön varmistaminen
- rikkinäiset linkit
- taulukon sisältö
- lomakekentät ja lomakkeen lähetys
- JavaScript ja Ajax-toiminnallisuus
- ponnahdusikkunat

4.3.2 Selenium IDE

Selenium IDE on työkalu Selenium-testien luomiseen. Käytännössä se on Mozilla Firefox -web-selaimen lisäosa, joka mahdollistaa testien luomisen lisäksi muun muassa web-sovelluksessa manuaalisesti suoritettujen toimintojen ”nauhoittamisen”, muokkaamisen ja ajamisen testeinä. Lisäksi se mahdollistaa testien viemisen eri ohjelmointikielille ja testauskehyksille.

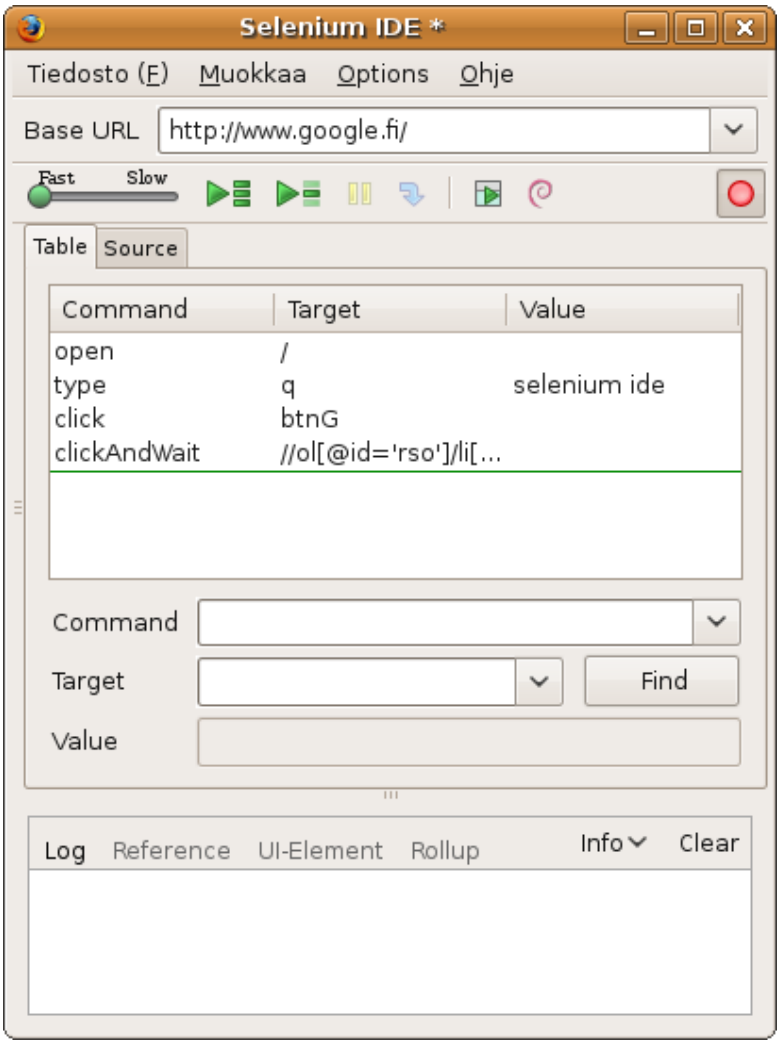
4.3.2.1 Asennus

Selenium IDE:n asennus tapahtuu kuten minkä tahansa Mozilla Firefox -lisäosan. Asennuspaketti löytyy projektin kotisivuilta osoitteesta http://seleniumhq.org/projects/ide/. Asennuksen jälkeen lisäosan voi käynnistää selaimen Työkalut-valikosta.

4.3.2.2 Testin luominen

Kokonaan uuden testin voi luoda asettamalla kohdeosoitteen (Base URL) sekä luomalla uusia Selenese-komentoja. Usein on kuitenkin kätevämpää ensin nauhoittaa haluttu toimintosarja ja lisätä testin lisäkomennot myöhemmin.

Testien nauhoittamiseksi varmistetaan, että nauhoitus on päällä painamalla Selenium IDE:n punainen nauhoituspainike pohjaan. Tämän jälkeen IDE nauhoittaa kaikki web-sivulla tehdyt käyttöliittymäkomennot Selenese-komentoina. Seuraavassa kuviossa (kuvio 8) on esimerkki nauhoitetusta Google-hausta Selenium IDE:ssä. Testi sisältää ”selenium ide” -hakusanan kirjoittamisen hakukenttään, hakupainikkeen valitsemisen sekä ensimmäisen hakutuloslinkin valitsemisen.



KUVIO 8. Nauhoitettu testi Selenium IDE:ssä (Kuvankaappaus Selenium IDE:stä)

Edellä kuvattuun testiin on nyt helppo lisätä komentoja, joilla voitaisiin esimerkiksi varmistaa, että Google-haku palauttaa tiettyjä tuloksia.

4.3.2.3 Testien ajaminen

Testit voidaan ajaa suoraan Selenium IDE:llä. Toimenpide käytännössä suorittaa Selenese-komennot hallitusti, suoraan avoinna olevassa web-selaimessa. Selenium IDE mahdollistaa muun muassa suorituksen hidastamisen, komentokohtaisten pysäytyskohtien (breakpoint) asettamisen ja testien suorittamisen

joukkona. Toiminto vastaa käytöltään yksinkertaista debug-työkalua.

4.3.3 Selenium RC

Joskus testit tarvitsevat muutakin kuin yksinkertaisia selainkomentoja ja sisällön varmistusta. Selenium RC mahdollistaa ohjelmointikielien ja testauskirjastojen käyttämisen monimutkaisempien testien luomiseen ja suorittamiseen. Selenium RC:n käyttö tulee yleensä ajankohtaiseksi siinä vaiheessa, kun Selenium IDE:n tarjoamat ominaisuudet eivät enää riitä ja testauksen automatisointia halutaan viedä pidemmälle.

4.3.3.1 Selenium Server

Selenium Server on palvelin, joka vastaanottaa komentoja testausohjelmasta HTTP-protokollan avulla, tulkitsee ne ja raportoi testien tulokset takaisin ohjelmalle. Käytännössä Selenium Server rakentaa vastaanottamistaan komennoista JavaScript-ohjelman, joka suoritetaan web-selaimessa. Tämän jälkeen testin tulokset lähetetään testiohjelmalle, joka voi tässä tapauksessa olla esimerkiksi Selenium-kirjastoja käyttävä JUnit-testi.

4.3.3.2 Selenium-kirjastot

Selenium-kirjastot tarjoavat rajapinnan, joka mahdollistaa Selenium-komentojen suorittamisen ohjelmointikielellä. Jos Selenium IDE:llä on jo luotu testausskriptejä, voi testit viedä esimerkiksi JUnit-testiksi, jolloin niihin voi lisätä monimutkaisempaa logiikkaa sekä raportointia. Selenium-testejä voi toki ohjelmoida myös ilman Selenium IDE:tä. JUnit-testiksi viety Selenium-testi voisi näyttää esimerkiksi seuraavalta:

```
public class SeleniumTest extends SeleneseTestCase {
    public void setUp() throws Exception {
          setUp("http://www.google.com/", "*firefox");
    }
    public void testSelenium() throws Exception {
          selenium.open("/");
```

```
            selenium.type("q", "selenium ide");
            selenium.click("btnG");
            selenium.click("//ol[@id='rso']/li[1]/h3/a/em");
            selenium.waitForPageToLoad("30000");
        }
    }
```

Testin setUp-metodi käynnistää Firefox-selaimen osoitteeseen http://www.google.com/. Tämän jälkeen Selenese-komennot suoritetaan järjestyksessä. Kuten esimerkistä huomataan, kyseessä on aivan puhdas JUnit-testi, joka vain vaatii rinnalleen Selenium Server -palvelimen sekä web-selaimen.

4.3.3.3 Asennus

Ennen testien ajamista tulee Selenium Serverin olla käynnissä. Koska Selenium Server on Java-ohjelma (selenium-server.jar), se ei tarvitse mitään erityistä asennusta. Riittää, että se ladataan työasemalle ja suoritetaan normaalisti Java-ohjelmana esimerkiksi komentoriviltä.

4.3.3.4 Testien ajaminen

Java-ohjelmointikielelle tarkoitettu Selenium-kirjasto (selenium-java-client-driver.jar) on helppo ottaa käyttöön. Riittää, että se lisätään projektin luokkapolkuun, jolloin sen tarjoama rajapinta on käytössä esimerkiksi JUnit-testejä varten. Kun Selenium Server on käynnissä, testien ajaminen rajapintaa käyttävillä ohjelmilla on mahdollista.

4.4 Suorituskykytestaus Jmeterillä

Apache JMeter on Java-työpöytäsovellus, joka on suunniteltu asiakas- ja palvelinpään ohjelmistojen, kuten web-sovellusten, suorituskykytestaukseen. Sitä voidaan käyttää sekä staattisten että dynaamisten resurssien testaukseen. Sil-

lä voidaan testata esimerkiksi Java Servletit, Java-oliot ja tietokannat. JMeter voi simuloida raskasta palvelin-, verkko- tai oliokuormaa tai analysoida järjestelmien yleistä suorituskykyä erilaisten kuormatyyppien alaisuudessa.

JMeteriä ei tulisi ajaa samalla työasemalla kuin testattavaa järjestelmää, koska testausohjelmiston käyttö saattaa vaikuttaa testattavan järjestelmän suorituskykyyn. Olisi myös suositeltavaa suorittaa testit erillisessä aliverkossa, johon muu verkkoliikenne ei vaikuta. Jos tämä ei ole mahdollista, tulee varmistaa, että muu verkkoliikenne vaikuttaa testeihin mahdollisimman vähän.

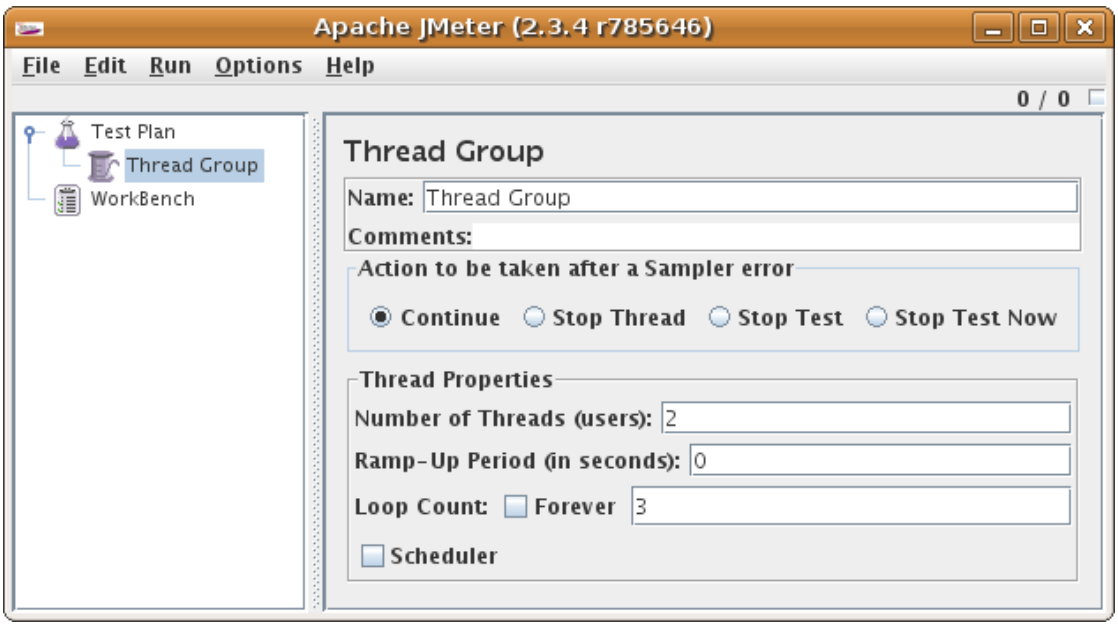
4.4.1 Käyttöönotto

Apache JMeterin voi ladata sivustolta http://jakarta.apache.org/jmeter/. Se on alustariippumaton Java-työpöytäsovellus, joten sen ajamiseen tarvitaan vain Javan ajoympäristö eli JRE.

4.4.2 Testien luominen ja ajaminen

Testin suorittamiseksi luodaan testisuunnitelma (test plan). Testisuunnitelma sisältää toimenpiteet, jotka JMeter tekee suorittaakseen testin. Testisuunnitelma voi sisältää muun muassa säiejoukkoja (thread groups), logiikkaohjaimia, ajastimia, raportteja sekä konfiguraatioelementtejä. Testisuunnitelmalla tulee olla ainakin yksi Thread Group -elementti, joka on testisuunnitelman pääelementti. Se sisältää kaikki muut JMeter-elementit ja ohjaa säikeitä, jotka simuloivat useita samanaikaisia käyttäjiä. Seuraavassa esimerkissä luodaan yksinkertainen Jmeter-testi.

Ensiksi lisätään testisuunnitelmaan Thread Group klikkaamalla hiiren oikealla näppäimellä Test Plan -elementtiä ja valitsemalla Add ja sitten Thread Group. Seuraavassa kuviossa (kuvio 9) näemme lisätyn Thread Group elementin.

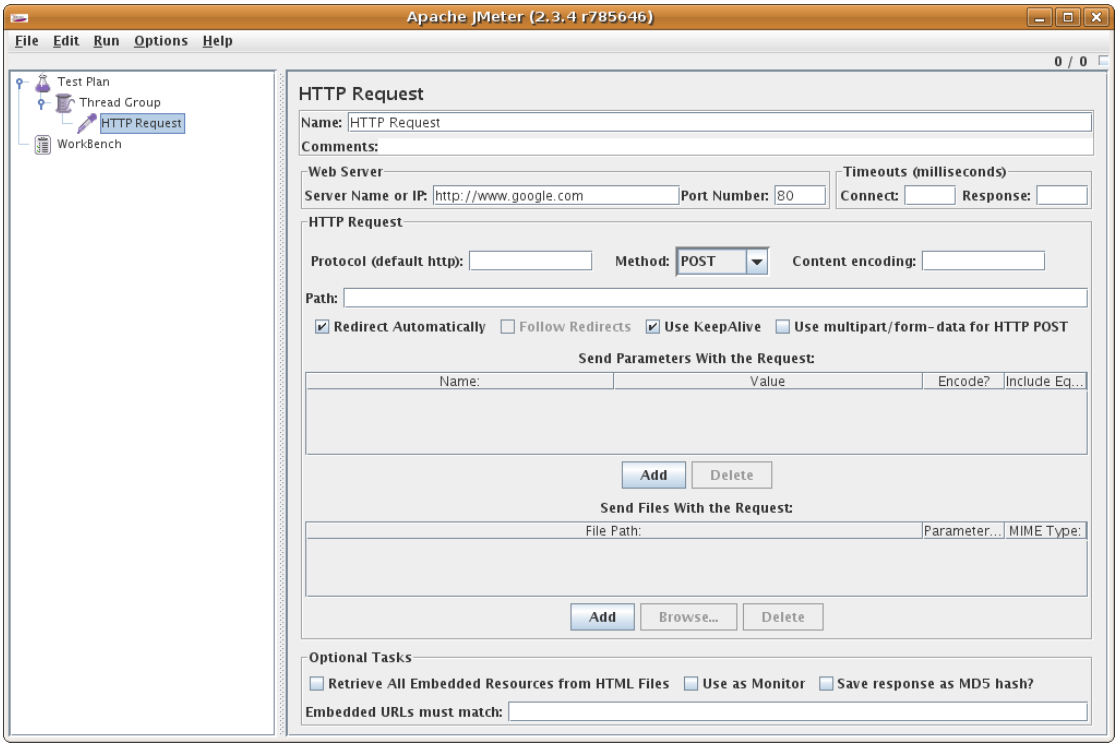


KUVIO 9. Thread Group -elementin lisääminen testisuunnitelmaan (Kuvankaappaus JMeter-ohjelmasta)

Thread Group -elementti sisältää kolme tärkeää attribuuttia

- Number of Threads – säikeiden lukumäärä, eli käyttäjämäärä
- Ramp-Up Period – aika sekunteina, joka varataan säikeiden luomiselle
- Loop Count / Forever – testin suorituskertojen lukumäärä.

Thread Group -elementin alle tarvitaan itse testi. Esimerkiksi HTTP-pyynnön lisääminen tapahtuu asettamalla HTTP Request -elementti Thread Group -elementin alle. Se tapahtuu klikkaamalla hiiren oikeaa näppäintä Thread Group -elementtiin ja valitsemalla Add, Sampler ja sitten HTTP Request. Seuraavassa kuviossa (kuvio 10) näemme lisätyn HTTP Request -elementin.

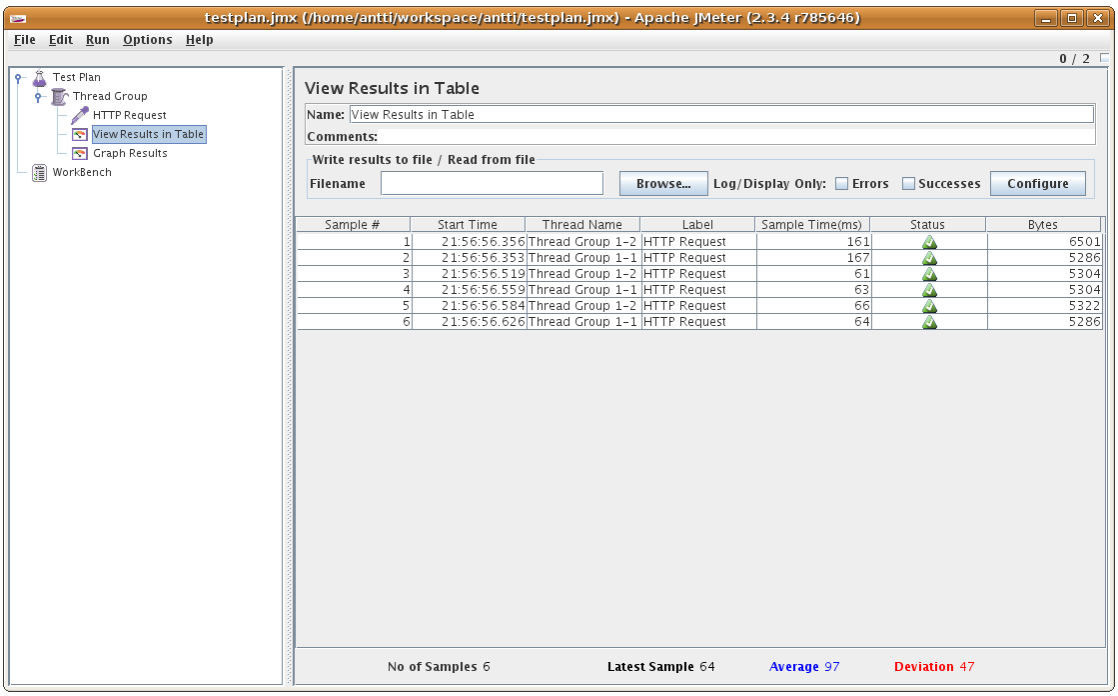


KUVIO 10. HTTP Request -elementin lisääminen testisuunnitelmaan (Kuvankaappaus JMeter-ohjelmasta)

Erimerkkitestinä toimiva HTTP Request -elementti sisältää muun muassa seuraavat kentät

- Server Name or IP – palvelimen nimi tai IP-osoite, johon kutsu lähetetään
- Method – kutsun tyyppi voi olla joko POST tai GET
- Path – polku, johon kutsu lähetetään.

Kun nämä kentät on asetettu, testi voidaan ajaa. Ennen sitä tarvitaan kuitenkin tapa esittää testin tulokset. Thread Group -elementin alle voi lisätä esimerkiksi View Results in Table -elementin, joka näyttää tulokset taulurakenteessa. Tämä on helpoin tapa esittää testin tuloksia, mutta JMeter sisältää myös useita muita esitystapoja. Seuraavassa kuviossa (kuvio 11) näemme View Results in Table -elementin.



KUVIO 11. Suorituskykytestin tulokset View Results In Table -elementissä (Kuvankaappaus JMeter-ohjelmasta)

4.5 Testauksen kattavuuden laskeminen Coberturalla

Cobertura on ilmainen työkalu Java-ohjelmointikielelle testien kattavuuden laskemiseen. Sen avulla kehittäjät voivat löytää ohjelmakoodistaan vähemmälle testaukselle jääviä alueita. Cobertura laskee prosenttimääräisesti testien kattaman koodirivien ja -haarojen määrän kullekin luokalle, paketille sekä koko projektille. Lisäksi se laskee McCaben syklomaattisen monimutkaisuuden kullekin luokalle ja näyttää keskimääräisen syklomaattisen monimutkaisuuden kullekin paketille ja koko projektille. Cobertura voi generoida laskemansa raportit sekä HTML- että XML-muodossa.

Kehittäjien helpotukseksi Maven tukee Coberturaa suoraan. Kattavuuslaskun voi suorittaa seuraavalla komennolla projektikansion juuressa:

```
mvn cobertura:cobertura
```

Koska Coberturan käyttäminen oletusasetuksilla on tehty näinkin helpoksi, on pelkästään hyödyllistä seurata testauksen kattavuutta sen avulla edes silloin tällöin. Kannattaa kuitenkin pitää mielessä, että testauksen kattavuuden valvominen on vain laatutyökalu. On hyvä pitää testauksen kattavuusprosenttia korkealla, mutta joissain tapauksissa kattavuuden väkinäinen nostaminen tuottaa enemmän haittaa kuin hyötyä, sillä mitä lähemmäs 100 %:n kattavuutta mennään, sen haastavammaksi ja siten kalliimmaksi kattavuuden nostaminen käy.

Seuraavassa kuviossa (kuvio 12) näemme esimerkin Coberturan tuottamasta HTML-raportista.

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	55	75%	1625/2179	64%	472/738	2.319
net.sourceforge.cobertura.ant	11	52%	170/330	43%	40/94	1.848
net.sourceforge.cobertura.check	3	0%	0/150	0%	0/76	2.429
net.sourceforge.cobertura.coveragedata	13	N/A	N/A	N/A	N/A	2.277
net.sourceforge.cobertura.instrument	10	90%	460/510	75%	123/164	1.854
net.sourceforge.cobertura.merge	1	86%	30/35	88%	14/16	5.5
net.sourceforge.cobertura.reporting	3	87%	116/134	80%	43/54	2.882
net.sourceforge.cobertura.reporting.html	4	91%	475/523	77%	156/202	4.444
net.sourceforge.cobertura.reporting.html.files	1	87%	39/45	62%	5/8	4.5
net.sourceforge.cobertura.reporting.xml	1	100%	155/155	95%	21/22	1.524
net.sourceforge.cobertura.util	9	60%	175/291	69%	70/102	2.892
someotherpackage	1	83%	5/6	N/A	N/A	1.2

KUVIO 12. Coberturan tuottama HTML-raportti

4.6 Debuggaus Eclipse IDE:llä

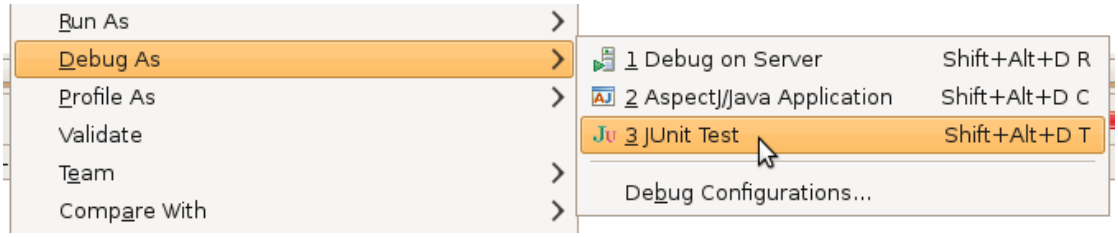
Debuggaus on erottamaton osa ohjelmistokehitystä, ja tämän vuoksi Eclipse IDE sisältää oletuksena debuggerin. Sen käyttöönottaminen ei vaadi mitään erityistoimenpiteitä, vaan käyttö on saumatonta IDE:n muiden toiminnallisuuksien kanssa. Eclipsen Java-lähdekoodieditori tukee pysäytyskohtien (breakpoints) lisäämistä suoraan tuotettavaan lähdekoodiin, mistä debugger osaa ne ohjelman suoritusvaiheessa poimia.

Eclipse IDE:n debug-tila tukee sekä tavallisten ohjelmien että JUnit-testien debuggausta. Koska web-sovellukset ajetaan yleensä erillisellä sovelluspalvelimella, Eclipse IDE tukee myös etädebuggausta (remote debugging), jolloin jopa tuotantoympäristön debuggaus on periaatteessa mahdollista.

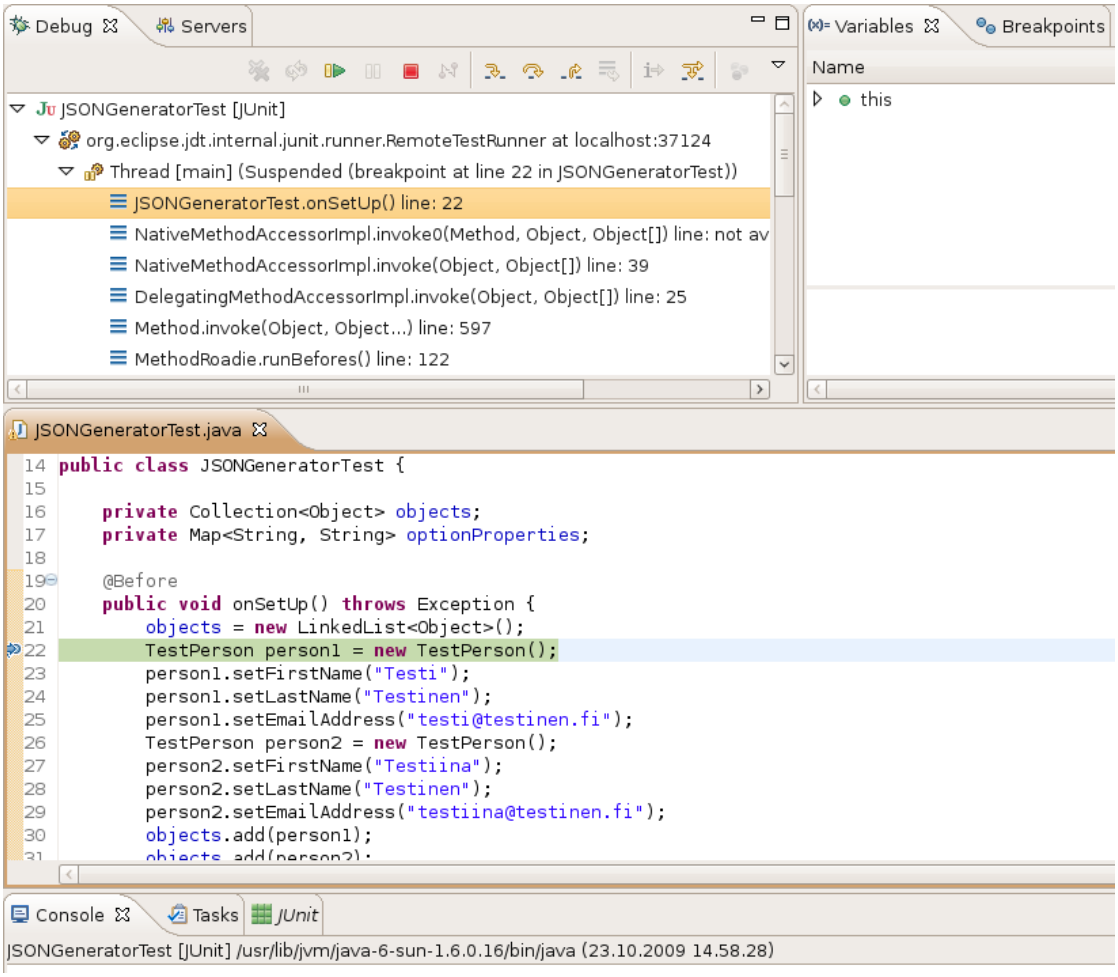
4.6.1 JUnit-testien debuggaus

Koska JUnit-testit voidaan ajaa suoraan Eclipse IDE:n kautta, tämä mahdollistaa myös niiden helpon debuggauksen. Debuggaus helpottaa testitapausten kehitystä olennaisesti, sillä testeissä itsessäänkin voi luonnollisesti olla ohjelmavirheitä. Debuggerin avulla testien kehitys on helpompaa, nopeampaa ja tehokkaampaa. Lisäksi olennainen osa testausta on selvittää testien löytämiä virheitä debuggerin avulla.

JUnit-testin ajaminen debug-tilassa voi tapahtua joko Run- tai pikavalikosta. Seuraavassa kuviossa (kuvio 13) testiluokan pikavalikko on avattu oikealla hiirennäppäimellä. Avautuneesta valikosta valitaan Debug As -alivalikko ja sen alta Junit-test. Kuviossa 14 näemme avautuneen Debug-tilan Eclipse IDE:ssä.



KUVIO 13. JUnit-testin suoritus aloitetaan debuggerin avulla (Kuvankaappaus Eclipse IDE:stä)



KUVIO 14. JUnit-testi ajetaan debug-tilassa (Kuvankaappaus Eclipse IDE:stä)

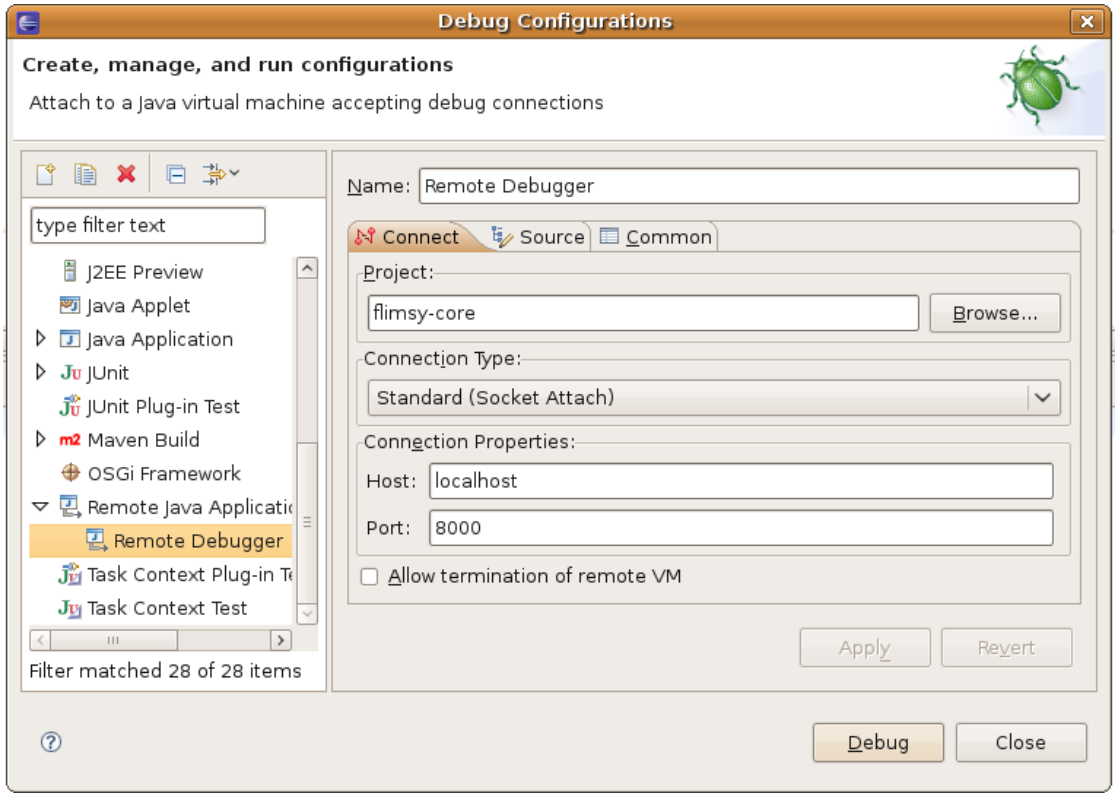
4.6.2 Sovelluspalvelimen etädebuggaus

Tomcat-sovelluspalvelin (versio 6.x) tukee debuggausta etäyhteydellä. Kehittäjän tulee varmistaa, että Tomcat-palvelimen *jpda-tuki* (Java Platform Debugger Architechture) on päällä. Tämä tapahtuu käynnistämällä palvelin esimerkiksi seuraavalla komennolla:

```
bin/catalina.sh jpda start
```

Eclipsessä etädebuggauksen saa päälle valitsemalla Run-valikosta *Debug Configurations*. Valinta avaa näkymän, jossa voi luoda uuden profiilin käytössä olevalle sovelluspalvelimelle kohdassa *Remote Java Application*.

Sekä Tomcatin että Eclipse IDE:n jpda-oletusportti on 8000. Portin asetuksia voi muuttaa Tomcatin konfiguraatiotiedostoista ja Eclipsen Debug Configurations -näkymästä. Ideaalisessa tilanteessa kehittäjän ei kuitenkaan tarvitse muuttaa asetuksia. Usein vain Tomcatin käynnistäminen jpda-tuella ja Remote Java Application -profiilin luominen projektille riittää. Usein tässä vaiheessa kannattaa myös lisätä Source-välilehdelle muut projektit, joista Eclipsen debugger etsii tarvitsemansa lähdekoodit ohjelman suoritusvaiheessa. Tämän jälkeen painetaan Debug-painiketta, jolloin etädebuggaus on käytössä. Seuraavassa kuviossa (kuvio 15) näemme avatun Debug Configurations -ikkunan.



KUVIO 15. Eclipsen Remote Java Application -konfiguraatio (Kuvankaappaus Eclipse IDE:stä)

5 POHDINTA

Työn tekeminen oli osittain vanhan tiedon kertaamista, mutta toisaalta se sisälsi myös paljon uutta. Koulusta saadusta teoriapohjasta ja työelämässä hankitusta kokemuksesta oli paljon hyötyä, sillä ohjelmistotestauksen taustatutkimusta tehdessä monet asiat olivat tuttuja. Kuitenkin perehtyminen syvällisemmin ohjelmistotestaukseen antoi monia uusia oivalluksia, mikä teki työn tekemisestä mielekästä.

Työn positiivisin asia oli se, että ohjelmistotuotannosta ja -testauksesta löytyi paljon taustatietoa. Tämä johtui luonnollisesti siitä, että ohjelmistotestaus on ohjelmistotuotannon tärkeimpiä osa-alueita ja siten hyvin tutkittu aihealue. Lisäksi Java EE -teknologia on testaustyökalujensa osalta erinomaisesti tuettu ja dokumentoitu, mikä osaltaan myös helpotti työn viimeisen luvun eli oppaan tekemisessä. Suurin haaste työn tekemiselle oli tiukka aikataulu, joka johtui suurelta osin arkielämän työkiireistä sekä päivittäisestä tuntien työmatkasta. Aikataulu tuntui välillä jopa mahdottomalta esteeltä. Jälkeenpäin ajateltuna aikatauluhaasteet kuitenkin opettivat paljon tehtävien priorisoinnista ja työn rajaamisesta.

Työn lopputuloksena saatiin aikaiseksi raportti ohjelmistotestauksesta ja opas Java EE -testaukstyökalujen käyttöön. Se riittää perusteiden opettelemiseen ja on siten hyödyllinen sekä uusille, että kokeneemmille kehittäjille. Työn parantamiseksi olisi voinut vielä tutkia tarkemmin ohjelmistotestausta ja siihen liittyvää prosessia. Lisäksi olisi ollut hyödyllistä syventyä hieman tarkemmin valittuihin työkaluihin.

Olen sitä mieltä, että työstä tulee olemaan hyötyä toimeksiantajayritykselle ja julkisena raporttina myös muille ohjelmistotestauksesta kiinnostuneille. Erityisesti Java EE -kehittäjille työkaluopas on hyödyllinen työkalujen käytössä ja käyttöönotossa. Mielestäni työn lopputulos on hyvä.

LÄHTEET

Beck, K. & Gamma, E. 2009. JUnit Cookbook. Opas JUnit-kehyksen käyttöön ja käyttöönottoon. Viitattu 15.10.2009. http://junit.sourceforge.net/doc/cookbook/cookbook.htm.

Haikala, I. & Märijärvi, J. 2004. Ohjelmistotuotanto. Helsinki: Talentum.

Loveland, S., Miller, G., Prewitt, R. & Shannon, M. 2005. Software Testing Techniques: Finding the Defects that Matter.

Massol, V. & Husted, T. 2004. JUnit in Action. Greenwich: Manning.

OPEN Process Framework. 2005. Artikkeli komponenttisuunnittelumallista. Viitattu 29.9.2009. http://www.opfro.org/, work unit, activity, design, software component design.

Pressman, R. 2005. Software Engineering: A Practitioner's Approach, Sixth Edition. Boston (MA): McGraw-Hill.

Software Errors Cost U.S. Economy $59.5 Billion Annually. 2002. Uutinen ohjelmistotestauksen tutkimustuloksista. Viitattu 29.9.2009. http://www.nist.gov/public_affairs/releases/n02-10.htm.

Sommerville, I. 2007. Software Engineering, Eighth Edition. Harlow: Pearson Education.