jamk.fi

Testauksen automatisoinnin kehittäminen

Kari Leskinen

Opinnäytetyö
Marraskuu 2015

Tietojenkäsittely
Luonnontieteiden ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Leskinen, Kari	Opinnäytetyö	16.11.2015
	Sivumäärä 43	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Testauksen automatisoinnin kehittäminen

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Työn ohjaaja(t)
Niko Kiviaho

Toimeksiantaja(t)
Musicinfo Finland Oy

Tiivistelmä

Opinnäytetyön tutkimuksen tavoitteena oli kehittää Music Info Finland Oy:n nykyistä testausautomaatio järjestelmää integraatio- ja järjestelmätestaustasoilla. Tutkimus toteutettiin kehittämistutkimuksena, jossa kartoitettiin toimeksiantajan testauksen nykyinen tila, etsittiin kehittämiskohteet sekä tutkittiin ja testattiin työkaluvaihtoehtoja.

Näiden tutkimusten tuloksien pohjalta toimeksiantajalle toteutettiin toimiva integraatio- ja järjestelmätestausautomaatio jatkuvan integroinnin mahdollistavaksi ja sitä käyttäväksi järjestelmäksi, jota voidaan tarvittaessa laajentaa kattamaan myös muut ohjelmistotestauksen tasot. Integraatiotestauksen työkaluksi valittiin Node.js ja Mocha pohjainen Chakram.js, joka on REST ja JSON-rajapintojen testaustyökalu.

Integrointitestaus suoritetaan päivittäin ajettavalla ajastuksella. Käyttöliittymätestaukseen valittiin työkaluksi Selenium pohjainen ja Python kielinen Splinter testaustyökalu. Käyttöliittymätestaus suoritetaan versionhallinnassa havaittujen muutosten perusteella. Valitut työkalut olivat ilmaisia avoimen lähdekoodin ohjelmia. Järjestelmään on helposti kirjoitettavissa uusia testitapauksia, jolloin myös ohjelmointiin perehtymättömät testaajat voivat ylläpitää ja käyttää järjestelmää.

Tutkimuksen tuloksena Music.infon ohjelmistotestausta kehitetään luomalla testausautomaatio integraatio- ja käyttöliittymätestaustasoilla Jenkins -testipalvelin ympäristöön käyttäen apuna Git versionhallintaa. Tutkimuksen tuloksia voidaan soveltaa pienten ja keskisuurten ohjelmistoalan yritysten ohjelmistotestaus tarpeisiin.

Avainsanat (asiasanat)
Ohjelmistotestaus, integraatiotestaus, jatkuva integrointi, järjestelmätestaus, testausautomaatio, Git, Jenkins

Muut tiedot

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author(s)	Type of publication	Date
Leskinen, Kari	Bachelor's thesis	16.11.2015
		Language of publication: Finnish
	Number of pages 43	Permission for web publication: x

Title of publication

Development of an automated software testing system

Degree programme

Business Information Systems

Tutor(s)

Kiviaho, Niko

Assigned by

MusicInfo Finland Oy

Abstract

The goal of the thesis was to research and develop Music Info Finland´s current software testing platform by generating an automated testing system for integration and system testing levels. The research included mapping the enterprise's current state of software testing, identifying systems in need of updating and valuating software testing tools to improve and develop the system. Based on the conclusions of this research an automated integration and user interface testing system was built which benefited from continuous integration and which could be expanded to also cover other software testing levels. Chakram.js was chosen for integration testing to test REST and JSON application programming interfaces by running a timed test every day. For user interface testing Python language based Splinter was chosen which ran every time if a change was detected by the source code management. The chosen testing tools are free and open source. The system could be easily scripted by testers not so fluent in programming languages because of the Python and JavaScript languages the test cases used.

The answer for the study´s research problem was: "Music Info Finland´s software testing system could be developed by generating an automated integration and user interface testing system by using Jenkins continuous integration platform and Git version control." These research conclusions could be applied to cover the needs of a small to a medium size enterprise in the field of software development.

Keywords/tags (subjects)

Software testing, integration, system testing, continuous integration, Git, Jenkins

Miscellaneous

Sisältö

Kuviot

Määritelmät ja termistö

Staattinen testaus

Ohjelmakoodin läpikäymistä ilman, että sitä suoritetaan.

Dynaaminen testaus

Testaustapa, jossa suoritetaan ohjelmakoodia.

Funktionaalinen testaus

Käydään läpi ohjelman toiminnalliset osat kuten esim. valintaruudut, ja tekstikentät.

Ei-funktionaalinen testaus

Käydään läpi ohjelman ei-toiminnallisia osia kuten turvallisuus tai nopeus.

Virhe

Ihmisen aiheuttama poikkeama, joka aiheuttaa vian ohjelmistoon, sen lähdekoodiin tai dokumentaatioon.

Vika

Ohjelmistosta tai sen dokumentaatiosta löytyvä poikkeama, joka voi estää ohjelmaa toimimasta oikein aiheuttaen häiriön.

Häiriö

Tilanne, jossa ohjelma toimii eri tavoin kuin sen pitäisi.

Skripti

Helposti omaksuttavaa ja selkolukuista ohjelmakoodia.

Verifiointi

Varmistaa, että tuote vastaa alkuperäisiä suunnitteluvaatimuksia. Vastaa kysymykseen: ”Tehdäänkö tuotetta oikein?”.

Validointi

Varmistaa, että tuote vastaa asiakkaan vaatimuksia. Vastaa kysymykseen: ”Tehdäänkö oikeaa tuotetta?”.

Build, käännös

Toiminta, jolla kerätään kasaan eli rakennetaan ohjelma testausta varten.

Trigger

Toiminnon laukaisin. Laukaisimiksi voidaan määrittää esimerkiksi tapahtuma, joka on edellytys seuraavalle askeleelle.

Build step

Rakennusvaiheeseen määriteltävä askel, esimerkiksi skriptin suoritus.

API

Rajapinta, josta palvelin voi tarjota resursseja ja jakaa tietoa.

JSON

Helppolukuinen tiedostomuoto, jota käytetään tiedonvälitykseen.

Polling

Versionhallinnan tekemä tiedustelu, joka etsii muutoksia versioista.

SCM

Lähdekoodien hallintamanageri, versionhallinta.

Item

Jenkins-järjestelmän käyttämä nimitys projekteille.

DevOps

Toimintamalli, jolla pyritään automatisoimaan sovellusten toimitus ja infra-struktuurin muutokset.

1 Johdanto

Opinnäytetyön aiheena on tutkia ja kehittää Music.info Finland Oy:n ohjelmistotestauksen automatisointia integraatio- ja järjestelmätestaustasoilla. Music.info Finland Oy on vuonna 2012 perustettu ohjelmistoalan yritys joka toimii Jyväskylässä. Heidän päätuotteenaan on musicinfo.io-verkkopalvelu, josta käyttäjät voivat hakea kaiken tiedon kerralla artistiin, albumiin tai musiikkikappaleeseen liittyen, kuten sanoitukset, nuotit, hintatiedot jne. Hakiessani yritykseen työharjoitteluun, opinnäytetyön ohjaajani ehdotti mahdollisuutta tehdä opinnäytetyö aiheena testauksen automatisointi. Music.infon ohjelmistotestauksen automatisoinnin kehittämisen tavoitteena on parantaa yrityksen ohjelmistojen, erityisesti musicinfo.io-palvelun laatua, tehostaa niiden kehitystä, sekä aikaistaa julkaisua jolloin saadaan nopeammin loppukäyttäjien palautetta palveluista.

2 Tutkimusasetelma

2.1 Toimeksiantaja

Opinnäytetyön toimeksiantajana toimii vuonna 2012 perustettu startup-yritys Music.Info Finland Oy (jatkossa Musicinfo tai toimeksiantaja). Yrityksen päätuotteena toimii musiikkipalvelu musicinfo.io, josta käyttäjät voivat etsiä haluamastaan artistista, kappaleesta tai albumista kaiken tiedon yhdellä hakukerralla. Musicinfo on voittanut vuonna 2013 San Francisco MusicTech Summitissa parhaan startupin palkinnon. Venäläinen Russian Startup Rating on antanut sille samana vuonna BBB-luokituksen, joka on heidän 4. ylin luokituksensa. Keski-Suomen kauppakamarin järjestämässä Kasvu Open -kisassa Musicinfo palkittiin vuoden 2013 startup sarjan voittajana. Musicinfon sijaintipaikkana on Jyväskylä ja se työllistää 8 henkilöä.(VRT Finland Oy ja Music.Info Finland Oy kolmannen Kasvu Open -kilpailun voittajat, 2013).

2.2 Tutkimusmenetelmä

Tutkimusmenetelmänä toimii kehittämistutkimus. Testaustyökalut ja testaustekniikat valitaan ensin kartoittamalla yrityksen nykyisin käytössä olevat työkalut ja tekniikat. Niille etsitään ja valitaan vaihtoehtoja. Vaihtoehtoja vertailemalla pyritään löytämään yrityksen tarpeisiin parhaiten sopivat työkalut ja tekniikat. Näistä vaihtoehdoista parhaiten Musicinfon tarpeisiin sopivat valitaan toimeksiantajan käyttöön, ja niillä toteutetaan automaattinen integrointi- ja käyttöliittymätestaus.

Kehittämistyössä kehitetään jotain asiaa tai toimintaa. Kehittämistyössä mennään kohti parempaa, ja siinä on kyse tietoisesta toiminnasta. Se edellyttää nykytilan kartoituksen, vaihtoehtojen etsinnän ja arvottamisen, tavoitteiden määrittämisen ja keinojen valinnan tavoitteiden pääsemiseksi (Kananen 2010, 159).

2.3 Tutkimuskysymykset

Tutkimuskysymys:

- Miten kehitetään Music.infon ohjelmistotestausta?

Alakysymykset:

- Miten kehitetään yrityksen integraatiotestausta?
- Miten kehitetään yrityksen käyttöliittymätestausta?
- Miten kehitetään yrityksen testausautomaatiota edellä mainittujen testaustasojen osalta?

2.4 Tutkimuksen toteutus ja tavoitteet

Tutkimuksen tavoitteena on kehittää yrityksen ohjelmistotestauksen automaatiota integraation ja käyttöliittymien osalta sekä samalla kuvata yrityksen testauksen prosessi. Konkreettisena tuotoksena yritykselle valitaan testaustyökalut, joilla testausautomaatio voidaan suorittaa ja joiden avulla voidaan luoda

toimiva testausautomaatio integraatio- ja käyttöliittymätestaukseen. Ohjelmistotestauksen kehitys auttaa parantamaan yrityksen ohjelmiston laatua ja varmistamaan, että ohjelmat toimivat niin kuin on määrittelyvaiheessa suunniteltu.

2.5 Rajaus

Opinnäytetyön tutkimus rajataan käsittämään vain testausautomaatiota, integraatiotestausta sekä käyttöliittymätestausta. Muita testausalueita, työkaluja tai prosesseja ei oteta työssä huomioon. Näitä ovat esimerkiksi yksikkötestaus, joka jätetään yrityksen ohjelmoijien vastuulle, sekä hyväksymistestaus musicinfo.io-webpalvelun nykyisen kehitystilan vuoksi. Myöskään staattisia testejä ei tulla tutkimaan. Opinnäytetyössä käsitellään vain funktionaalista puolta, ja ei-funktionaaliseen testaukseen, kuten suorituskykyyn tai turvallisuuteen, liittyvät testauksen tasot rajataan ulos. Poisrajatut testaustasot kuitenkin esitellään ohjelmistotestauksen taustateoriaosuudessa, sillä ne kuuluvat oleellisena osana testauksen teorian kokonaisvaltaisen ymmärryksen saavuttamiseen.

2.6 Aiempi tutkimus

Testausautomaatiosta aiempaa tutkimusta ovat tehneet:

- Web-sovellusten testauksen automatisointi, Jäsberg Jarkko, http://urn.fi/URN:NBN:fi:amk-201104285055.
- Ohjelmistotestauksen automatisointi, Sacklén Timo, https://www.theseus.fi/handle/10024/10190.
- Jatkuva integraatio ohjelmistokehityksessä, Vesa Vilkman, https://jyx.jyu.fi/dspace/handle/123456789/24657.

3 Ohjelmistotestaus

Ohjelmistotestaus on työtä jolla varmistetaan toteutettavan ohjelmistotuotteen olevan toivotun kaltainen ja se, että kaikki valmiit ominaisuudet toimivat tarkoituksenmukaisesti ja ne vastaavat asiakkaan tarpeita. Kasurinen(2013, 10) tiivistää testauksen määritelmän seuraavasti:

”Varmistetaan, että tehdään oikeaa tuotetta ja että tuote on tehty oikein.”

Testaus on jatkuvaa vertailua, jolla tarkastetaan, onko se, mitä on saatu aikaiseksi vastaavanlaista kuin oli tarkoitus ja jolla tunnistetaan kohdat, joissa tuotos poikkeaa suunnitellusta (Kasurinen 2013, 10). Lewisin mukaan ohjelmistotestaus on suosittu riskienhallintastrategia, jolla verifioidaan, että funktionaaliset vaatimukset on saavutettu (Lewis 2000, 7). Verifikaatiolla todistetaan että tuote vastaa sille kehityksen alussa ja sen aikana määritellyt vaatimukset. Testausvaihe päättyy, kun kokonaisuudessa ei enää ole merkittäviä virheitä tai kun testattavana oleva ohjelmisto on ainakin niin toimintakuntoinen, että se toteuttaa kaikki määrittelyvaiheessa sille asetetut odotukset. Kun testaus on saatu loppuun, tuote on valmis käyttöönottoon. (Kasurinen 2013, 13.)

Testaus jaetaan neljään eri päätasoon jotka ovat:

- Yksikkötestaus-
- Integraatiotestaus-
- Järjestelmätestaus-
- Hyväksymistestaus

Integraatio- sekä järjestelmätestaus (johon käyttöliittymätestaus pohjautuu) käsitellään omissa alaluvuissaan, koska ne liittyvät oleellisemmin aiheeseen tutkimusrajauksen vuoksi. Seuraavaksi käydään kuitenkin lyhyesti läpi yksikkö- ja hyväksymistestauksen teoriaa, sillä se liittyy oleellisesti ohjelmistotestauksen vaatiman taustateorian kokonaiskuvan saavuttamiseen.

Yksikkötestauksella käsitetään yksittäisten yksiköiden eli moduulien testausta. Yleensä tämän testauksen suorittavat yksiköstä vastuussa olevat ohjelmoijat, jotka varmistavat että yksikkö toteuttaa sille asetetut funktionaaliset ominaisuudet ja että se vastaa syötteisiin määritysten mukaisesti. Yksikkötestaus on testaustoimenpiteistä tavallisin, ja sitä käytetään kaikissa ohjelmisto-organisaatioissa. Yksikkötestauksella tarkoitetaan testausta, jossa moduulin, funktion tai olion toimintaa tarkastellaan heti toteutuksen yhteydessä. Sillä varmistetaan, että juuri toteutettu toiminto tai muunnos toimii ainakin periaatteessa. Testausta varten voidaan esimerkiksi ohjelmoida funktio- tai oliokutsuja, joihin

ohjelman tulee vastata oikein tai palauttaa oikea arvo annetun syötteen perusteella. (Kasurinen 2013, 51 – 54.)

Hyväksymistestaus on testauksen viimeinen työvaihe. Sen tavoitteena on validoida valmiin ohjelmiston riittävän korkea laatu ja näyttää, että ohjelmisto täyttää vaatimusmäärittelyn sille asettamat vaatimukset. Hyväksymistestauksessa järjestelmä tarkastetaan virallisesti asiakkaan toimesta, minkä jälkeen ohjelmisto siirtyy asiakkaan omaisuudeksi. Hyväksymistestauksessa järjestelmää käytetään kohdeympäristössä eikä sille rakennetussa testausympäristössä, kuten järjestelmätestauksessa on tapana. (Kasurinen 2013, 57 – 58.) Hyväksymistestaus on rajattu opinnäytetyön aiheen ulkopuolelle musicinfo.io-webpalvelun tämän hetkisen kehitystilan vuoksi.

Testauksen mallit

Testaukseen liittyvät mallit, jotka kuvaavat suhdetta testauksen ohjelmistonkehityksen prosessin kanssa. Mallit kuvaavat myös prosessin eri vaiheita ja niiden suoritusjärjestyksiä. Valitulla mallilla on suuri vaikutus siihen, miten testaus käytännössä toteutetaan. Malli määrittää sen, milloin, mitä ja miten testataan. Valinnat vaikuttavat regressioon, ja niiden mukaan päätetään, mitä tekniikoita testauksessa tullaan käyttämään.

Testaukseen liittyvät eri prosessimallit ovat:

- Vesiputousmalli
- Inkrementaalimalli
- Iteratiivinen malli
- Spiraalimalli
- RAD (Rapid Application Development)
- Agilet (Ketterät) mallit
- V-malli

Nykyisin käytetyin malli on Agile, joka on korvannut vanhemman vesiputousmallin. Vesiputousmallissa testaus suoritetaan aina kehityksen jälkeen viimeisenä vaiheena. Tästä johtuen virheet huomataan vasta, kun niiden korjaaminen on kallista. Ketterässä mallissa jokaisen ns. sprintin jälkeen saadaan aikaiseksi asiakkaalle esiteltävä ominaisuus, jolloin ominaisuus voidaan samalla heti validoida. V-malli on tehty nimenomaisesti verifiointiin ja validointiin tehty, mutta se käy testauksen malliksi myös kaikkiin muihin malleihin. (Turpeinen 2015). Siinä kehittäjän ja testaajan tuotteen elinkaari on nidottu yhteen, jolloin testaus suoritetaan aina jokaisessa kehitysvaiheessa. Inkrementaalinen, RAD (Rapid Application Development), iteratiivinen ja spiraalimalli perustuvat myös kaikki asiakkaan luomiin tarpeisiin ja vaatimuksiin. (What are the software development models, n.d.)

3.1 Testauksen laatu

Testauksen kattavuus on yksi laadun mittareista. Ohjelman täydellinen testaus riippumatta sen koosta on kuitenkin käytännössä mahdotonta.

```
i=0;
do {
  switch (A) {
    case 1: ....
    case 2: ....
    case 3: ....
  }
  i++;
} while (i < 20);
```

Kuvio 1. Esimerkki täydellisen testauksen mahdottomuudesta (Kautto 1996).

Kuvio 1. kaikkien mahdollisten ohjelmapolkujen lukumäärä on 3^20, kun edellytetään, että muuttuja A saa arvoja 1 -3 while-silmukassa. Tämän ohjelman täydellinen, kaikki ohjelmapolut kattava, testaaminen vaatisi n.3,5 miljardia testitapausta. Nykyaikaiset monimutkaiset ohjelmat koostuvat vielä suuremmista palasista ja niitä on paljon enemmän. Riittävän kattavan testialueen valinta onkin yhtä luovaa suunnittelun kanssa (Kautto 1996). Testauksen kattavuuteen on kehitetty eri testausmenetelmiä, kuten mustalaatikko-menetelmät EP ja BVA, joita avataan syvemmin opinnäytetyön järjestelmätestausta käsittelevässä luvussa.

Toinen tapa parantaa testauksen laatua on vähentää testaajien riippuvuutta ohjelmakoodin suhteen. Riippumattomuus vähentää ennakkoasenteita ja tehostaa virheiden ja vikojen löytymistä. Riippumattomuuden eri tasot alhaisimmasta suurimpaan ovat (What is independent testing? It's benefits and risks, n.d):

1. Ohjelmoija testaa itse.
2. Testit suorittaa toinen ohjelmoija samasta tiimistä tai projektista.
3. Itsenäinen testaaja.
4. Testaaja on täysin eri organisaatiosta tai testaus on täysin ulkoistettu.

Erillinen testiryhmä löytää enemmän ja erilaisia virheitä kuin testaaja, joka toimii ohjelmointiryhmässä tai on ammatiltaan ohjelmoija. Itsenäisellä testaajalla on oma kuvansa ja oletuksensa testauksesta, mikä auttaa löytämään piileviä virheitä ja ongelmia. Itsenäinen testaaja uskaltaa myös helpommin ja rehellisemmin raportoida löydöksistään ilman pelkoa siitä, että jonkun tekemä työ vaikuttaisi huonolle ja antaisi väärän kuvan yksittäisen työntekijän työn laadusta. (What is independent testing? It's benefits and risks, n.d.)

Itsenäiseen testaukseen liittyy kuitenkin myös omat riskinsä. On mahdollista että testausryhmä joutuu eristäytyneeksi muusta organisaatiosta, mikä huonontaa tämän kuvaa laajemmista laatuvaatimuksista ja yrityksen tavoitteista. Itsenäisyys voi myös aiheuttaa omat kommunikaatio-ongelmansa. Työntekijöille voi syntyä tunne siitä, että itse työntekijää kohtaan hyökätään, vaikka näin ei välttämättä olisikaan. Itsenäinen ryhmä voi myös aiheuttaa pullonkaulan ja viivästyksiä ohjelmistokehitykseen, ja jotkin ohjelmoijat voivat luopua laadukkaasta koodista sillä verukkeella, että ne joka tapauksessa tullaan testaamaan, jolloin heidän ei tarvitse alusta alkaen tehdä laadukasta koodia. (What is independent testing? It's benefits and risks, n.d.)

3.2 Testitapaukset

Testitapaus on lajitelma olosuhteita tai muuttujia, joiden perusteella testaaja päättelee, onko järjestelmä riittävän hyvä tyydyttämään sille määritellyt vaatimukset. Testitapausten luominen, kehittäminen ja ylläpito voivat prosesseina

auttaa löytämään virheitä ohjelman suunnittelussa. Hyvässä testitapauksessa testataan yhtä asiaa kerrallaan. Siinä ei pitäisi olla päällekkäisyyksiä, eikä sen pitäisi olla turhan monimutkainen. Testitapausten pitäisi pilkkoa testattava ominaisuus tarpeeksi pieniin osiin, ja niiden tulisi käydä läpi kaikki positiiviset ja negatiiviset skenaariot. Kielen tulisi olla käskevää ja ohjeellista: ”Tee näin” -tyyppistä kieltä. Testattavien kohteiden nimien tulisi myös olla johdonmukaisia ja tarkkoja esimerkiksi lomakkeiden komponenttien, tekstikenttien ja painonappien kohdalla, jotta vältytään sekaannuksilta eikä jouduttaisi varsinaisen testauksen aikana arvailemaan, mitä komponenttia ollaan testaamassa. Lisäksi testitapausten tulisi olla sellaisia, että niitä voidaan toistaa loputtomasti uudelleen. (Test case fundamentals, 2011). Testitapauksia voidaan tehdä sarjassa niin, että ensimmäinen testitapaus käynnistää seuraavan, kun edellinen on suoritettu onnistuneesti (Turpeinen 2015). Esimerkki manuaalisesta testitapauksesta on liitteessä 1.

Toinen testitapausten suoritustapa manuaalisten lisäksi on testitapausten automatisointi. Automaattisten testitapausten eri ohjelmointimetodeja ovat (Klärck 2012, 4-8):

- Nauhoitus- ja toistometodi
- Lineaarinen skriptaus
- Modulaarinen skriptaus
- Data-ajettu testaus
- Avainsana pohjainen testaus

Nauhoitusmetodissa testausohjelmalla nauhoitetaan jokainen manuaalisesti suoritettava testitapaus yhden kerran. Nauhoitettu tapaus voidaan tämän jälkeen suorittaa niin usein kuin on tarpeellista. Lähestymistavan etuna on, että tapauksia on helppo tehdä eikä niiden teko vaadi ohjelmointitaitoja. Huonona puolena on, että testitapaukset menevät helposti toimimattomiksi, jos testattavaa muutetaan. Lisäksi niitä on vaikea ylläpitää ja järjestelmän tulee olla täysin valmis, jotta sitä voidaan testata. Nauhoitus- ja toistometodi soveltuvat vain hyväksymistestaustasolle. Ne eivät myöskään sovellu suuren mittakaavaan automaatioihin. (Klärck 2012, 4-8.)

Lineaarisessa skriptauksessa testitapaukset ovat suorassa yhteydessä testattavaan järjestelmään. Niiden ohjelmoimiseen voidaan käyttää mitä tahansa yleistä skriptaus-ohjelmointikieltä, jolloin ei tarvita myöskään maksullisia lisenssejä. Lineaarisia testitapauksia on helppo luoda, ja ne taipuvat moneen tarkoitukseen, mutta samalla ne ovat herkkiä hajoamaan, jos järjestelmää muutetaan. Yksi järjestelmän muutos voi rikkoa kaikki testitapaukset. Lisäksi niitä on työlästä ylläpitää, kun testitapauksia on monia. Skriptejä ei voida myöskään käyttää uudelleen. Ne soveltuvatkin parhaiten yksinkertaisten tapauksien tarpeisiin, eikä niitä suositella suurten mittakaavojen automaatioihin. (Klärck 2012, 9 -13.)

Modulaarisessa skriptauksessa testitapauksia ajaa erillinen ajuri. Interaktion testattavan järjestelmän kanssa hoitavat testikirjaston funktiot. Modulaarisella skriptauksella testitapaukset ovat uudelleen käytettäviä, ja uusia testejä voidaan näin ollen luoda nopeammin. Niitä on helppo ylläpitää, ja yleensä muutokset järjestelmään vaativat vain pienen alueen muutoksia itse testitapauksiin. Ajureita pystyvät luomaan ja muokkaamaan myös ohjelmointitaidottomat testaajat. Modulaaristen tapausten ongelmana on, että ne vievät aluksi paljon aikaa, kun ajureita luodaan. Lisäksi niissä testitapausten tieto on skriptien sisällä, mikä hankaloittaa tapausten ymmärtämistä. Uudet testit vaativat myös aina uudet ajurit. Modulaarisuus soveltuu yksinkertaisiin testauksiin, mutta myös suurempiin jos kaikki testaajat osaavat ohjelmoida. (Klärck 2012, 14-18.)

Data-ajetuissa testitapauksissa itse testaustieto on irrotettuna varsinaisista testitapauksista. Testaustieto voi sijaita esimerkiksi Excel-kaaviossa. Data-ajossa yksi ajuri voi ajaa yhtä aikaa montaa samanlaista testitapausta, mutta erilaisiin tapauksiin vaaditaan uudet ajurit. Testikirjastot luovat metodiin modulaarisuutta kuten modulaarisessa skriptauksessa aikaan-saaden samat edut. Vanhojen testien muokkaus ja uusien luonti on helppoa, eikä se vaadi ohjelmointitaitoja, mutta ohjelmoijat joutuvat silti ohjelmoimaan testien automaation ja ajurit. Testitapausten osalta ongelmiksi muodostuvat tapausten samankaltaisuus sekä se, että uudenlaiset testit vaativat aina uuden ajurin. Metodin käyttö vaatii myös alussa paljon työtä, kun luodaan uudelleen käytettäviä komponentteja, kuten esimerkiksi tiedon jäsentäjiä (parsers). Data-ajo soveltuu

paremmin suuriin kuin pieniin projekteihin, joihin se on liian monimutkainen. (Klärck 2012, 19-23.)

Avainsana-pohjaisessa testitapauksessa ei ole ainoastaan tietoa vaan myös avainsanoja, jotka kertovat, miten saatua tietoa voidaan hyödyntää. Avainsanat ja niiden testitieto ajavat testausta. Avainsanoilla saavutetaan samat edut kuin data-ajetuissakin tapauksissa, kuten erilliset testitapaukset ja testitiedot. Testejä voidaan myös luoda avainsanoista vapaasti, jolloin ohjelmointikyvyttömät testaajat pystyvät luomaan uudenlaisia testejä ilman tarvetta uudelle ajurille. Oikeanlaiset avainsanat mahdollistavat myös data-ajetut testitapaukset. Metodissa ei tarvita erillisiä ajureita, vaan kaikki testit voidaan suorittaa samalla ohjelmakehyksellä. Avainsana-pohjaisen järjestelmän pystytys voi vaatia alussa paljon, mutta se soveltuu hyvin suuren mittakaavan projekteihin. Siinä voidaan käyttää jo luotuja ratkaisuja, mikäli järjestelmää ollaan kehittämässä esimerkiksi data-ajetusta ratkaisuista avainsana-pohjaiseen testaukseen. Toisaalta se ei sovellu pienen mittakaavan projekteihin sen vaativuuden vuoksi. (Klärck 2012, 24-28.)

3.3 Integrointitestaus

Integrointitestaus on testausvaihe, jossa järjestelmän osia sovitetaan yhteen perimmäisenä tarkoituksena saada järjestelmä toimimaan kokonaisuutena. Integrointivaiheessa uusi komponentti integroidaan osaksi testattua kokonaisuutta. Jos uudessa osassa ei vielä ole tarvittavia kytköksiä, joudutaan tekemään niiden tilalle ns. tynkiä (stubs), jotka toimittavat näiden virkaa sen ajan, kunnes integraatio saadaan testatuksi. Ideana siis on, että toimivaan kokonaisuuteen lisätään osia ja tarkastetaan kokonaisuuden toimivuus. (Kasurinen 2013, 54.)

Komponenttien integrointijärjestykseen on neljä lähestymistapaa (Kasurinen 2013, 55):

- Alhaalta ylöspäin
- Ylhäältä alaspäin
- Voileipätestaus

- Big bang

Alhaalta ylöspäin -järjestyksessä tuodaan järjestelmään ensin inkrementaalisesti kaikista matalimman tason komponentit, jotka kommunikoivat suoraan raudan tai käyttöjärjestelmän kanssa. Sitten kytketään komponentteja, jotka käyttävät edellisiä kytkettyjä ja testattuja komponentteja niin kauan, kunnes järjestelmään on tuotu kaikki komponentit. Alhaalta ylöspäin menetelmän etuna on, että sillä löydetään helposti matalan tason virheet, joita ei löydettäisi käyttäessä muita järjestyksiä. (Kasurinen 2013, 55.)

Ylhäältä alaspäin -menetelmässä tehdään integrointi samoin kuin alhaalta tapahtuvassa, mutta käänteisessä järjestyksessä, ylhäältä kohti syvempiä järjestelmän komponentteja. Ylhäältä alaspäin -menetelmällä saadaan kokonaiskuva järjestelmästä aiemmin, ja samalla löydetään siitä puuttuvat toiminnot. (Kasurinen 2013, 55). Voileipätestauksessa integraatio aloitetaan molemmista päistä yhtä aikaa ja edetään kohti niiden yhdistymispistettä. Voileipätestauksen etuna on, että tynkiä ei tarvitse tehdä niin paljon kuin muissa menetelmissä, mutta osien yhteensovittaminen vaikeutuu sitä mukaa, kun integraatio etenee. (Kasurinen 2013, 55.)

Big bang -testauksessa kaikki integraatiotestatut komponentit lyödään yhteen kerralla ja katsotaan, miten hyvin järjestelmä kestää. Tätä kutsutaan myös ”savutestiksi”. Sillä voidaan nopeasti tarkastaa järjestelmän toimivuus. (Kasurinen 2013, 55.)

Näitä menetelmiä kutsutaan yhdessä ”Integraatio pienessä mittakaavassa” (Integration in the small) -menetelmiksi, joille yhteistä on, että järjestelmään tuodaan eristyksissä yksikkötestattuja komponentteja osaksi järjestelmää ja näin ollen testataan, miten ne toimivat keskenään, keskittyen niiden välisiin rajapintoihin (interfaces). Näin löydetään ne virheet, joita ei voi löytää yksikkötestauksen aikana ja testataan, että järjestelmän komponentit toimivat suunnitelmien mukaan molemmalta puolelta integraatiota. (Ghahrai 2008.)

”Integraatio suuressa mittakaavassa” (integration in the large) -vaiheessa testataan kokonaisen järjestelmän toimivuutta toisten järjestelmien ja verkkojen

kanssa. Analogiana Ghahrai käyttää taloa: Talo tarvitsee sähköä, vettä, puhelinlinjat jne. toimiakseen kunnolla. Järjestelmäkin tarvitsee rajapinnan, jolla kommunikoida eri verkkojen ja käyttöjärjestelmien lävitse. (Ghahrai 2008.)

3.4 Jatkuva integrointi

Jatkuvalla integroinnilla tarkoitetaan tapaa, jossa ryhmän jäsenet integroivat työtään jatkuvasti ja vähintään päivittäin, jopa monesti päivässä. Jokainen integraatio verifioidaan automaattisesti käännöksellä, johon sisältyy testaus. Tällä mahdollistetaan virheiden löytyminen mahdollisimman varhaisessa vaiheessa, mikä johtaa parantuneeseen laatuun ja vähentää integrointiongelmia, sekä auttaa kehittäjiä luomaan ohjelmia nopeammin. (Fowler 2006.)

Onnistuneen jatkuvan integroinnin ainekset ovat (Fowler 2006):

- Toimiva versionhallinta
- Automatisoitu käännös
- Testaus on integroitu käännösvaiheessa
- Jokainen ohjelmoija committaa versionhallinnan päähaaraan päivittäin
- Jokainen commit kääntää versionhallinnan päähaaran integrointi-koneella
- Korjataan rikkoutuneet käännökset heti
- Pidetään käännös nopeana

Jatkuvan integroinnin etuina normaaliin integrointiin on, että sen avulla integroinnin kesto on paljon helpommin ennustettavissa ja että siinä nähdään aina, miten pitkällä prosessi on. Joka hetki tiedetään, missä mennään, mikä toimii ja mikä ei, sekä nähdään räikeimmät virheet. Kun virheet löydetään aikaisessa vaiheessa, ne on helpompi korjata. Lisäksi koska järjestelmästä on muutettu vain pientä osaa, vian syytä ei tarvitse etsiä niin syvältä järjestelmästä. Jatkuvalla integraatiolla saavutetaan myös nopea julkaisu. Nopea julkaisu tuo uudet

ominaisuudet käyttäjille nopeammin, jolloin he pääsevät antamaan niistä palautetta aikaisemmin, ja näin ollen osallistuvat järjestelmän kehitykseen. (Fowler 2006.)

3.5 Käyttöliittymätestaus

Käyttöliittymätestaus kuuluu järjestelmätestaustasolle. Järjestelmätestauksessa testataan, vastaako järjestelmä sille suunnitteluvaiheessa asetettuja funktionaalisia vaatimuksia. (Functional Testing 2011). Funktionaalinentestaus voidaan toteuttaa kahden eri näkökulman kautta, vaatimus- tai liiketoimintaprosessipohjaisesti. Vaatimuspohjaisessa testauksessa funktionaaliset vaatimukset priorisoidaan riskien mukaan, jolloin huolehditaan siitä, että kriittisimmät ja tärkeimmät testit varmasti suoritetaan. Liiketoimintaprosessipohjaisessa testauksessa testaus tehdään liiketoiminnan näkökulmasta käyttäen hyväksi tietoa kyseisistä prosesseista, esimerkiksi: henkilön palkkaus, tämän työura yrityksessä ja henkilön poistuminen yrityksestä. (What is functional testing?, n.d.)

Tyypillisesti funktionaalisessa testauksessa käydään läpi seuraavat työvaiheet (Functional Testing 2011):

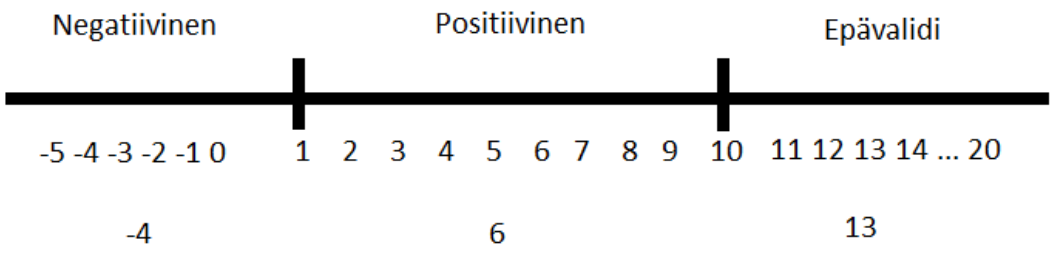
1. Tunnistetaan toiminnot, jotka ohjelman tulisi tehdä.
2. Luodaan syötteet tunnistettujen toimintojen pohjalta.
3. Määritetään haluttu lopputulos, jonka syötteet tuottavat.
4. Käydään läpi testitapaus.
5. Verrataan haluttua ja saatua tulosta.

Menetelmänä käyttöliittymätestauksessa käytetään ”musta laatikko” (black box) -menetelmää. Musta laatikko -testauksessa ohjelmalle annetaan syötteitä ja katsotaan, mitä ohjelma tekee, näkemättä mitä ohjelman sisällä eli ohjelmakooditasolla tapahtuu. Annettavat syötteet ja tehtäväsarjat määritellään testitapauksissa. Niissä kuvataan, miten laitetta tulisi käyttää ja miten sen pitäisi reagoida eri tehtäväsarjoihin. Musta laatikko testausta voidaan käyttää missä tahansa testauksen funktionaalisessa työvaiheessa tai sellaisissa vaiheissa,

joissa ohjelmasta on olemassa käynnistyvä versio. Muita yleisiä testauksen ”laatikkomenetelmiä” ovat ”lasilaatikko”, jossa nähdään ohjelman sisälle eli voidaan testata kooditasolla, sekä ”harmaa laatikko”, joka on yhdistelmä edellisiä menetelmiä. Harmaa laatikko yhdistää aiempien menetelmien parhaat ominaisuudet: mustan laatikon vaatimusmäärittelyjen pohjalta tehtyjen testitapausten hyödyntämisen sekä lasilaatikkotestien mahdollisuuden tarkastella järjestelmää myös sen sisältä. Harmaa laatikko -testaus käy hyvin esimerkiksi tapauksiin, jossa järjestelmää ei voida kokonaisvaltaisesti testata lasilaatikon tasolla, mutta jossa testattavan järjestelmän paikallisiin komponentteihin voidaan päästä käsiksi. Esimerkiksi verkkopalvelut voidaan luokitella harmaa laatikko -testauksen alaisuuteen. (Kasurinen 2013, 65.)

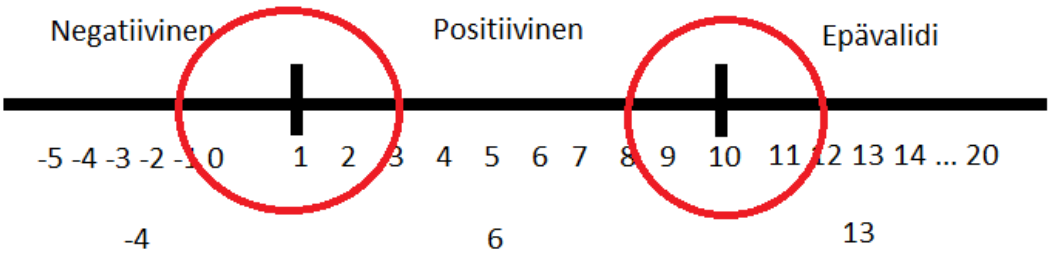
Musta laatikko menetelmän tekniikoina käytetään EP (Ekvivalenssi partitio) sekä BVA (Boundary value analysis, raja-arvo analyysi) -tekniikoita. EP-tekniikassa jaetaan samanlaiset testisyötteet luokkiin (partitio), joissa järjestelmä kohtelee niitä samoin (ekvivalenssi, samankaltaisuus). EP -tekniikassa tarvitsee testata vain yksi syöte per ryhmä, koska voidaan olettaa, että ryhmän sisältämiä syötteitä käsitellään samoin koko järjestelmän sisällä. Jos yksi syöte luokassa toimii, voidaan olettaa loppujenkin toimivan, jolloin niitä ei tarvitse testata. Toisaalta, jos jokin syöte ei toimi, silloin mikään muukaan ryhmän syöte ei toimi. Tämä alentaa huomattavasti testitapausten määrää sekä varmistaa, että valitaan sellaiset testitapaukset, joilla käydään kaikki tarvittavat syötteet läpi. (What is equivalence partitioning in software testing, n.d.)

BVA -tekniikassa testataan luokkien väliset raja-arvot. Raja-arvo analyysissä voidaan esimerkiksi valita testattavaksi määritetyn rajan lisäksi sen ylittävä ja alittava arvo. (Vihari 2015).



Kuvio 2. EP -luokiksi on valittu negatiiviset, positiiviset sekä epävalidit syötteet.

Luokkien syötteiksi valitaan luvut -4, 6 ja 13.



Kuvio 3. BVA -menetelmällä on valittu raja-arvot 0,1,2 sekä 9,10 ja 11.

Yhdistämällä EP ja BVA -tekniikat, saadaan kuvioiden 4 ja 5 mukaan valituksi testattavat syötteet -4, 0,1,2,6,9,10,11 ja 13.

BVA -tekniikkaa voidaan käyttää myös kokonaisten sanojen tarkistukseen. Sanan sisältämiä kirjaimia voidaan pitää luokkana. Esimerkiksi jos sanassa on 30 kirjainta, EP -luokiksi voidaan valita: 0 kirjainta, 30 kirjainta ja yli 30 kirjainta, joista validi luokka on siis 1 -30. Epävalideina luokkina voidaan pitää nollaa tai alle nollaa kirjainta ja 31 kirjainta sekä sen ylittävät kirjaimet. (What is Boundary value analysis in software testing? n.d.)

3.6 Testausautomaatio

Testausautomaatiolla tarkoitetaan testaustoimintaa, jossa ohjelman testaamista varten rakennetaan automaattivälineitä testien suorittamista ja niiden luomista varten. Sen tavoitteena on ottaa toistuvia testitapauksia ja rakentaa niiden nopeaa tarkastamista varten erillinen järjestelmä, jonka pohjimmaisena ajatuksena on vapauttaa testaajat muihin työtehtäviin. Jos ohjelmasta tehdään esimerkiksi joka yö uusi käännös, ei testaajien ajankäytön kannalta ole järkevää käyttää joka päivä aikaa samoihin perustesteihin. Automaattitestit voivat olla myös tarkastuksia, jotka tietokone tekee yön aikana, jolloin kehittäjät voivat aloittaa työpäivänsä läpikäymällä tarkastuksen tulokset ja korjaamalla havaitut virheet sekä ongelmat. Testausautomaatiolle parhaiten soveltuvia koh-

teita ovat esimerkiksi moduulien rajapinnat sekä yksittäisten moduulien yksikkötestaukset. Testausautomaatiolla voidaan myös luoda toimintojen nauhoitusohjelmilla käsky- tai toimenpidesarja, jolla esimerkiksi käytetään hakukonetta tai tiedostoja. Ohjelma suorittaa sarjan toimenpiteitä ja tarkistaa vastaavatko ne hyväksymisehtoja. (Kasurinen 2013, 76.)

3.7 Testaustyökalut

Testaustyökaluilla tarkoitetaan ohjelmia, jotka on tehty ohjelmistotestaukseen tai testausprosessin helpottamiseen. Testaustyökalut voidaan jakaa viiteen kategoriaan (What are different types of software testing tools, n.d):

1. Testauksen hallintatyökalut
2. Staattisentestauksen työkalut
3. Testausspesifikaatio työkalut
4. Työkalut testien ajamiseen ja dokumentointiin
5. Monitoroinnin ja suorituskyvyn työkalut

Kasurinen (2013) jakaa testaustyökalut kahteen pääluokkaan: hallintatyökaluihin ja tekemisen työkaluihin. Hallintatyökaluilla hallitaan testauksen prosessia, ja tekemisen työkalut ovat ohjelmia, joilla varsinainen testaus voidaan suorittaa. (Mts. 84 -91.)

Työkalujen valintakriteereinä Musicinfon puolelta ovat avoin lähdekoodi, käytettävyys sekä testien ylläpidon helppous. Lisäksi niiden tulisi integroitua nykyisiin järjestelmiin saumattomasti. (Turpeinen 2015). Testien ylläpitoa helpottavat testitapausten ohjelmointikielten ominaisuudet ja ohjelmakoodin helppolukuisuus. Ohjelmointikieli määrittää osittain myös sen, miten monimutkaisia testitapauksia voidaan kirjoittaa.

Integraatio- ja järjestelmätestaustyökalujen alustana Musicinfolla on Jenkins. Se oli otettu käyttöön yrityksessä ennen tätä opinnäytetyötä Samuli Heinovirran opinnäytetyön yhteydessä, joten se oli jo integroitu nykyisiin järjestelmiin. Versionhallintaa hoitaa Git, ja kanbantaulun visualisointiin sekä projektinhallintaan on käytössä Taiga.

Ohjelmistotestauksen tekemisen työkalut

Seuraavaksi esitellään opinnäytetyön tutkimuksen aikana testattavat ja verrattavat käyttöliittymä- ja järjestelmätestauksen tekemisen työkalut.

Splinter ja Selenium

Splinter on Python -kielellä ohjelmoitava avoimen lähdekoodin testaustyökalu web-pohjaisten sovellusten testaukseen. Sillä voidaan skripiteillä automatisoida selaintoimintoja, kuten käydä osoitteissa, ja käyttää sivujen erilaisia funktionaalisuuksia. (Splinter docs, n.d.)

Siihen lukeutuu seuraavia ominaisuuksia (Splinter docs, n.d):

- Usealla eri selaimella testaus samanaikaisesti
- Css ja xpath rakenteiden valinta
- Iframe ja alert testaus
- Voidaan suorittaa Javascriptiä sekä testata Ajaxia ja asynkronista Javascriptiä (Splinter docs, n.d)

Selenium on selainten automatisointityökalu, jolla testataan websivun funktionaalisuutta Selenese-skripteillä, mutta sillä voidaan lisäksi nauhoittaa manuaalisesti makroja. Selenium tukee kaikkia yleisimpiä selaimia. (What is Selenium, n.d.) Splinter pystyy suorittamaan myös Selenium-testitapauksia sen webdriverin avulla (Splinter docs, n.d.)

Rester

Rester on ilmainen komentorivipohjainen avoimen lähdekoodin ohjelmistokehys, jolla voidaan testata REST-rajapintoja. Sitä ohjataan joko YAML tai JSON -tiedostoilla, jolloin se soveltuu myös ohjelmoinnista tietämättömille testaajille. Resterillä voidaan tehdä kokonaisia testipaketteja tai yksittäisiä testejä. (Rester n.d.)

Soap UI

Soap UI on maksullinen graafisen käyttöliittymän sekä avoimen lähdekoodin ohjelmisto funktionaaliseen testaukseen. Soap UI:lla voidaan suorittaa automaattisia testejä, regressiota sekä rasitustestejä. Se tukee kaikkia standardeja protokollia ja teknologioita, kuten OAuth2, REST ja SOAP. Ohjelmasta on olemassa myös ilmainen versio, jolla testaus on huomattavasti rajoitetumpaa. (SoapUI vs. SoapUI NG Pro n.d.)

Chakram

Chakram on ilmainen avoimen lähdekoodin API -testauksen ohjelmistokehys, jolla voidaan suorittaa ”päästä päähän” JSON REST -testausta. Se on rakennettu Node.js, Mocha ja Chai -järjestelmien jatkeeksi, ja sitä ohjelmoidaan Javascriptillä. (Chakram, n.d.)

Frisby.js

Frisby on ilmainen REST API -testauksen ohjelmistokehys, joka on rakennettu Node.js:n ja Jasminen jatkeeksi. Sitä ohjelmoidaan Javascriptillä. Sillä voidaan testata HTTP -statuskoodeja sekä JSONia. (Frisby.js, n.d.)

Watir

Watir on kokoelma ilmaisen ja avoimen lähdekoodin Ruby-ohjelmointikielen kirjastoja, joilla voidaan automatisoida internetselaimien toimintoja. Windows-versio tukee vain Internet Exploreria, mutta webdriverilla voidaan ajaa kaikkia suosituimpia selaimia. Watirilla voidaan ottaa yhteys tietokantoihin, lukea tiedostoja ja siitä saa ulos Xml-raportin. (Watir, n.d.)

Windmill

Windmill on ilmainen ja avoimen lähdekoodin testauksen ohjelmistokehys. Sillä voidaan automisoida selainten toimintoja kirjoittamalla testitapauksia Javascriptillä, Pythonilla tai nauhoittaen makroja graafisella käyttöliittymällä. Se tukee myös asynkronista Ajaxia. (Meet Windmill, n.d.)

Ranorex

Ranorex on maksullinen testaustyökalu, jolla voidaan testata muitakin testaustasoja käyttöliittymän lisäksi, kuten suorittaa hyväksymistestausta. Sen skriptauskielenä käytetään .NET –teknologioita. (Ranorex, n.d.)

Ohjelmistotestauksen hallinnantyökalut

Testauksen hallinnantyökaluja toimeksiantajalla ovat:

Git

Git on Linus Torvaldsin suunnittelema ja kehittämä ilmainen ja avoimen lähdekoodin hajautettu versionhallinta järjestelmä, joka on suunniteltu kaikenkokoisille ohjelmistoprojekteille. Versionhallinnalla tarkoitetaan järjestelmää, joka tallentaa tiedostoissa tapahtuvat muutokset niin, että muutokset voidaan tarvittaessa kumota tai palauttaa. Gitissä ohjelmien versiot sijaitsevat repositorioissa. Repositoriot sijaitsevat paikallisina hakemistoina käytetyssä järjestelmässä, mutta ne voidaan tarvittaessa ladata erilisiin internetissä sijaitseviin varastoihin, kuten Github. Ohjelmakoodien versiot voidaan tallentaa repositorioiden niin sanottuihin haaroihin, joita ovat esimerkiksi Master ja Development, jolloin ohjelman kehitysversio ja lopullinen versio voidaan pitää kehityksen aikana erillään. Kehityksen edetessä Development-haarassa tapahtuneet muutokset voidaan yhdistää Master-haaraan ”merge”-toiminnolla. Haaroja voidaan tehdä tarpeen mukaan lisää ja yhdistellä niitä taas muihin haaroihin. Hajauttamisella tarkoitetaan tapaa, jossa ohjelmakoodista kloonataan koko repositorio. Tällöin jokaisella kehittäjällä on käytännössä aina kokonainen varmuuskopio repositoriosta. (About Git, n.d.)

Taiga

Taiga on ilmainen ja avoimen lähdekoodin projektinhallintaohjelma ketterille kehittäjille. Taigalla voidaan visualisoida sekä Scrum- että Kanban-taulua. Taulujen kokoa ja sarakkeiden määrää voidaan muokata projektin tarpeiden mukaan. Taiga tukee kaikkia Kanbanin sääntöjä. (What is Taiga n.d.)

Jenkins

Jenkins on testipalvelinohjelma, joka monitoroi toistettavien töiden, kuten ohjelmistojen, rakentamisen suorittamista. Siihen on saatavilla yli tuhat liitännäistä, joilla sen ominaisuuksia voidaan laajentaa, kuten Python ja Git -liitännäiset järjestelmätestaukseen ja versionhallinnasta testauksen käynnistämiseen. Jenkinsistä saa testauksen tulokset Rss-syötteenä sekä sähköpostilla. Jenkinsissä ohjelmistot rakennetaan askeleittain niin sanotuissa käännöksissä (englanniksi build). Askelten sekä rakentamisen jälkeen voidaan määrittää niiden jälkeen tehtävät lisätestausautomaatiot. (Shatzer 2015.)

Jenkinsin perusvalikot ovat (Shatzer 2015.):

- Projekti (englanniksi Item), josta luodaan uusia käännös -projekteja.
- Käyttäjät, jolla hallitaan ja muokataan Jenkinsin käyttöoikeuksia.
- Käännöshistoria, josta näkee aikajanalla käännöksien historian.
- Hallitsee Jenkinsiä, jolla hallitaan Jenkins järjestelmän asetusten lisäksi muun muassa liitännäisiä.
- Credentials, jolla hallitaan ulkoisia käyttöoikeuksia ja käyttäjiä, esimerkiksi versionhallinnan ja SSH -yhteyksien oikeuksia.

Projekti-valikosta luotuun Jenkins -projektiin voidaan määrittää ”Build trigger” eli tapahtuma, joka käynnistää projektin kääntämisen. Tämä voi olla esimerkiksi versionhallinnassa havaittu lähdekoodin muutos, kun ohjelmoija lähettää sinne uuden version. Versionhallinnan asetukset määritellään kohdassa Source Code Management. Siellä asetetaan Git-osoite, valitaan repositoriota vastaava credentials sekä repositorion-haara, jota ollaan kääntämässä. Build triggeriksi voidaan määritellä myös ajastus, jolla saavutetaan jatkuva integrointi. Kolmantena vaihtoehtona on käynnistää käännös, kun muut projektit on käännetty. Nämä projektit voidaan määrittää itse, jolloin ne voidaan ketjuttaa. Build environment -kohtaan määritetään tarvittavat ympäristötekijät, kuten virtuaalinäytöt tai erilaiset ohjelmakoodi wrapperit. Projektien Build -kohtaan voidaan määrittää askeleet, joiden tulee tapahtua käännöksen aikana. Näitä askeleita voidaan lisätä tarpeen mukaan. Askeleet voivat olla mm. skriptin ajoa

tai komentorivikäskyjä. Askelten jälkeisiin post-build actionseihin määritetään, mitä tehdään, kun käännös on suoritettu onnistuneesti. Näihin lukeutuvat muiden käännöksien suoritus, testiraporttien tulostus näytölle, sähköposti-ilmoitukset ja erilaisten analyysien tulokset. (Shatzer 2015.)

3.8 Virheraportit

Hyvän virheraportin ominaisuuksia ovat (Montvelisky 2008):

- Lyhyt ja tarkka otsikko
- Lyhyt sanainen ja ytimekäs kuvaus
- Hyvät liitteet
- Tarkka konteksti
- Oikea vakavuuden- ja järjestyksenaste

Lyhyellä ja tarkalla otsikoinnilla virheraportin lukija ymmärtää ja muistaa heti mihin työhön tai työvaiheeseen virhe liittyy. Se ei voi olla yleispätevä, kuten ”Virhe ohjelmassa”, koska silloin virhettä ei kohdisteta mihinkään. Liian pitkä otsikko tekee virheiden lukemisesta työlästä ja vie turhaan aikaa. Otsikossa pitäisi olla vain yksi lause, joka tarkentaa tärkeimmät virheen kohdat, samalla selventäen kuinka käyttäjä tai virheenkorjaaja voi sen kohdata. (Montvelisky 2008.)

Kuvauksella testaaja voi kertoa kaiken tiedon, joka voi auttaa virheen ymmärtämisessä, sekä sen, kuinka virhe voidaan korjata. Tärkeimmät kohdat kuvauksessa ovat työvaiheet, joilla virhe toistetaan, virheen seuraukset, eli se, mitä tapahtuu, jos käyttäjä käynnistää virheen, sekä odotettu tai ehdotettu toiminta virheen poistamiseksi. Kuvaukseen voidaan kirjoittaa kaikki se tieto, jota ei voida kirjoittaa muualle virheraporttiin, mutta toisaalta siihen ei pitäisi kirjoittaa liian paljon tai asiaan liittymätöntä informaatiota. Sääntönä voidaan pitää, että 3 -10 riviä tekstiä riittää 80 prosentille tapauksista. (Montvelisky 2008.)

Hyvät liitteet ovat erityisen tärkeitä, jos tarvitsee näyttää graafisen käyttöliittymän tulosteita, virhetilanteita, virheilmoituksia tai log-tiedostoja, jotka täydentävät virheen kuvausta. Tärkeintä on liittää mukaan vain sellainen tieto, joka

liittyy asiaan. Esimerkiksi kuvakaappauksissa voidaan rajata kuva-ala kattamaan vain virheen sisältämä kohta kuvasta. Lisäksi liitetiedostot tulisi pakata, jotta säästetään tilaa. (Montvelisky 2008.)

Jos käytetään rakenteellista virheidenjäljitysjärjestelmää, virheraportissa on tällöin kentät virheen kategorioimiseen, kuten moduuli, infrastruktuuri, selain jne. Kentät helpottavat kategorisoimaan, ja joissakin tapauksissa helpottavat tuottamaan virheen uudelleen. (Montvelisky 2008.)

Oikealla vakavuuden- ja järjestyksen asteella pyritään pitämään vakavuusasteet hallinnassa, nostamatta kaikkia virheitä kiireisimpään luokkaan, jolloin tärkeysjärjestys sotkeutuu ja vakavat virheet hukkuvat muiden joukkoon. Tämä voi madaltaa mahdollisuutta, jolla virhe tullaan korjaamaan. Liiallinen kriittisten virheiden raportointi voi myös saada raportin vastaanottajan skeptiseksi virheen vakavuudesta. Näin ollen hän ei ota virheitä vastaan virheen vaatimalla asenteella, koska on tottunut virheellisyyden tärkeysjärjestyksen ja pitää kaikkia virheitä samanarvoisina. Jos on vaikea päättää tärkeysjärjestyksestä, voidaan konsultoida muita testaajia, kehittäjiä ja johtoa yleisten standardien luomiseksi. (Montvelisky 2008.)

Virheraportteja tulisi seurata ja kommentoida aina kun se nähdään tarpeelliseksi. Tämä on erityisen tärkeää silloin, kun kehittäjä tai muu sidosryhmä ei ymmärrä virhettä ja jättää sen tämän seurauksena myöhempään tai hylkää sen korjaamisen kokonaan. (Montvelisky 2008.)

Musicinfon pääohjelmoija Antti Jokipii (2015) kuvaa toimeksiantajan virheen raportoinnin prosessia seuraavasti:

”Ohjelmoija lataa työnsä versionhallintaan, joka käynnistää testauksen testauspalvelimella. Testauksen yhteydessä löydetyt virheet tulevat versionhallinnan kautta suoraan latauksen tehneelle ohjelmoijalle. Testauspalvelin ja versionhallinta pitävät automaattisesti kirjaa löydetyistä virheistä.”

4 Tutkimuksen toteutus

4.1 Lähtökohdat

Ohjelmistotestauksen keskeisen taustateorian keräämisen ja kirjoittamisen jälkeen tutkimus aloitettiin selvittämällä toimeksiantajan nykyinen testausprosessi, malli, testausalustat ja työkalut. Selvitys tehtiin tutustumalla testipalvelimeen SSH-etäyhteydellä. Toimeksiantajaa haastateltiin nykyisen testausprosessin tilasta ja käytössä olevasta testauksenmallista sekä niiden kehittämistarpeista. Samalla päätettiin ohjelmistojen ja työkalujen valintakriteereistä toimeksiantajan puolelta, jotka olivat hinta, avoin lähdekoodi sekä testien ylläpidon helppous.

Selvityksestä kävi ilmi, että toimeksiantajalla oli selkeä idea ohjelmistotestauksen automatisoinnista sekä siitä mitä haluttiin testata, mutta tälle ei ollut vielä tehty Jenkins-järjestelmään sopivia automaatioita. Toimeksiantaja oli selvittänyt vain käyttöliittymän osalta, mitä työkaluja käytettäisiin automaation toteuttamiseksi.

Ohjelmistotestauksen automatisoinnin kehittämiseksi selkeytyi alla kuvattu prosessi.

1. Ohjelmoija lataa työnsä Git -versionhallintaan.
2. Lataus käynnistää Jenkins -palvelimella testauksen.
3. Jenkins pitää kirjaa testauksen tuloksista.
4. Ohjelmoija saa Jenkinsin kautta tiedon testauksen tuloksesta.
5. Integraatio testataan päivittäin ajoitetulla testillä testaamalla järjestelmän JSON -rajapintoja.

Integraatiotestaus tulisi toteuttaa ”suuressa mittakaavassa”, koska rajapinnat olivat jo valmiina ja testattuja ”pienessä mittakaavassa”. Selvityksen aikana kävi myös ilmi tarve työkalujen Linux-yhteensopivuudesta testauspalvelimen ollessa Ubuntu -käyttöjärjestelmällä varustettu. Seuraavassa alaluvussa käydään läpi toimeksiantajan käyttämä testauksen malli.

4.2 Musicinfon tuotantomalli

Musicinfo käyttää Kanbania, joka on Lean -työkalu. Lean on johtamis- ja tuotantomenetelmä, joka juontaa juurensa autoteollisuudesta. Sen kehitti alun perin Toyota. (Turpeinen 2015). Kanbanissa on kolme sääntöä. Ensimmäinen sääntö on työmäärän visualisointi Kanban-taululla. Toisessa säännössä tarkoituksena on rajoittaa työmäärää työntekijäkohtaisesti. Kolmas sääntö mittaa työmäärää ja etenemistä. (The 3 most important Kanban rules, 2013.) Kanbanista käytetään Musicinfolla yksinkertaistetumpaa versiota, jossa käytössä on ainoastaan työnmäärän visualisointi kanban-taululla. Yhden säännön käyttö johtuu yrityksen startup-maisuudesta, jolloin se ei vielä näe tarpeelliseksi mitata työn määriä tai aikatauluja eikä rajoittaa niitä. (Turpeinen 2015.). Kanban-taulun visualisointiin yrityksessä käytetään Taiga-ohjelmaa. Taigassa voi määrittää itse tarvittavan määrän sarakkeita projekteille. Musicinfolla Taigan kanbantaulussa on kuusi saraketta eri työvaiheille. Testaus suoritetaan ”Ready to test” -kohdassa. Kanban-taulun rakenne taulukkona on liitteenä 2.

4.3 Tutkimus ja käytännön toteutus

Toimeksiantajan tarpeiden selvityksen jälkeen etsittiin ja tutkittiin työkaluvaihtoehtoja, joilla voitaisiin käytännössä toteuttaa automaatio integraatio- ja käyttöliittymätestaukseen niin, että ne pystyttäisiin suoraan sulauttamaan nykyisin käytössä olevaan testipalvelimen Jenkins -ympäristöön.

Integroinnin työkalujen vertailu

Tutkimus toteutettiin asentamalla testipalvelimelle integroinnin testaustyökalut, joita olivat Chakram, Rester, Frisby.js ja Soap UI. Taustateorian 3.7 Työkalut -luvussa esiteltiin kaikki integroinnin testaustyökalut sekä käyttöliittymän testaustyökalut. Valitut testaustyökalut täyttävät tutkimuksen valintakriteerit, koska ohjelmat integroituvat nykyiseen järjestelmään eli Jenkinsiin. Kaikilla työkaluilla suoritettiin seuraava automaattinen testitapaus:

1. Hae http GET -metodilla rajapinnasta test.musicinfo.io/json/v1/artist/87c5dedd-371d-4a53-9f7f-80522fb7f3cb/.

2. Tarkista, onko Björk -artistin alias "Bjørk" haetussa JSON -rakenteessa.

Odotettuna tuloksena on, että Björk artistin alias on Bjørk saadussa JSON -oliossa. Ekvivalenssin validiksi luokaksi on valittu sana ”Bjørk”. Tämän lisäksi testattiin integroituminen Jenkisin rakentamisen askeliin lisäämällä testauksen ajo komentorivikomennoksi. Näiden tutkimusten pohjalta valittiin tarkoitukseen sopivimmat työkalut.

Työkalujen arviointimenetelmänä käytettiin testaustyökalujen tutkimuksen aikana tehtyjä havaintoja niiden soveltuvuudesta testausjärjestelmään. Lisäksi valintaan vaikuttivat toimeksiantajan luomat valintakriteerit eli avoin lähdekoodi, ilmaisuus sekä integroituminen nykyiseen järjestelmään. Integroinnin testauksen työkaluksi valittiin Chakram, koska se täytti kaikki toimeksiantajan valintakriteerit. Lisäksi sillä oli helpoin tehdä ja ylläpitää testitapauksia. Resteriin verrattuna Chakramin ohjelmointikieli sopii paremmin toimeksiantajan tarpeisiin, sillä Javascriptillä voidaan suoraan lisätä testitapauksiin monipuolisuutta. Näitä etuja ei saavuteta JSON tai YAML -testitapauksilla, koska ne ovat merkintäkieliä, jolloin niistä puuttuvat ohjelmointikielten ominaisuudet, kuten silmukoilla toistaminen. Rester ei myöskään tukenut kunnolla utf-8 -merkistöä, jolloin sillä ei voinut hakea JSON-tietoa jossa oli esimerkiksi Bjørk -nimen ”ø” merkki. Soap UI:ta ei valittu integrointitestauksen työkaluksi, koska sillä oli rajalliset ominaisuudet ilmaisversiossa ja komentoriviltä toimimaton graafinen käyttöliittymä. Frisby.js ja Chakram olivat testitapausten ohjelmointiominaisuuksiltaan lähes identtiset, sillä molemmat käyttivät ohjelmointikielenään Javascriptiä. Chakramilla sai kuitenkin selkeämmät komentorivitulosteet ja virheilmoitukset, jolloin sen testitapausten luonti ja ylläpito helpottui. SoapUI:ta lukuun ottamatta kaikki työkalut integroituivat vaivatta Jenkins -ympäristöön rakennusaskeleessa suoritettavana komentorivi käskynä. Testaustyökalujen vertailutaulukko on liitteenä 3.

Käyttöliittymätestauksen työkalujen vertailu

Käyttöliittymäntestaukseen soveltuvia työkaluja testattiin samoin kuin integroinnin työkaluja. Testaustyökaluille tehtiin yhteinen testitapaus, joka suoritettiin kaikille taustateorian luvun 3.7 Testaustyökalut käyttöliittymätestaustyökaluille. Työkaluja olivat Selenium, Splinter, Watir, Windmill, Ranorex, Sahi, sekä Tellurium.

1. Mene musicinfo.io -sivulle.

2. Kirjoita hakukenttään Cmx ja paina enter tai klikkaa hakupainiketta.
3. Tarkista, onko sivulla teksti ”CMX”.
4. Tulosta "artisti löytyi", jos on, ja "artistia ei löytynyt", jos ei.

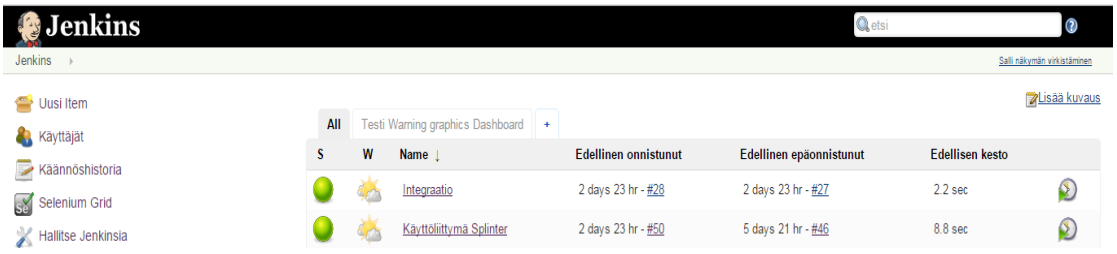
Testitapauksen odotettuna tuloksena on, että artisti löytyy, jolloin se läpäisee testauksen. Ekvivalenssin validiksi luokaksi on siis valittu sana ”CMX”.

Työkalujen arviointimenetelmänä käytettiin testaustyökalujen tutkimuksen aikana tehtyjä havaintoja niiden soveltuvuudesta testausjärjestelmään. Lisäksi valintaan vaikuttivat toimeksiantajan luomat valintakriteerit. Tellurium, Ranorex ja Sahi olivat maksullisia, jolloin ne eivät vastanneet toimeksiantajan valintakriteereitä. Watir on lähinnä kokoelma käyttöliittymän testaukseen soveltuvia Ruby-kielen kirjastoja kuin valmis työkalu, jolloin se ei ollut suoraan käyttöönotettava. Lopullinen valinta tuli tehdä Windmill, Selenium ja Splinter -ohjelmien välillä. Splinter oli parempi vaihtehto kuin Selenium, koska Seleniumilla oli hankalampi ajaa montaa selainta yhtä aikaa niin sanotussa ”headless”-tilassa, jossa selain on virtualisoitu. Splinterillä voidaan ajaa myös Selenium -testitapauksia tarvittaessa. Splinter oli helposti ja monipuolisesti skriptattavaa johtuen sen Python -ohjelmointikielestä. Lopulta valinta tuli tehdä Windmill ja Splinter -ohjelmien välillä, jolloin päädyttiin Splinteriin, koska siitä on saatavilla kattavampi dokumentaatio. Se on Python -kielensä johdosta helposti skriptattavaa ja ylläpidettävää ja samalla se on avoimen lähdekoodin työkalu. Kaikki testatut käyttöliittymätestaustyökalut integroituivat Jenkins käännöksiin saumattomasti komentorivikäskynä. Testaustyökalujen vertailutaulukko on liitteenä 3.

Jenkins

Jenkins -testausalusta asennettiin ensin paikallisesti työasemalleni. Työasemalla tutustuin sen tärkeimpiin taustateoriassa esiteltyihin ominaisuuksiin. Saatuani SSH-tunnukset toimeksiantajan palvelimelle käynnistin palvelimen Jenkins -palvelun. Selvitin järjestelmän osoitteen ja loin kansiot integraatiolle ja käyttöliittymätesteille testipalvelimelle sekä päivitin sen uusimpaan versioon. Poistin konfiguraatiotiedostosta kirjautumisen, koska se esti järjestelmän käyttämisen, ominaisuuksien muokkauksen ja lisäyksen. Asensin Python, Node.js, Selenium ja Git -liitännäiset ja päivitin jo asennetut liitännäiset uusimpiin versioihin. Credentials -valikkoon lisäsin Git -yhteyden tunnukset ja salasanat. Lopuksi kytkin kirjautumisen takaisin päälle.

Testauksen toteuttamista varten Jenkinsiin tarvittiin kaksi uutta projektia, yksi integraation testaukseen ja toinen käyttöliittymän. (Ks. Kuvio 4.)



Kuvio 4. Jenkins -käyttöliittymä, jossa projektii kullekkin testaustasolle

Tämän lisäksi konfiguroitiin Jenkins -järjestelmään Git -liitännäisen ja paikallisen repositorion avulla ns. ”post receive hook”. Sen avulla käyttöliittymä testaus käynnistyy automaattisesti jokaisesta repositorion muutoksesta. Tämä tapahtui lisäämällä repositorion paikallisen asennuksen .git kansion post-receive -tiedostoon rivi:

curl http://palvelimenosoite/git/notifyCommit?url=<git@musicinfo.fi:patternlab.git>[&branches=master]*

Curl -ohjelman palvelimelle lähettämä komento käynnistää Jenkinsinsissä skannauksen, joka käy läpi kaikki työt joissa:

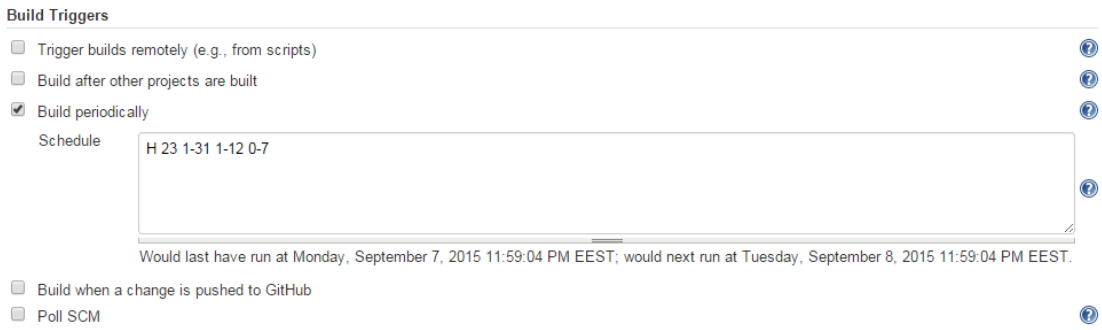
- Build trigger on "Poll SCM",

- on tarkoitus kääntää komennossa määritelty Git URL,
- on tarkoitus kääntää komennossa valinnaisena määritellyt haarat tai commit ID:t.

Git -liitännäinen etsii käännösprojekteista näitä ehtoja ja laukaisee Gitin pollingin. Polling etsii repositoriossa muutoksia, kuten uusia committeja, version paikallisia päivityksiä tai pusheja jos versio ladataan palvelimelle etänä. Käännös käynnistyy, kun polling on havainnut näitä muutoksia.

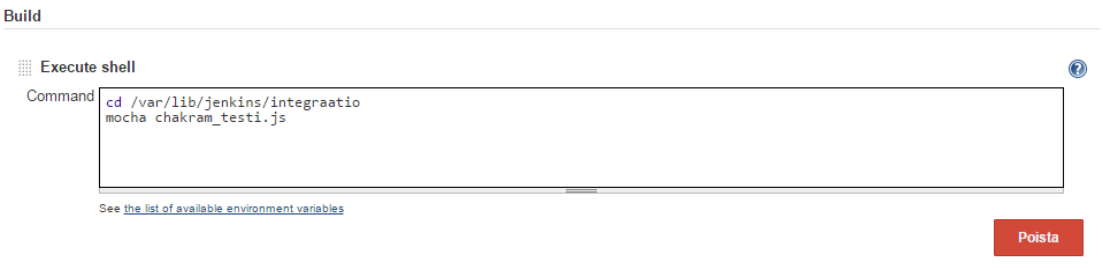
Integrointitestaus ja jatkuva integrointi

Chakram ajetaan Jenkinsissä Integrointi-projektin rakentamisen viimeisenä askeleena komentorivikomentona. Se suoritetaan joka päivä puolen yön aikaan. Tämä toteutettiin valitsemalla build triggeriksi ”Build periodically” eli ajastettu käännös. Ajastus muistuttaa syntaksiltaan Linux -ympäristöissä käytettyä ”cron”
-ohjelman ajastamista.



Kuvio 5. Integrointitestauksen ajastus

Itse integrointitestaus suoritetaan komentorivikomentona Build-kohdassa lisäämällä execute shell eli komentorivikomento. Komento ohjaa järjestelmän oikeaan kansioon sekä suorittaa testitapauksen komennolla mocha chakram_test.js.

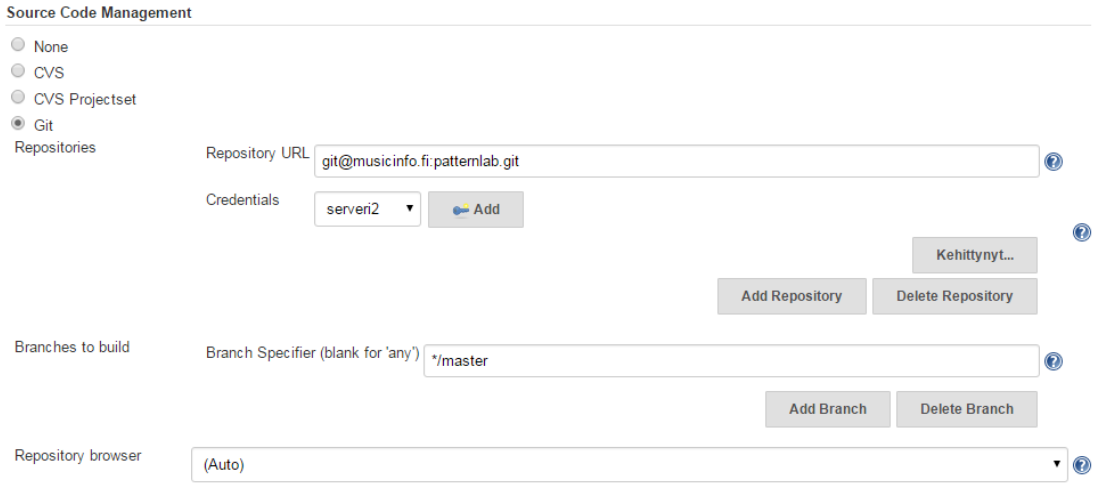


Kuvio 6. Integrointitestauksen suoritus

Integrointitestauksen testitapauksen sisältö on liitteenä 4.

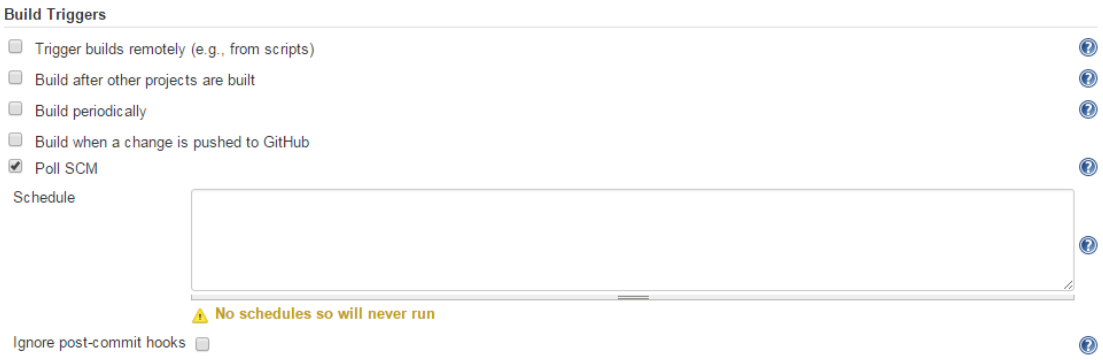
Käyttöliittymätestaus

Käyttöliittymätestauksen työkaluksi valittiin Splinter. Se integroitiin Jenkinsin -käyttöliittymätestauksen projektin viimeiseksi askeleeksi komentorivikomentona. Source code management kohtaan valittiin Git. Repository -kohtaan asetettiin palvelimen paikallisen Git -repositorion osoite sekä valittiin tätä varten aiemmin luotu credentials. Rakennettaviin haaroihin valittiin master, ja repository browser jätettiin oletusarvoiseksi.



Kuvio 7. Käyttöliittymäprojektin source code management -asetukset

Testaus käynnistyy aina, kun versionhallinnassa tapahtuu muutoksia. Tämä tehtiin lisäämällä projektiin build triggeriksi ”Poll SCM” eli rakenna, kun versionhallinnassa havaitaan muutoksia. Ajastukseksi ei tarvitse tässä tapauksessa määrittää mitään, koska käännös laukaistaan repositorion muutoksista. Siihen voi tarvittaessa määrittää myös tarkat ajankohdat sille, milloin versionhallinnan muutokset tarkistetaan.



Kuvio 8. Käyttöliittymäprojektin build trigger -asetukset

Käyttöliittymätestauksen varsinaiset testitapaukset ajetaan komentorivikomentona valitsemalla Build kohdassa execute shell ja ohjaamalla järjestelmä oikeaan kansioon sekä suorittamalla testi.

Kuvio 9. Käyttöliittymätestin komentorivikomento

Käyttöliittymätestauksen automaattisen testitapauksen sisältö on liitteenä 5.

Testien ketjutus

Lopuksi toteutettiin integraatio- ja käyttöliittymätestausketju:

- Tarkistetaan JSON –rajapinnasta, löytyykö artisti.
- Jos artisti löydetään, on käännös onnistunut, jolloin käynnistetään ketjun toinen käännös.
- Uusi käännös testaa, löytyykö sama artisti käyttöliittymästäkin.

Käytännössä toteutus vaati kaksi projektia, rajapintatestin ja käyttöliittymätestin. Käyttöliittymätesti käynnistyy vain, jos rajapintatesti on onnistunut. Ketju määriteltiin alkamaan ajoituksella. Testitapausten suoritus ja tuloksien tulostus tapahtuvat samalla tavoin kuin aiemmissa projekteissa.

5 Tutkimuksen tulokset

Tutkimuksen tuloksena toimeksiantajalle kehittyi jatkuvaan integraatioon ja käyttöliittymätestaukseen soveltuva Jenkins -ympäristö. Opinnäytetyötä aloittaessa toimeksiantajan lähtökohtana ohjelmistotestaukselle oli, että yritykselle saataisiin toimiva testausjärjestelmä. Heillä ei ollut ollut minkäänlaista testausta integraatio- tai käyttöliittymätestaustasoille, mutta yrityksellä oli jo olemassa järjestelmä, jonka pohjalta se voitaisiin rakentaa. Tutkimuksen avulla löydettiin oikeat vaihtoehdot toimivan testauksen toteuttamiseksi olemassa olevaan järjestelmään.

Käyttöliittymätestausta kehitettiin luomalla automaatio, joka käynnistyy versionhallinnan muutoksista. Käytännössä käyttöliittymää testataan aina, kun ohjelmoija lataa versionhallintaan uusimman version työstään. Automaatio testaa musicinfo.io -sivuston hakutoiminnon funktionaalisuutta Splinter -työkalulla ja antaa testaustuloksen konsolitulosteena Jenkins -järjestelmään. Tulosteesta voidaan luoda raportti.

Integraatiotestausta kehitettiin luomalla automaatio, joka on ajastettu käynnistymään puolen yön aikaan. Tällä, ja käyttöliittymän versionhallintaan liittämisellä, saavutetaan jatkuvan integraation tuomat edut. Automaatio testaa musicinfo.io palvelun rajapintoja ja niiden JSON -olioiden oikeellisuutta Chakram -työkalulla. Testitulos saadaan konsolitulosteena Jenkins -järjestelmään, josta voidaan edelleen luoda raportti. Järjestelmä luo myös automaattisesti RSS -syötteen kaikkien projektien tilasta.

Integraatio- ja käyttöliittymätestaukseen luotiin myös ketju, joka testaa ensin rajapinnan JSON -rakenteesta esimerkiksi artistia ja käynnistää onnistumisen jälkeen käyttöliittymätestin. Tämä tarkistaa saman artistin löytymisen myös käyttöliittymästä. Näin voidaan todeta, että käyttöliittymä tarjoaa käyttäjälle saman datan kuin rajapinta.

Toimeksiantajan testausmalli kehittyi ketterämmäksi, kun testaus suoritetaan aina versionmuutoksen jälkeen eikä kuten vesiputousmaisessa mallissa, jossa testaus suoritetaan vasta ohjelmointityön lopuksi. Jatkuva integraatio mahdollistaa virheiden varhaisemman havaitsemisen ja vähentää integroinnissa havaittuja ongelmia parantaen koko järjestelmän laatua.

Vastaukseksi tutkimuskysymykseen saatiin: Music.infon ohjelmistotestausta kehitetään luomalla yritykselle testausautomaatio. Integraatiotestaustasolla automaatiota kehitetään luomalla sille jatkuva integraatio. Käyttöliittymätestaustasolla automaatiota kehitetään luomalla yhteys versionhallinnan ja testauksen välille. Automaatiot mahdollistaa Jenkins -järjestelmä, joka on yhdistetty Git -versionhallintaan.

6 Pohdinta

Jarkko Jäsbergin (2011), Timo Sacklénin (2009) sekä Vesa Vilkmanin (2010) tutkimustuloksiin verrattuna tässä opinnäytetyössä päädyttiin erilaisiin työkaluihin toteutuksessa, mutta teoria säilyi samankaltaisena. Tämä johtuu siitä, että ohjelmistotestauksen teoria on pysynyt käytännössä samanlaisena testauksen alkuajoista lähtien, sillä käytössä eivät ole olleet täysin ketterät ohjelmistokehitysmenetelmät. Työkalujen valintaan liittyvät monet tekijät, kuten yrityksen ohjelmistokehityksen monimutkaisuuden luomat tarpeet testaukselle sekä se, millä tasolla testausta on. Jokaisella yrityksellä on omat valintakriteerinsä ja tarpeensa testaustyökaluille.

Tutkimuksen työkalujen vertailu- ja valintamenetelmä oli epätieteellinen ja perustui suurilta osin empiirisiin havaintoihin näiden toiminnasta ja soveltuvuudesta kyseiseen järjestelmään ja testattaviin kohteisiin. Kehittämistutkimus tutkimusmenetelmänä soveltui kuitenkin opinnäytetyöhön erinomaisesti, koska siinä tutkittiin erityisesti menetelmän laadullisia ominaisuuksia eikä pyritty keräämään kattavaa tietojoukkoa analysoitavaksi. Lisäksi siinä pyrittiin kehittämään jo olemassa olevaa järjestelmää paremmaksi. Opinnäytetyö ei kuitenkaan lisännyt tietoa tutkittavalla tai kehitettävällä alueella yleisesti, mikä johtui testauksen yleisestä kehityksestä.

Tutkimustuloksia voidaan yleistää pienten tai keskisuurten yritysten tarpeisiin. Kasvatettaessa testitapausten määrää ja testauksen monimutkaisuutta vaaditaan testitapauksilta siirtymistä modulaarisista data-ajettuihin tai avainsanapohjaisiin toteutuksiin niiden ohjelmoinnissa. Jenkins -alustan kattava liitännäisten määrä ja yhteensopivuus eri järjestelmiin ja työkaluihin mahdollistavat testauksen laajentamisen käytännössä mille tahansa testaustasolle. Toteutuksessa käytetyt työkalut ovat ilmaisia, avoimen lähdekoodin ohjelmistoja, jotka

eivät vaadi suurta perehtyneisyyttä ohjelmointiin, jolloin testitapauksia voivat tehdä ja ylläpitää ohjelmointitaidottomatkin testaajat. Tämä parantaa testauksen laatua, kun itse ohjelmoijat eivät testaa omaa ohjelmakoodiaan. Jatkossa järjestelmää voidaan kehittää myös jatkuvan julkaisemisen alustaksi tai tutkia mahdollisuuksia ”DevOps” -ohjelmistokehitykseen.

Lähteet

About Git. N.d. Viitattu 31.8.2015. https://git-scm.com/about.

Chakram. N.d. Readme.md tiedosto Github repositoriossa. Viitattu 27.8.2015. https://github.com/dareid/chakram.

Cohn, M. 2009.The Forgotten Layer of the Test Automation Pyramid. 17.9.2009. Viitattu 30.6.2015. http://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid.

Fowler, M. 2006. Continuous Integration. 1.5.2006. Viitattu 17.8.2015. http://www.martinfowler.com/articles/continuousIntegration.html.

Frisby.js. N.d. REST API Testing. Viitattu 27.8.2015 http://frisbyjs.com/.

Functional Testing. 2011. 9.9.2011. Viitattu 23.6.2015. http://softwaretesting-fundamentals.com/functional-testing/.

Ghahrai, A. 2008. Integration Testing in Small. 9.11.2008. Viitattu 16.6.2015. http://www.testingexcellence.com/integration-testing-in-small/.

Jokipii, A. Pääohjelmoija. Haastattelu 11.8.2015.

Kananen, J. 2008. Kvali: kvalitatiivisen tutkimuksen teoria ja käytänteet. Jyväskylä: Jyväskylän ammattikorkeakoulun julkaisuja -sarja.

Kananen, J. 2010. Opinnäytetyön kirjoittamisen käytännön opas. Jyväskylä: Jyväskylän ammattikorkeakoulun julkaisuja -sarja.

Khalili, M. 2013. Maintainable Automated UI Tests. 9.10.2013. Viitattu 16.6.2015. http://code.tutsplus.com/articles/maintainable-automated-ui-tests--net-35089.

Kasurinen, J.P. 2013. Ohjelmistotestauksen käsikirja. Jyväskylä: Docendo.

Kautto, T. 1996. Ohjelmistotestaus ja siinä käytettävät työkalut. Ohjelmistotekniikan seminaariesitelmä 21.11.1996. Viitattu 24.6.2015. http://www.mit.jyu.fi/opiskelu/seminaarit/ohjelmistotekniikka/testaus/#RTFToC1.

Klärck, P. Introduction to Test Automation. 12.4.2012. Viitattu 8.9.2015. http://www.slideshare.net/pekkaklarck/introduction-to-test-automation/.

Lewis, W.E. 2000. Software testing and continuous quality improvement. Florida: CRC Press LLC.

Meet Windmill. N.d. Viitattu 31.8.2015. http://www.getwindmill.com/.

Montvelisky, J. 2008. Principles of Good Bug Reporting. Viitattu 7.9.2015. http://qablog.practitest.com/2008/12/principles-of-good-bug-reporting/.

Musicinfo. N.d. Artikkeli Music Info Finland Oy:n sivuilla. Viitattu 3.7.2015. http://musicinfo.io/index.html.

Ranorex. N.d. Artikkeli Ranorex -ohjelman sivuilla. Viitattu 31.8.2015. http://www.ranorex.com/.

Rester. N.d. Readme.md tiedosto Github -repositoriossa. Viitattu 27.8.2015. https://github.com/chitamoor/Rester.

Shatzer, R. 2015. Meet Jenkins. 28.7.2015. Viitattu 24.8.2015. https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins.

SoapUI vs. SoapUI NG Pro. N.d. Viitattu 27.8.2015. http://www.soapui.org/about-soapui-pro/product-comparison/soapui-and-soapui-pro.html.

Splinter docs. N.d. Features. Viitattu 24.8.2015. https://splinter.readthedocs.org/en/latest/,

The 3 most important Kanban rules. 17.9.2013. Viitattu 3.7.2015. http://blog.elpassion.com/3-most-important-rules-of-kanban/.

Test case. Test case fundamentals. 17.10.2011. Viitattu 26.6.2015. http://softwaretestingfundamentals.com/test-case/.

Turpeinen, K. 2015. Karzasol Oy:n projektipäällikkö. Keskustelu 14.8.2015.

Vihari, C. 2015. Equivalence Class Partitioning and Boundary Value Analysis - Black Box Testing Techniques. 7.1.2015. Viitattu 17.8.2015. http://www.test-nbug.com/2015/01/equivalence-class-partitioning-and-boundary-value-analysis-black-box-testing-techniques/.

VRT Finland Oy ja Music.Info Finland Oy kolmannen Kasvu Open -kilpailun voittajat. N.d. Viitattu 3.7.2015. http://www.kasvuopen.fi/blogi/vrt-finland-oy-ja-musicinfo-finland-oy-kolmannen-kasvu-open-kilpailun-voit.

Watir. N.d. Automated testing that doesn't hurt. Viitattu 31.8.2015 http://watir.com/.

What are software development models. N.d. Viitattu. 6.7.2015. http://istqbexamcertification.com/what-are-the-software-development-models/.

What are the different types of software testing tools?. N.d. Viitattu 16.6.2015. http://istqbexamcertification.com/what-are-the-different-types-of-software-testing-tools/.

What is Boundary value analysis in software testing?. N.d. Viitattu 14.9.2015. http://istqbexamcertification.com/what-is-boundary-value-analysis-in-software-testing/.

What is Functional testing (Testing of functions) in software?. N.d. Viitattu 23.6.2015.http://istqbexamcertification.com/what-is-functional-testing-testing-of-functions-in-software/.

What is independent testing? Its benefits and risks. N.d. Viitattu 9.7.2015. http://istqbexamcertification.com/what-is-independent-testing-its-benefits-and-risks/.

What is equivalence partitioning in sotware testing. N.d. Viitattu 17.8.2015. http://istqbexamcertification.com/what-is-equivalence-partitioning-in-software-testing/.

What is Selenium. N.d. Viitattu 24.8.2015. http://docs.seleniumhq.org/.

What is Taiga. N.d. Viitattu 24.8.2015. https://taiga.io/.

What are different types of software testing tools. N.d. Viitattu 17.8.2015. http://istqbexamcertification.com/what-are-the-different-types-of-software-testing-tools/.

Liitteet

Liite 1. Manuaalinen testitapauspohja.

ID	Testitapaus	Alkuehdot	Kohdat	Odotettu tulos	Loppuehdot	Pass/Fail	Saatu tulos	Huomiot

Liite 2. Kanbantaulun rakenne.

New	To do	In Progress	Ready to test	Done	Archive
Ohjelmoija ottaa uuden työn.	Ohjelmoija varaa itselleen töitä.	Työt, jotka ovat työnalla.	Valmis työ valmiina testattavaksi.	Työ testattu ja valmis.	Arkisto säilytystä varten.

Liite 3. Testaustyökalujen vertailutaulukko.

Testitapaus	Nimi	Kohdat	Odotettu tulos	Pass/Fail	Saatu tulos	Huomiot
1	Selenium	Testitapaus 1	artisti löytyi	pass	artisti löytyi	
2	Splinter	Testitapaus 1	artisti löytyi	pass	artisti löytyi	
3	Watir	Testitapaus 1	artisti löytyi	pass	artisti löytyi	
4	Windmill	Testitapaus 1	artisti löytyi	pass	artisti löytyi	
5	Ranorex	Testitapaus 1	artisti löytyi	fail		Maksullinen
6	Sahi	Testitapaus 1	artisti löytyi	fail		Maksullinen
7	Tellurium	Testitapaus 1	artisti löytyi	fail		Maksullinen
8	Rester	Testitapaus 2	Bjørk	fail		Ongelmia skandinaavisten kirjainten kanssa

9	Chakr am	Testita-paus 2	Bjørk	pass	Bjørk	
10	Frisby. js	Testita-paus 2	Bjørk	pass	Bjørk	
11	Soap UI	Testita-paus 2	Bjørk	fail		Pelkästään graafinen

Liite 4. Integrointitestauksen automaattinen testita-paus.

```
var chakram = require('chakram'),
    expect = chakram.expect;

describe("Chakram", function () {
  it("Tarkistetaan onko Björkin alias Bjørk.", function () {
    var artist = "Bjørk";
    return chakram.get("http://test.musicinfo.io/json/v1/artist/87c5dedd-371d-4371d-
4a53-9f7f-80522fb7f3cb/")
    .then(function (aliasResponse) {
      var alias = aliasResponse.body.aliases[0].name;
      expect(alias).to.contain(artist);
    });
  });
});
```

Liite 5. Käyttöliittymätestauksen automaattinen testita-paus.

```
# -*- coding: utf-8 -*-

from pyvirtualdisplay import Display
from splinter import Browser

display = Display(visible=0, size=(1024, 768))
display.start()

browser = Browser('firefox')

browser.visit('http://musicinfo.io')
browser.fill('search', 'cmx\r')

if browser.is_text_present('CMX'):
    print "Artisti löytyi"
else:
    print "Artistia ei löytynyt"
```