Testaussuunnitelma

Metaxa

Helsinki 14.12.2005
Ohjelmistotuotantoprojekti
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Kurssi

581260 Ohjelmistotuotantoprojekti (6 ov)

Projektiryhmä

Väinö Ala-Härkönen
Reima Halmetoja
Antti Laitinen
Kalle Pyykkönen
Oskari Saarekas
Tuomas Tanner
Juuso Vanonen

Asiakas

Olli Niinivaara

Johtoryhmä

Juha Taina
Joni Salmi

Kotisivu

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

Versiohistoria

20.09.2005 Versio 1.0
27.09.2005 Versio 2.0
04.10.2005 Versio 3.0
20.10.2005 Versio 4.0
02.11.2005 Versio 5.0
06.11.2005 Versio 6.0
14.11.2005 Versio 7.0
06.12.2005 Versio 8.0
13.12.2005 Versio 9.0
14.12.2005 Versio 10.0 (kirjoitettu puhtaaksi)

Sisältö

1. Johdanto

Tämä dokumentti on testaussuunnitelma Metadatan hallintatyökalu-projektille. Työkalun avulla metadataa voidaan luoda ja koota metadatan ominaisuuksien tarkastelua varten. Työkalu mahdollistaa metadatan harvestoinnin (noutamisen) tällaista palvelua tarjoavalta palvelimelta tai paikallisesta tietovarastosta (cd-rom), metadatan tallentamisen paikallisesti jatkokäsittelyä varten, sekä paikallisesti tallennetun metadatan poimimisen ja poimintatulosten edelleen tallentamisen tiedostoiksi jatkoanalyysia varten.

Testaussuunnitelmassa kartoitetaan se, miten projektissa tuotettavan järjestelmän testaus suoritetaan. Testauksen tavoitteena on osoittaa sekä asiakkaille että projektille, että ohjelmisto täyttää sille asetetut vaatimukset (validointitestaus) 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 (Sommerville: defect testing).

2. Yksikkötestaus

Komponenttien testaus perustuu koodin lisäksi myös komponentille asetettuihin vaatimuksiin. Kaikille testattaville komponenteille tulisi 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 kommenttehin tulee sisältyä selvitys sen toiminnasta, jotta palvelupohjaista testausta voidaan suorittaa (jotta sen voisi suorittaa joku muukin kuin luokan kirjoittaja). Jos joitakin metodeja on testattu luokan kehitysvaiheessa tulee nämä metodit kuitenkin testatata 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

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 parametreillä. 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 luokan rajoilla olevat arvot,

sekä jokin arvo luokan arvoalueen "keskeltä".

Luokan kenttien lukeminen ja asettaminen on yleensä kapseloinnin hengen mukaisesti toteutettu get- ja set-metodeilla, jolloin testaus tulee tehtyä metoditestauksen lomassa. Jos luokassa on edellämainituilla 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 edellämainitulla 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, ja sen voi korvata kompositiolla, joten sitä kannattaa välttää.

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 stubeilla. Näitä muutaman tiivissä yhteistyössä olevan luokan kokoelmia kutsutaan ryppäiksi. Integrointitestaus koskee ryppäiden yhteistoiminnan testaamista.

Black-box (mustalaatikkotestaus, 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 annetaan 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. API:n perusteella voidaan myös päätellä metodin palauttama arvo. Jos API:ssa 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 API:ssa. Muita poikkeuksellisia parametreja voidaan testata harkinnanvaraisesti. Tilanteissa joissa API-kuvaus on virheellinen tai puutteellinen tulee se korjata.

Glass-box (lasilaatikkotestaus, toteutuspohjainen testaus)

Täydentää black-box -testausta ottamalla huomioon testattavan komponentin lähdekoodin ja tilakaavion. Metodin testiparametrien ekvivalenssiluokat johdetaan palveluvaatimusten lisäksi metodin lähdekoodista. Ekvivalenssluokista 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 toiminnalisuuden 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 stubeja 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 komponettien 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

Integroinitestaus 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 komponettien 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 tyngan vaihtaminen itse uuteen komponenttiin.

Tavoitteellisesti integrointitestaus pyritetään 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. Lisäksi suunnitelman mukaan (ainakin tällä hetkellä) komponettien toteutusjärjestys tukee tätä integrointitestaustapaa. Jotta strategia olisi mahdollinen tulee projektisuunnelmassa 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 (I, II ja III tässä järjestyksessä), 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

Toiminnallisten vaatimusten testaamiseksi käytetään Extended Use Case Test (EUCT) -menetelmää, joka perustuu laajennettuihin järjestelmän käyttötapauksiin. Tavallinen

käyttötapaus määrittelee toimintatavat, sidosryhmät ja syöte/tulostiedot. Laajennetussa käyttötapauksessa sidosryhmien antamat syötteet ja niiden saamat tulokset selvitetään arvoalueineen, ja syötteille selvitetään vastaava tulospari. Lisäksi määritetään käyttötapauksien riippuvuussuhteet.

Kaikkien vaatimusten pitää sisältyä ainakin yhteen käyttötapaukseen. Testaus on valmis kun kaikista laajennetuista käyttötapauksista on muodostettu ns. päätöstaulut, ja niiden riveille on tehty vähintään true/false -testit.

Raakadatan keruu ja transformointi -osajärjestelmän syöte/tulos -testaus tehdään antamalla järjestelmälle ennalta määrättyjä testisyötteitä ja tarkastelemalla, vastaako saatu transformoitu metadata syötettä semanttisesti. Eri lähdetyyppejä simuloidaan itse tehdyillä tiedostoilla ja OAI-PMH -stubeilla. 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.

Kahteen muuhun osajärjestelmään sovelletaan samanlaisia testaustekniikoita siltä osin kuin mahdollista. Olennaista testidatan suunnittelussa kaikkien järjestelmien osalta on datan sopiva määrä. Esim. integrointivaiheessa tarvitaan sellainen syöte, josta ohjelman on mahdollista luoda verkkoja. Ohjelman antama tuloste tietylle testidatalle pitää pystyä päättelemään vaatimusmäärittelyn ja ohjelmiston suunnittelun pohjalta, jolloin testiajon antama syöte voidaan havaita oikean- tai vääränlaiseksi välittömästi. Testidatan ei siis kannata olla liian laaja, jolloin oikean tulosteen määrittäminen käy hankalammaksi.

Selaus ja poiminta -vaiheen graafiset tulosteet tarkistetaan oikeiksi vertaamalla vastaavaan tiedostotulosteeseen.

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

Askel 1

Testimenettely:

- Käynnistetään datankeruun ja integraation käyttöliittymä
- Lisätään ohjelmaan peräkkäin neljä uutta lähdettä - yksi kutakin seuraavista tyypeistä: XML DC, oai_citeseer, nimi-pikaformaatti, dokumentti-pikaformaatti
- Jokaisesta lähteestä spesifioidaan tietolähteen tyyppi, formaatti ja tiedoston nimi (tiedostossa oltava sisällä formaatin mukaista validia testidataa)
- Suoritetaan update kaikille tietolähteille
- Odotettu tulos: ohjelma tallentaa kustakin tiedostosta .source-asetustiedoston ja kopion tiedoston sisältämästä datasta ohjelman data/-hakemistoon sekä transformoi tiedostojen sisällöt tallettaen ne atomilausetietokantaan

Testatut vaatimukset:

- K1.1 Valmiin metadatan tuonti
- K1.1.5 oai_citeseer-muotoisten tiedostojen tuonti
- K1.1.6 XML Dublin Core-muotoisten tiedostojen tuonti
- K1.2 Itse tehdyn metadatan tuonti
- K1.2.1 Pikaformaatti-tiedostojen tuonti
- K1.3 Metadatan säilytys
- K1.3.1 Raakadatan säilytys
- K1.3.2 Atomilauseiden säilytys

Askel 2

Testimenettely:

- Lisätään ohjelmaan OAI-PMH -tyyppinen lähde: lähteestä annetaan tietolähteen tyyppi oaipmh, formaattina DCXML ja URLina validi repositorion osoite.
- Suoritetaan update tälle tietolähteelle.
- Odotettu tulos: lähteestä tallentuu .source-asetustiedosto sekä kopio noudetuista tietueista data-hakemistoon sekä transformoidut atomilauseet atomilausekantaan

Testatut vaatimukset:

- K1.1.1 XML Dublin Core harvestointi

Askel 3

Testimenettely:

- Lisätään jonkin tiedostolähteistä alkuperäiseen tiedostoon vähintään yksi uusi validi tietue ja suoritetaan ko. tietolähteelle update
- Odotettu tulos: lisätty tietue lisätään atomilausekantaan

Testatut vaatimukset:

- K1.4 Lähteiden päivitys

Askel 4

Testimenettely:

- Otetaan data-hakemistosta noudetun OAI-PMH-lähteen data ja kopioidaan se toiseen paikkaan
- Lisätään kopioitu data uutena tiedostotyyppisenä lähteenä - tietolähteen tyyppi file, formaatti DCXML
- Suoritetaan update tälle tietolähteelle
- Odotettu tulos: samat tietueet ilmestyvät uuden lähteen alle atomilausetietokantaan

Testatut vaatimukset:

- K1.5 Metadatan siirto
- K1.5.1 Raakadatan siirto

Askel 5

Testimenettely:

- Integroidaan yksi transformoiduista lähteistä uuteen resurssiverkkoon
- Integroidaan toinen transformoitu lähde samaan resurssiverkkoon
- Integroidaan kolmas transformoitu lähde toiseen uuteen resurssiverkkoon
- Odotettu tulos: ensimmäisessä verkossa on kahden ensimmäisen lähteen ja toisessa verkossa kolmannen lähteen tietueet integroituina

Testatut vaatimukset:

- K2.1 Resurssiverkon integrointi
- K2.3 Integraation toistettavuus
- K2.2.1 Atomilauseiden valinta
- K2.3.1 Useat resurssiverkot

Askel 6

Testimenettely:

- Käynnistetään selauksen käyttöliittymä
- Valitaan resurssiverkko jossa on resursseja integroituna
- Tehdään itse tai muokataan valmiista hakupohjasta haku, jossa haetaan taulusta Resource resursseja rajatuilla hakuehdoilla - esimerkiksi kaikki joiden id alkaa A:lla - ja ResourceRelation-taulusta näihin liittyvät yhteydet (tähän kannattaa keksiä joku valmis hakuesimerkki ettei testatessa tarvitse aina miettiä uudestaan)
- Suoritetaan haku, tuloksena pitäisi käyttöliittymässä näkyä resursseja mikäli hakuehdoilla niitä löytyy
- Tallennetaan haun tulos Pajek-verkoksi ja CSV-listaksi, tuloksena pitäisi olla vastaavat tiedostot joissa näkyy hakutulos verkkona ja listana (listamuodossa kaksi erillistä tiedostoa joista toisessa listataan resurssit ja toisessa yhteydet)

Testatut vaatimukset:

- K3.2 Resurssien selaus
- K3.2.5 Hakutuloksen avaaminen selausikkunaan
- K3.1.1 Resurssien hakeminen ominaisuuksien perusteella

- K3.1.3 Hakuehtojen muokkaus käsin
- K3.4.3 Yhteysverkon tallennus Pajek-muodossa
- K3.2.3 Hakutulos tallennus

Askel 7

Testimenettely:

- Laajennetaan edellistä hakua niin, että määritellään että haettujen resurssien tulee löytyä vähintään n kertaa yhteystaulusta (ts. yhteyksien määrä >n, missä n on testiaineistolle sopiva määrä, esimerkiksi 1 jos yhteyksiä on niukanlaisesti) ja suoritetaan haku
- Tuloksena pitäisi olla hakutulos jossa on vähemmän resursseja kuin edellisessä

Testatut vaatimukset:

- K3.1.2 Resurssien hakeminen yhteyksien perusteella

Askel 8

Testimenettely:

- Laajennetaan edellistä hakua niin, että resurssien välisten yhteyksien pitää olla tietyntyyppisiä (role = joku testiaineistolle sopiva arvo), suoritetaan haku
- Talletetaan tulos Pajek-verkkona

Testatut vaatimukset:

- U4 Tietyn tyyppisten yhteyksien tallennus

Askel 9

Testimenettely:

- Muutetaan edellistä hakua niin, että yhtä tai useampaa tuloksessa olevista

resursseista ei otetakaan mukaan (esimerkiksi id ei saa päättyä K:hon tms.)
- Suoritetaan haku, tuloksena pitäisi olla jälleen vähemmän resursseja hakutuloksessa

Testatut vaatimukset:
- U2 Resurssien poisto selauksesta

Askel 10

Testimenettely:
- Muutetaan edellinen haku delete-lauseeksi, suoritetaan
- Tuloksena tietokannasta pitäisi hävitä edellisen haun tuloksena olleet resurssit

Testatut vaatimukset:
- K3.2.7 Resurssiverkon puhdistus

5. Työskentelytavat

Testauksessa käytettävät työkalut:

- "CVS": Katso projektin CVS-ohje
- "JUnit": Katso projektin JUnit-ohje
- "Eclipse": http://www.eclipse.org/ (tarvittava informaatio selviää CVS-ohjeesta ja oheisesta URL:ista)
- "Coverlipse": http://coverlipse.sourceforge.net/index.php (käyttö ja asennus yksinkertaista - selviävät tuolta osoitteesta). Kyseessä on siis plugin Eclipseen.

Ohjeisiin kannattaa tutustua ylläolevassa järjestyksessä.

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 tai melkinkari.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 stubit määritetään samannimiseen pakettiin kuin testattava luokka, mutta tiedostot sijoitetaan eri hakemistoon. Ohjelmiston varsinaisten luokkien hierarkia alkaa alihakemistosta src, ja testiluokkien hierarkia alihakemistosta test. JUnit-FAQ: parallel directory structure (http://junit.sourceforge.net/doc/faq/faq.htm#organize_1). Eli CVS:n juureen tehdään koodia varten nuo kaksi alihakemistoa "test" ja "src", joihin lähde- ja testikooditiedostot sijoitetaan pakkaushierarkian mukaisiin alihakemistoihin.

Jokaiseen testiluokkia sisältävään hakemistoon tehdään myös luokka UnitTests, jossa kootaan yksi TestSuite samassa hakemistossa olevista testiluokista, sekä luokka SubTests, jossa kootaan yksi TestSuite saman hakemiston ja alihakemistojen SubTests-luokista. Näin kaikki ohjelmistoon liittyvät JUnit-yksikkötestit voidaan ajaa kerralla ajamalla ylimmän SubTests-luokan JUnitissa.

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 puhelimitse. Toisisijainen ilmoituskanava on sähköposti. 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 JUnit-ohjelmassa käytettävien testiluokkien kommentteihin. Muita erillisiä dokumentteja testeistä ei tarvitse kirjoittaa.

Testauskirjanpitoon lisätään yksikkötestatun luokan osalta seuraavat kohdat:

- Luokan nimi
- Luokan kirjoittaja
- Luokan testaaja
- Testausmenetelmä
- Tila
- Viimeisin tilan muutos

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

kannattaa luokka lisätä heti testauskirjanpitoon kun se on tullut yksikkötestausvaiheeseen.

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 toisetettavissa olevat rivikattavuuslaskelmat. Jos testiluokka käyttää hyväkseen joitakin muita luokkia tai esimerkiksi tynkiä (stub) ulkoisten osien toiminnallisuuden emuloimiseen, tulee nämä luokat listata testiluokan "c-tyylisessä headerissa" (KTS. koodin tyyliohje). 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.

Integrointi- ja järjestelmätestaus on puolestaan siitä vastaavien henkilöiden murhe.

6. Testausaikataulu

Tiedot testauksen aikataulusta löytyvät projektisuunnitelmasta.