Testaussuunnitelma

Boa Open Access

Helsinki 5.5.2006

Ohjelmistotuotantoprojekti

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Kurssi

581260 Ohjelmistotuotantoprojekti (6 ov)

Projektiryhmä

Ilmari Heikkinen
Timo Hintsa
Erno Härkönen
Arto Vuori
Mikko Kautto

Asiakas

Olli Niinivaara

Johtoryhmä

Juha Taina
Riikka Kaven

Kotisivu

`http://www.cs.helsinki.fi/group/boa`

Versiohistoria

Versio	Päiväys	Tehdyt muutokset
1.0	3.2.2006	Ensimmäinen versio
1.1	17.2.2006	
1.2	24.2.2006	
1.3	13.3.2006	Ensimmäisen iteraation lopullinen versio
1.4	4.5.2006	
1.5.	5.5.2006	Lopullinen versio testauskirjanpitoineen

Sisältö

1 Johdanto

Tämä dokumentti on testaussuunnitelma BOA Open Access -ryhmän transformaattori-projektille. Transformaattorin avulla metadataa voidaan koota ja muuttaa Atomilauseformaattiin jatkokäsittelyä varten. Työkalu mahdollistaa metadatan harvestoinnin (noutamisen) tällaista palvelua tarjoavalta palvelimelta tai paikallisesta tietovarastosta (esim. cd-rom) sekä metadatan tallentamisen paikallisesti jatkokäsittelyä varten. Tämä dokumentti pohjautuu ohjelmistotuotantoprojekti Metaxan (http://www.cs.helsinki.fi/group/metadata) tuottamaan testausdokumenttiin.

Testaussuunnitelmassa kartoitetaan se, miten projektissa tuotettavan järjestelmän testaus suoritetaan. Testauksen tavoitteena on osoittaa sekä asiakkaille että projektille, että ohjelmisto täyttää sille asetetut vaatimukset (järjestelmätestaus) ja löytää ohjelmistosta puutteita ja virheitä, joiden johdosta ohjelmisto ei toimi, toimii väärin tai ei vastaa sille asetettuja määrittelyjä. Tämä on määritysten ja syntaksin testausta (yksikkötestaus ja integrointitestaus).

2 Yksikkötestaus

Komponenttien testaus perustuu ensisijaisesti komponentille asetettuihin vaatimuksiin. Kaikille testattaville komponenteille tulee olla perustelu niiden käytölle, eli lista vaatimuksia palveluista, jotka niiden oletetaan virheettömästi toteuttavan. Koodin täytyy luonnollisesti olla syntaktisesti virheetöntä, jotta sen pystyy kääntämään ja suorittamaan, ja sitä kautta soveltamaan dynaamisia testausmenetelmiä.

Luokan koodaaja voi halutessaan testata luokan yksittäisiä metodeita ennen kuin koko luokka on kirjoitettu loppuun (ja on yksikkötestausvaiheessa). Testattavat metodit ja myöhemmin koko testiluokka tulee kommentoida JavaDoc-tyylisesti (katso projektin koodin tyyliohje ja Sunin suositus http://java.sun.com/j2se/javadoc/writingapispecs/index.html).

Testiluokan kommentteihin tulee sisältyä selvitys sen toiminnasta, jotta palvelupohjaista testausta voidaan suorittaa (ja jotta sen voisi suorittaa joku muukin kuin luokan kirjoittaja). Jos joitakin metodeja on testattu luokan kehitysvaiheessa tulee nämä metodit kuitenkin testata uudelleen ainakin kertaalleen luokan tultua yksikkötestausvaiheeseen. Uudelleen testaaminen tulee olemaan suoraviivaista koska testit toteutetaan kirjoittamalla niitä varten JUnit-testiluokat. Ohjelmoitu luokka on valmis kun se on kirjoitettu loppuun ja kokonaisuudessaan testattu toimivaksi.

Jos valmista luokkaa muutetaan siten että luokan API muuttuu on muutoksesta ja vaikutuksista keskusteltava muiden komponenttien tekijöiden, suunnittelijoiden tai testaajien kanssa ennen muutoksen tekemistä. Testaus on uusittava ainakin muutetun komponentin osalta, ja mahdollisesti muidenkin komponenttien osalta. Käytännössä tämä tarkoittaa siis sitä että jos joku muu on kirjoittanut luokan joka tulee toisen henkilön testattavaksi ovat muutokset sallittuja jos ne eivät vaikuta muihin komponentteihin.

2.1 Testausmenetelmät

2.1.1 Luokan testaaminen

Luokka testataan testaamalla sen metodit (kun ne on integroitu luokkaansa). Ulkopuolelta voidaan kutsua vain public-metodeja ja Javan oletusnäkyvyydessä olevia metodeita, joten oletetaan että luokan private- ja protected-metodit tulevat testatuksi kun kaikki julkiset metodit on testattu. Parametrisoitavat metodit testataan kutsumalla niitä erilaisilla parametreilla. Lisäksi metodin käyttäytyminen voi riippua luokan tai olion tilasta, jolloin testaus tulisi toistaa eri tiloissa.

Metodille annettavat testiparametrit muodostavat ekvivalenssiluokkia. Saman ekvivalenssiluokan sisällä olevien parametrien oletetaan tuottavan samankaltaisen käyttäytymisen metodissa. Testiparametreiksi valitaan metodin ulkoisten tai sisällä olevien rajoitusten lähellä olevat arvot (raja itse ja raja +-1), sekä jokin arvo luokan arvoalueen ”keskeltä”. Esimerkiksi if -lauseet ja silmukat tuottavat helposti tällaisia rajoja.

Luokan kenttien lukeminen ja asettaminen on yleensä kapseloinnin hengen mukaisesti toteutettu get- ja set-metodeilla, jolloin testaus tulee tehtyä metoditestauksen lomassa. Jos luokassa on edellä mainituilla metodeilla varustamattomia kenttiä jotka eivät ole muuttumattomia (FINAL) ja joiden näkyvyys on public on testattava saako kenttä oikeita arvoja eri tilanteissa ja vaikuttaako kentän asettaminen kuten sen kuuluisi. Tämäntyyppistä kapseloinnin hengen vastaista ohjelmointitapaa pyritään kuitenkin välttämään ja suositellaan että ohjelmakoodi mietitään kentän osalta uusiksi ennen testausta. Tapauksissa joissa esimerkiksi käytetään uudelleen muiden tahojen tuottamaa ohjelmakoodia jossa on ohjelmoitu kenttiä tällä tavalla voidaan edetä testaustauksessa edellä mainitulla tavalla.

Ryhmän tuottamien luokkien aliluokkien testauksen yhteydessä on testattava kaikki yläluokankin metodit, myös ne joita aliluokka ei korvaa, tai näytä vaikuttavan niiden toimintaan (koska se voi vaikuttaa). Perintä lisää testattavan aineiston määrää, mutta sen voi korvata kompositiolla. Tämä onkin suositeltava toimintatapa.

Jos luokka sisältää yhteyksiä muihin luokkiin, tulee näiden luokkien olla yksikkötestattuja jos ne ovat välttämättömiä testin suorittamisen mahdollistamiseksi. Javan valmiina tarjoamia luokkia pidetään virheettöminä (ts. ne täyttävät tällöin myös yksikkötestauksen kriteerit). Muiden komponenttien osalta tämä tarkoittaa, että ne on testattu vähintään tässä dokumentissa määrätyllä kattavuudella. Jos muita luokkia ei ole testattu riittävästi, täytyy niitä käyttävän luokan testausta lykätä, tai korvata muut luokat vastaavilla tynkäluokilla. Näitä muutaman tiiviissä yhteistyössä olevan luokan kokoelmia kutsutaan ryppäiksi. Integrointitestaus koskee ryppäiden yhteistoiminnan testaamista.

2.1.2 Black-box -testaus (vastuupohjainen testaus)

Black-box -testaus perustuu testattavalle komponentille asetettuihin vaatimuksiin. Tiedetään mitä palveluja komponentin halutaan toteuttavan ja mahdollisesti myös miten (esim. aikavaatimukset). Komponentti on ”musta laatikko”, jolle voidaan antaa syöte, ja se palauttaa siihen perustuvan tuloksen. Komponentin tiedetään toimivan oikein, kun sille an-

netaan jokin syöte jota vastaava tulos tiedetään etukäteen, ja komponentti antaa tuloksena saman tuloksen.

Komponentin eri tiloja ja niiden vaikutusta sen toimintaan ei ehkä voida tietää, joten testataan olion metodeita ainakin sellaisessa järjestyksessä, kuin niitä ”luonnollisesti” kutsuttaisiin. Järjestyksiä voi olla useita, joten olio joudutaan ehkä luomaan useampaan kertaan testin aikana. Niitä julkisia metodeita, joita ei tulla koskaan suorittamaan, ei tarvitse testata. Mikäli koodiin jätetään tämäntyyppisiä metodeja, esimerkiksi kun sovelletaan valmiita ohjelmakomponentteja on suositeltavaa kommentoida metodit pois projektiryhmän koodiohjeen mukaisesti. Luonnollisesta poikkeavien kutsumisjärjestysten testaus on sallittua. Poikkeavien suoritusjärjestysten testaaminen on suositeltavaa jos on mahdollista että esimerkiksi jokin muu komponentti käyttää luokassa olevia julkisia metodeja ”luonnollisesta” poikkeavassa järjestyksessä. Jos metodien suoritusjärjestys on tarkasti tunnettu ei muiden suoritusjärjestysten testausta tarvita.

Metodin testiparametrien ekvivalenssiluokat päätellään komponentille asetettujen vaatimusten perusteella. Komponentin vaatimukset käyvät ilmi kunkin komponentin API-kuvauksesta. APIn perusteella voidaan myös päätellä metodin palauttama arvo. Jos APIssa määritellään jokin syöte kielletyksi tulee metodi testata myös tuolla syötteellä jotta varmistutaan siitä että mahdolliset poikkeukset yms. poimitaan. Jos kiellettyä syötettä ei poimita tai muuten tarkisteta tulee asiasta olla maininta APIssa. Muita poikkeuksellisia parametreja voidaan testata harkinnanvaraisesti. Tilanteissa joissa API-kuvaus on virheellinen tai puutteellinen tulee se korjata.

2.1.3 Glass-box -testaus (toteutuspohjainen testaus)

Täydentää black-box -testausta ottamalla huomioon testattavan komponentin lähdekoodin ja tilakaavion. Metodin testiparametrien ekvivalenssiluokat johdetaan palveluvaatimusten lisäksi metodin lähdekoodista. Ekvivalenssiluokista muodostettuja testitapauksia tulee olla niin monta että metodin jokainen lause suoritetaan testin aikana ainakin kerran (100% lausekattavuus). Metodi täytyy testata erikseen luokan/olion jokaisessa eri tilassa, jolla on vaikutusta metodin toimintaan. Lisäksi testataan metodeita sellaisessa järjestyksessä, että tilakaavion jokainen siirtymä tulee testatuksi ainakin kertaalleen. Tähän ei välttämättä riitä olion luominen kertaalleen testin aikana.

Jotta 100% lausekattavuus saavutetaan ei metodeissa saa olla turhia rivejä. Luokkien koodaajien tulee siis pyrkiä varmistumaan siitä että koodiin ei jää rivejä joita ei voida saavuttaa. Käytännössä tämä tarkoittaa esimerkiksi sitä että switch-käskyssä korvataan jokin case default-nimikkeen alle. Mikäli ohjelmakoodiin jää kaikesta huolimatta rivejä joita ei pystytä koskaan suorittamaan ja täten testaamaan niin asiasta tulee mainita testiluokassa. Tällöin Coverlipse-pluginin laskema lausekattavuus jää alle 100%:n. Luokka pidetään riittävän kattavasti testattuna jos 100%:sta vajaa luku selittyy koodiriveillä joista on tehty maininta testiluokkaan.

2.2 Ulkoiset vaatimukset

Testattava komponentti voi toteuttaa tai käyttää hyväkseen jotain ryhmän ulkopuolisen tahon määrittämää standardia tai spesifikaatiota (esim. OAI-PMH -protokolla).

Tällöin on testattava, että komponentti toteuttaa vähintään spesifikaation pakolliset osat virheettömästi niiltä osin kuin niitä tarvitaan komponentille asetettujen vaatimusten tyydyttämiseksi. Valinnaisista toiminnoista testataan vain ne, joita tarvitaan. Yleensä spesifikaatioissa olevat optiot eivät ole olennaisia komponentin toiminnallisuuden kannalta. Komponentin lähdekoodia tulisi tutkia siltä varalta, että komponentti saattaa toteuttaa ja käyttää speksin valinnaisia osia pakollisten osien toteutukseen.

2.3 Menetelmien soveltaminen

Ryhmän itse tuottamat komponentit testataan black-box-menetelmällä, jota täydennetään glass-box-menetelmällä.

Valmiista komponenteista arvioidaan ensin, kuinka kattavasti ne on valmiiksi testattu. Jos testaus arvioidaan riittämättömäksi, sovelletaan black- tai glass-box -menetelmää riippuen esim. komponentin laajuudesta, dokumentaatiosta ja selkeydestä. Käytännössä kaikkien valmiiden komponenttien lähdekoodin pitäisi olla saatavilla, koska lisenssiasioiden takia voidaan käyttää vain GPL:n alaisia komponentteja.

Ulkoisia vaatimuksia toteuttavat komponentit testataan glass-box -menetelmällä ainakin ko. vaatimusten osalta. Lisäksi voidaan käyttää tynkiä ym. tässä määrittelemättömiä sopivia testausmenetelmiä.

3 Integrointitestaus

Integrointitestauksessa koostetaan yksikkötestauksen läpäisseitä komponentteja ryppäiksi joiden toimintaa testataan. Perusajatuksena integrointitestauksessa on testata miten komponentit toimivat yhteistyössä toistensa kanssa ja paikantaa mahdollisia virheitä komponenttien välisistä rajapinnoista.

Integrointitestauksessa testataan (yksikkötestauksesta erillisesti) kaikkien komponenttien kaikki komponenttien väliset rajapinnat. Tämä toteutetaan käytännössä siten että ajetaan uudelleen integrointitestattavien komponenttien yksikkötestit. Ne yksikkötestit joissa käytetään tynkiä uusitaan todellisilla luokilla (tai vastaavilla komponenteilla kuten Internetissä sijaitsevilla tietovarastoilla) jos siihen on mahdollisuus.

Minimikriteereitä kattavampi testaus on sallittua resurssien ne salliessa.

3.1 Testausjärjestys

Integrointitestaus uusille komponenteille pyritään tekemään heti kun uusi komponentti on valmis (ts. se on ohjelmoitu, dokumentoitu ja yksikkötestattu kattavuuskriteereiden mukaan ja testistä on olemassa JUnit-luokka). Uuden luokan valmistuessa se integroidaan sopivilta osin jo testattuun ryppääseen. Tällöin testataan rajapinnat vain uuden komponentin osalta, ts. jo tehtyjä testejä ei tarvitse suorittaa uudelleen uusien komponenttien valmistuessa ellei komponentti itsessään vaikuta olennaisilta osin muun ryppään komponenttien toimintaan. Tarkempi testaus tässä mielessä voi tulla kyseeseen esimerkiksi silloin kun osajärjestelmä on jouduttu testaamaan tyngän avulla joka vaikuttaa merkittävästi testattavan ryppään toimintaa. Tällöinkin tavoitteena on se ettei testauksessa käytettävää luokkaa jouduta kirjoittamaan uudelleen vaan riittää esimerkiksi tässä tapauksessa testissä käytettävän tyngän vaihtaminen itse uuteen komponenttiin.

Tavoitteellisesti integrointitestaus pyritään tekemään alhaalta ylös (bottom-up) strategialla. Tällöin siis pyritään aloittamaan testaus mahdollisimman alhaisen tason komponenteista ja edetä integroinnissa korkeamman tason komponentteihin. Tällä pyritään minimoimaan tynkien ohjelmointiin käytetty aika. Jotta strategia olisi mahdollinen tulee projektisuunnitelmassa aikataulutus hoitaa siten että komponenttien toteutus aloitetaan mahdollisimman alhaisen tason komponenteista edeten korkeamman tason komponentteihin (mahdollisuuksien mukaan).

Jos jonkin komponentin integrointitestaus on erityisen suurella prioriteetilla ja komponentti on niin korkealla tasolla että sitä ei voida testata jo valmiiden luokkien kanssa voidaan käyttää ylhäältä alaspäin etenevää testaustapaa. Tässä testaustavassa komponentteja varten ohjelmoidaan sopivat tyngät. Tehokkuussyistä tätä testaustapaa pyritään kuitenkin välttämään.

Järjestelmä ei ole valmis järjestelmätestausta varten ennen kuin integrointitestaus on valmis koko järjestelmän osalta. Integrointitestauksessa ei kuitenkaan pyritä tekemään järjestelmätestauksen piiriin kuuluvia testejä (käyttötapausten testaus) vaan pitäydytään rajapintojen testauksessa.

3.2 Virheet

Mikäli integrointitestauksessa huomataan testaajan korjattavissa oleva virhe pyritään se korjaamaan saman tien. Jos virhe on merkittävä ja integrointitestausta tekevä taho ei ole toteuttanut testattavaa luokkaa eikä osaa korjata sitä omin avuin, otetaan yhteyttä luokan toteuttajaan (kiireestä riippuen joko puhelimitse tai sähköpostitse) ja sovitaan luokan korjaamiseen liittyvistä seikoista.

4 Järjestelmätestaus

Järjestelmätestausta voidaan suorittaa erikseen eri osajärjestelmille, kun osajärjestelmän integrointitestaus on valmis. Testausmenetelmä vastaa yksikkötestauksen black-boxia, eli

järjestelmätestausvaiheessa ei enää oteta kantaa ohjelmiston toteutusmenetelmiin, ja järjestelmää käytetään testauksen aikana vain sen tarjoamien käyttöliittymien kautta. Testaus on käytännössä vaatimusmäärittelyssä määriteltyjen vaatimuksien (sekä toiminnalliset, että ei-toiminnalliset/ laadulliset) toteutumisen tarkistamista, mutta voi sisältää myös järjestelmän laadun arvioimista muista näkökulmista. Testauksen on tapahduttava kohdeympäristössä. Ohjelmisto läpäisee järjestelmätestauksen, jos se toteuttaa ja täyttää onnistuneesti kaikki vaatimusmäärittelyn vaatimukset.

4.1 Toiminnalliset vaatimukset

Tuotteen syöte/tulos -testaus tehdään antamalla järjestelmälle ennalta määrättyjä testisyötteitä ja tarkastelemalla, vastaako saatu transformoitu metadata syötettä semanttisesti. Eri lähdetyyppejä simuloidaan itse tehdyillä tiedostoilla ja myöhemmässä vaiheessa OAI-PMH -tyngillä. Järjestelmä testataan myös todellisella datalla lähteistä, joita tullaan todennäköisesti käyttämään ohjelmiston oikeassa käytössä. Näistä lähteistä saadun datan laajuuden takia testi jätetään pintapuoliseksi. Riittää tarkistaa että keräyksen ja transformaation tuloksena saadaan jotain oikean näköistä transformoitua dataa.

4.2 Ei-toiminnalliset vaatimukset

Alla listatuista vaatimustestauksista voitaneen tinkiä, sikäli kun vaatimusmäärittelyssä ei ole erikseen määrätty jotain aihepiiriä koskevia vaatimuksia.

Yhteensopivuusvaatimukset Varmistetaan järjestelmän yhteensopivuus ympäristön kanssa. Tässä projektissa ympäristö on teoriassa rajattu melko tarkkaan, ja käytetty ohjelmointiympäristö on joustava, joten tarpeellisuus voidaan kyseenalaistaa.

Suorituskykyvaatimukset Varmistetaan että järjestelmä selviää esim. vaatimusmäärittelyssä sen joillekin toiminnoille asetetuista aikavaatimuksista. Lisäksi voidaan suorittaa volyymitestaus, eli kokeillaan järjestelmää valtavilla datamäärillä, ja katsotaan miten se vaikuttaa suorituskykyyn ja myös luotettavuuteen.

Eheys- ja vikasietoisuusvaatimukset Testataan kuinka luotettava järjestelmä on, ja miten hyvin se toipuu poikkeustilanteista.

Käytettävyysvaatimukset Arvioidaan kuinka helppoa järjestelmän käyttäminen on sen tarjoamalla käyttöliittymällä. Tämä teetettäneen asiakkaalla.

Asennustestaus Arvioidaan kuinka helppoa järjestelmän asentaminen ja käyttöönotto on kohdeympäristössä.

4.3 Hyväksymistestaus

Kun järjestelmätestaus on suoritettu muilta osin, järjestelmä annetaan asiakkaan testattavaksi ja arvioitavaksi. Asiakkaan havaitsemia puutteita voidaan korjata aikataulun, prosessimallin, vaatimusmäärittelyn ja asiakkaan kanssa tehtyjen sopimusten puitteissa.

4.4 Järjestelmätestauksen käyttötapaukset

4.4.1 Askel 1

Testimenettely:

- Käynnistetään datankeruun käyttöliittymä.
- Lisätään lähteeksi paikallisella levyllä oleva XML Dublin Core muotoinen tiedosto
- Käynnistetään eräajo

Odotettu tulos: ohjelma transformoi lähdedatan atomilauseiksi.

Testatut vaatimukset:
5.1 Raakadatan muunnos levyllä olevasta XML Dublin Core muotoisesta raakadatatiedostosta QS-muotoon

4.4.2 Askel 2

Testimenettely:

- Käynnistetään datankeruun käyttöliittymä.
- Lisätään lähteeksi paikallisella levyllä oleva QS XML muotoinen tiedosto
- Käynnistetään eräajo

Odotettu tulos: ohjelma lisää tiedostoon käyttäjän otsikkotiedot.

Testatut vaatimukset:
5.2 Otsaketietojen lisääminen levyllä olevaan QS-muotoiseen tiedostoon

5 Työskentelytavat

Testauksessa käytettävät työkalut:

CVS Katso projektin CVS-ohje

JUnit Katso projektin testausohje

Eclipse http://www.eclipse.org/ (tarvittava informaatio selviää CVS-ohjeesta ja oheisesta URL:ista)

Coverlipse http://coverlipse.sourceforge.net/index.php (käyttö ja asennus testausohjeessa).

5.1 Yleistä työskentelytavoista

Testaus pyritään tekemään siten että testit voidaan haluttaessa uusia sellaisinaan. Testaukseen liittyvä koodi, ym. ladataan projektin CVS-järjestelmään kuten muukin projektissa tuotettu materiaali.

On olennaista että suoritetut testit tehdään laitoksen Linux-ympäristössä, jossa ohjelmiston on luvattu toimivan. On asia erikseen missä testiluokat kirjoitetaan, mutta on tärkeää että lopulliset testit tehdään laitoksen Linux-koneilla. Eräs vaihtoehto on ottaa kotikoneelta vapaavalintaisella ohjelmalla SSH-yhteys jollekin laitoksen koneelle (esim. melkki.cs.helsinki.fi) ja ajaa testit tätä kautta. Testejä voidaan tehdä myös suoraan kotikoneella mutta luokkaa ei hyväksytä testatuksi ennen kuin testit on ajettu hyväksytysti laitoksen Linux -ympäristössä.

5.2 Työskentelytavat yksikkötestauksessa

Testauksessa käytettävien työkalujen osalta tulee Eclipsen, JUnitin ja Coverlipsen käytöstä olla jonkinlainen käsitys ennen kuin yksikkötestausta voidaan aloittaa. Perusideana käytännön yksikkötestauksessa on se, että jokaista testattavaa komponenttia varten kirjoitetaan JUnitille erillinen testausluokka. Glass-box -testauksessa testiluokan ajo suoritetaan siten, että Coverlipse laskee samalla myös kattavuudet.

Testiluokat ja tyngät määritetään pakettiin org.qriterium.unit_tests ja tiedostot sijoitetaan testattavan luokan paketin nimiseen alihakemistoon. Ohjelmiston varsinaisten luokkien hierarkia alkaa siis alihakemistosta src/org/qriterium, ja testiluokkien hierarkia alihakemistosta src/org/qriterium/unit_tests.

Testien kattavuutta kannattaa tarkastella kun ekvivalenssiluokista muodostettujen testitapausten avulla löydetyt virheet ovat korjattu. Lausekattavuuden (engl. line/statement coverage) täytyy testattavassa luokassa olla 100%, kun käytetään glass-box testausmenetelmää. Black-boxilla kattavuuslaskentaa ei tarvitse tehdä. Kattavuusluku jää alle 100%:n jos luokassa on rivejä joita ei pystytä suorittamaan. Vajaa luku on hyväksyttävä jos suorittamatta jäävistä riveistä tehdään maininta testiluokan kommentteihin.

Pääsääntöisesti koodin kirjoittaja yksikkötestaa itse oman koodinsa. Jos koodaaja ja testaaja ovat eri henkilö, koodaaja ilmoittaa testaajalle, kun komponentti on valmis testattavaksi, jos ajankohtaa ei ole sovittu etukäteen. Ilmoittaminen tapahtuu ensisijaisesti sähköpostitse. Puhelimitse voidaan toki ilmoittaa että sähköposti on lähetetty. Testaajalle ilmoitetaan testattavat tiedostot ja niiden sijainti CVS:ssä. Jos testaaja löytää pieniä helposti korjattavia virheitä, testaaja korjaa virheet siten että testit menevät läpi, päivittää korjatut tiedostot CVS:ään, ja ilmoittaa komponentin tekijälle korjatut virheet. Suurempien tai

monimutkaisempien virheiden löytyessä testaaja ilmoittaa riittävällä tarkkuudella tiedot virheistä komponentin tekijälle. Testiprosessi toistetaan kun koodaaja on mielestään korjannut virheet.

Käytännössä yksikkötestaukseen sisältyy karkealla tasolla seuraavat vaiheet:

- Testattavan komponentin tuoreimman version noutaminen ja tutkiminen.
- JUnit-testiluokan kirjoittaminen ja kommentoiminen.
- Testien ajaminen Junitilla
- Mahdollinen testiluokan korjaaminen tai ilmoittaminen komponentin tekijälle -> uusi testiajo.
- Komponentin API-dokumentaation muuttaminen, jos siihen ilmeni tarvetta (tai ilmoittaminen komponentin tekijälle) -> uusi testiajo.
- Glass-box -testauksessa tarkistetaan testien kattavuus Coverlipsellä (100% lausekattavuus soveltuvin osin).
- Jos kattavuus ei täyty, päivitetään testiluokkaa siten että testitapauksilla saavutetaan vaadittu kattavuus. Ajetaan JUnit-testit ja tarkistetaan uudelleen kattavuus.
- Komponentin päivittäminen CVS:ään, jos korjattiin virheitä. Komponentin API:n päivittäminen jos siihen on tehty muutoksia.

Kaikkien ryhmän tuottamien komponenttien on läpäistävä yksikkötestaus, ennen kuin niitä voidaan käyttää (seuraavissa testausvaiheissa tai muiden luokkien yhteydessä). Luokka pidetään yksikkötestattuna kun CVS:ään on ladattu luokkaa vastaava testiluokka.

5.3 Testidokumentaatio

Testauksessa syntyvä dokumentaatio sisällytetään wikin testauskirjanpitoon ja JUnitohjelmassa käytettävien testiluokkien kommentteihin. Muita erillisiä dokumentteja testeistä ei tarvitse kirjoittaa.

Testauskirjanpitoon lisätään yksikkötestatun luokan osalta seuraavat kohdat:

- Luokan nimi
- Luokan testaaja
- Luokan tila

Katso testauskirjanpidosta tarkemmat ohjeet kohtien täyttämiselle. Kunkin henkilön tulee varmistua siitä että kaikki tuottamansa luokat yksikkötestataan.

Testauskirjanpito löytyy tämän dokumentin liittenä.

Testiluokat kirjoitetaan ja kommentoidaan projektiryhmän yleisen koodin tyyliohjeen mukaisesti. Testiluokan kommenteissa tulee ilmoittaa jos luokan jotain metodia ei testata jostain erityisen hyvästä syystä johtuen (esimerkiksi valmiin komponentin käyttämätön metodi jota ei kuitenkaan ole poistettu). JUnit-luokkaan ei tarvitse liittää yksikkötestaukseen liittyviä vuokaavioita, tms. Perusajatuksena on se että yksikkötestauksen kattavuuteen liittyväksi tulosmateriaaliksi riittää Coverlipsen avulla toistettavissa olevat rivikattavuuslaskelmat. Jos testiluokka käyttää hyväkseen joitakin muita luokkia tai esimerkiksi tynkiä (stub) ulkoisten osien toiminnallisuuden emuloimiseen, tulee nämä luokat listata testiluokan otsikkotiedoissa. Jos testauksessa käytetään tynkää tulee tästä mainita erillisesti testiluokan kommentoinnissa.

Lopullinen testausdokumentti kirjoitetaan kun koko projekti on järjestelmätestattu eikä se ole yksikkötestaajan vastuulla. Koska testit ovat toistettavissa, voidaan testausdokumenttiin tarvittaessa tuottaa uudelleen testien sisältö.

6 Testausaikataulu

Tiedot testauksen aikataulusta löytyvät projektisuunnitelmasta.

Testauskirjanpito

org.qriterium.connection

Luokka	Testaaja	Kattavuus	Kommentit
AwareOfConnection	Arto	100%	
Connection		-	Interface.
ConsistOfConnection	Arto	100%	
ContactedAtConnection	Arto	100%	
CreatedByConnection	Mikko	100%	
CreatedForConnection	Arto	100%	
CreatedInConnection	Arto	100%	
IdentifiedAsConnection	Arto	100%	
ManagedByConnection	Arto	100%	
PublishedByConnection	Arto	100%	
QConnection		100%	
QConnectionFactory	Arto	96%	Luokasta ei voi luoda olioita -> konstruktori jää testaamatta.
StoredByConnection	Arto	100%	
SupportedByConnection	Arto	100%	
UnsupportedConnectionTypeException	Arto	100%	

org.qriterium.io

Luokka	Testaaja	Kattavuus	Kommentit
ConsolePrinter		-	Pelkkä debug-luokka, ei testata.
DataLeetcher		-	Interface.
FileReader	Mikko	100%	
NullOutputStream		-	Vain yksi void metodi, joka ei tee mitään. Ei testattavissa.
QSPrinter	Timo	97%	Kaksi poikkeusta testaamatta. Näitä poikkeuksia on käytännössä mahdoton saavuttaa, mutta poikkeuskäsittelijää ei voi poistaa.
StatementPrinter		-	Interface.
Storable		-	Interface.

org.qriterium.settings

Luokka	Testaaja	Kattavuus	Kommentit
FileState	Arto	100%	
Loader	Erno	84%	
Source	Erno	100%	
SourceState	Erno	100%	
User	Arto	100%	

org.qriterium.statement

Luokka	Testaaja	Kattavuus	Kommentit
Organization	Arto	100%	
Person	Arto	100%	
QActor	Arto	100%	
QContent	Arto	100%	
QDocument	Arto	100%	
QHeader	Arto, Timo	100%	
QSource	Arto	100%	
QStatement	Arto	100%	
QStatements	Arto	100%	
RawConnection	Arto	100%	

org.qriterium.transformer

Luokka	Testaaja	Kattavuus	Kommentit
CiteSeerTransformer	Mikko	100%	
DPLPTransformer	Ilmari	100%	
QSTransformer	Mikko	100%	
Transformer		-	Abstrakti luokka.

org.qriterium.ui

Luokka	Testaaja	Kattavuus	Kommentit
Boa			Pääohjelmaluokka. Testattu järjestelmätestauksen aikana.

org.qriterium.util

Luokka	Testaaja	Kattavuus	Kommentit
IdFactory	Erno	88%	Saavuttamaton keskeytys jota ei voi simuloida tai poistaa.
LogFormatter	Erno	100%	
LogHandler	Erno	100%	
QConnectionHandler	Arto	74%	Testit eivät kata tiedoston luku-/kirjoitusoikeusongelmia. Rivi 110 on taas mahdoton saavuttaa käytännössä.
QConnectionHandlerCachehandler		88%	Testattu QConnectionHandler -luokan yhteydessä. Saavuttamaton keskeytys jota ei voi simuloida tai poistaa.
QStatementIndex	Ilmari	100%	
QStatementIndexCacheHandler	Timo	100%	Testattu QStatementIndex -luokan yhteydessä.
SimpleNamespaceContext	Arto	100%	
StatusPrinter	Arto	100%	
StringUtilities		89%	Luokasta ei voi luoda olioita -> konstruktori jää testaamatta.
XMLDataElement	Arto	100%	
XMLProperties		-	Sisältää vain XML:n generoinnissa käytettyjä muuttujia.