HAAGA-HELIA
ammattikorkeakoulu

Yksikkötestaaminen projektissa Guild Charter

Minja Kataja

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2014

HAAGA-HELIA
ammattikorkeakoulu

Tiivistelmä

5.12.2014

Tekijä
Minja Kataja

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko
Yksikkötestaaminen projektissa Guild Charter

Sivu- ja liitesivumäärä
22 + 51

Opinnäytetyön otsikko englanniksi
Unit testing in project Guild Charter

Tässä opinnäytetyössä tehtiin testausta Guild Charter-projektiin. Guild Charter on Hannilan ja Wikströmin opinnäytetyöprojekti vuodelta 2014. Tämä opinnäytetyö käynnistettiin, jotta Guild Charter tulisi testatuksi. Tämän opinnäytetyön tavoitteena oli tuottaa Guild Charter-projektille testaussuunnitelma ja sen pohjalta edelleen yksikkötestejä.

Guild Charter-kehitysympäristöstä otettiin testausta varten oma kopio, johon lisättiin PHPUnit-viitekehys sekä sen tarvitsemat lisäosat. Yksikkötestit ja koodikattavuusanalyysi tehtiin PHPUnitin käytäntöjen mukaan. Testaussuunnitelma ja testausraportit luotiin eri kirjallisten lähteiden suositusten mukaan, ottaen kuitenkin huomioon tämän projektin tarpeet.

Työtä tehtiin noin kahdeksan viikon jaksoissa, jonka päätteeksi oli ohjauskokous. Ohjauskokoukset pidettiin Skype-ryhmäpuheluiden välityksellä. Työn edistymistä seurattiin edistymisraporttien avulla, jotka käytiin läpi ohjauskokouksissa.

Opinnäytetyön tuloksena syntyi testaussuunnitelma, jonka pohjalta edelleen tuotettiin yksikkötestejä sen verran mihin aika riitti. Lisäksi tuotettiin testausraportteja sekä koodikattavuusanalyysi.

Tulosten perusteella voitiin todeta, että ne osat jotka ehdittiin testata, toimivat. Joitakin osia ei pystytty testaamaan ongelmien vuoksi. Projektin aikana todettiin myös, että ohjelmakoodi kaipaa paikoitellen uudelleenkirjoitusta.

Guild Charterin kehittäjät saavat työn tuloksista eväät jatkaa itse testausta. He voivat käyttää tuotettuja dokumentteja ja koodia esimerkkeinä.

Asiasanat
Testaus, testaussuunnitelma, yksikkötestaaminen, PHPUnit

HAAGA-HELIA
University of Applied Sciences

Abstract

5.12.2014

Author
Minja Kataja

Degree programme
Degree programme in Information Technology

Thesis title
Unit testing in project Guild Charter

Number of pages and appendix pages
22 + 51

This thesis deals with testing within the framework of Guild Charter-project. Guild Charter was a Bachelor's thesis project carried out by Hannila and Wikström in 2014. The objective of this study was to create a test plan to the Guild Charter-project and then unit tests based on the test plan.

A copy of the Guild Charter-development environment was created, and then the required PHPUnit-framework and its essential parts were added to it. Unit tests and a code coverage analysis were made according to the PHPUnit practices. Test plan and test reports were created based on various recommendations found from various literary sources considering the needs of this project.

The work was accomplished in 8-week periods, which ended with a meeting. The meetings were held via Skype-conference calls. The progression of work was followed with progression reports, which were looked into at the meetings.

As a result of the study, there came a test plan, and unit tests based on it. Unit tests were made as long as there was time available in the project. Test reports and a code coverage analysis were also carried out during this project.

Based on the results, it can be stated that the tested parts of the program work. Some parts of the program were not tested because there occurred problems which could not be solved within given time constraints. During project it became clear that some parts of the code require refactoring.

Guild Charter's developers can use the results as an example how to test their code, and therefore they are able to continue testing by themselves.

Keywords
Testing, test plan, unit testing, PHPUnit

Sisällys

1 Johdanto

Tässä opinnäytetyöprojektissa tehtiin Guild Charter-ohjelman testausta. Tavoitteena oli tuottaa testaussuunnitelma, sekä edelleen sen pohjalta yksikkötestejä. Projekti käynnistettiin, jotta Guild Charter-ohjelma tulisi testattua mahdollisten virheiden varalta. Testaussuunnitelma toteutettiin tekstidokumenttina, ja yksikkötestit PHP-ohjelmointikielellä PHPUnit-viitekehyksen käytäntöjen mukaan. Lisäksi tuotettiin testausraportteja ja koodikattavuusanalyysi.

1.1 Projektin tausta

Guild Charter on Marko Hannilan ja Krista Wikströmin opinnäytetyönä keväällä 2014 tuottama alfa-versio verkkosivustosta, joka on suunnattu MMO-pelejä (Massive Multiplayer Online) pelaaville yhteisöille. Yhteisöt voivat perustaa sivuille omia alasivuja, joissa päätoimintona on keskustelupalsta. Sivustolle voi rekisteröityä ja käyttäjä voi olla jäsenenä useassa eri yhteisössä. Guild Charterin kehitys jatkuu edelleen, ja siihen on tulossa opinnäytetyönä vielä ulkoasu.

Tämä opinnäytetyöprojekti sai alkunsa, kun Hannila ja Wikström ehdottivat aiheeksi Guild Charterin testausta. Heillä oli tällöin oma opinnäytetyöprojektinsa vielä kesken ja aikaa oli vähän suhteessa työmäärään. Mikäli testaus toteutettaisiin erillisenä projektina, he voisivat jättää sen pois omasta projektistaan ja käyttää siihen varatun ajan muuhun.

Kirjoittajalla oli PHP-kielestä hallussa vain perusteet ja yksikkötestaamisesta ei ollut aiempaa kokemusta millään kielellä. Asiaa pohdittiin pari päivää tutustuen samalla Guild Charter-projektiin sekä yksikkötestaamiseen PHP-kielellä. Lopputuloksena haaste päätettiin ottaa vastaan.

Alkuvaiheen odotuksina oli oppia projektin aikana yleensä yksikkötestaamisesta ja syventää osaamista testaamisprosessista. Lisäksi odotuksina oli syventää osaamista PHP-kielestä ja oppia perusteet käytössä olleista viitekehyksistä.

1.2 Projektin rajaus

Tässä opinnäytetyössä keskityttiin testauksen suhteen ainoastaan yksikkötestaamiseen. Koska ohjelma on aika laaja ja opinnäytetyön tunteja rajallinen määrä, päätettiin mukaan ottaa ainoastaan kolme ohjelman ydinaluetta (Core, Forum ja SubCore). Näistä edelleen mukaan otettiin vain controllerit, modelit ja repositoryt. Esimerkiksi mustalaatikko-tyyppistä

testausta (black box testing) ei voitu ottaa mukaan, koska projektin ulkoasua ei ole vielä tehty. Testit päätettiin tehdä versioon alfa 1.0.1.

1.3 Miksi ohjelmia testataan?

Ohjelmistokriisi on termi, joka kehitettiin vuonna 1968 saksalaisessa konferenssissa. Ohjelmistokriisi kuvaa ongelmia, joita syntyi tietokoneiden suoritustehon sekä ohjelmien koon kasvaessa. Ohjelmissa oli paljon enemmän tarkastettavia asioita ja useampia mahdollisuuksia vikatiloille. Myös ohjelmistoprojekteja oli vaikeampi hallita, koska ne paisuivat ohjelmien mukana. Ohjelmistokriisin tärkein saavutus oli ohjelmistotuotannon ja ohjelmistoprosessien ajattelutavan synnyttäminen, ja siinä sivussa syntyi myös pohja ohjelmistotestauksen toiminnalle. (Kasurinen 2013, 10–11.)

Ohjelmistotestausta tehdään sen varmistamiseksi, että toteutettavasta ohjelmistotuotteesta tulee toivotun kaltainen, ja että kaikki siihen valmiiksi saadut ominaisuudet varmasti toimivat niin kuin oli tarkoitus. Toisin sanoen: varmistetaan että tehdään oikeaa tuotetta ja että tuote on tehty oikein. Testauksessa on tärkeää tunnistaa ne kohdat, joissa tuotos poikkeaa suunnitelmista. Ohjelmista ei kuitenkaan voida testata joka ikistä koodiriviä tai tapahtumaa, koska se vaatisi helposti kymmeniä tuhansia testejä, mikä taas vaatisi aikaa ja rahaa. Tämän vuoksi koskaan ei voida luvata, ettei ohjelmassa olisi virheitä. (Graham ym. 2011, 11, 18; Kasurinen 2013, 10.)

Vuonna 2002 Yhdysvalloissa tehtiin tutkimus, jonka mukaan yhdysvaltalaiset ohjelmistotalot menettävät vuosittain 59,5 miljardia dollaria puutteellisen tai vajavaisen testauksen aiheuttamina tappioiona ja tuotannonmenetyksinä. Summassa on huomioitu myös yritysten asiakkaille koituneet vahingot. Näissä luvuissa ei ollut mukana muita vahinkoja, kuten tahriutuneen julkisuuskuvan korjaamisen tai sen aiheuttamien menetettyjen kauppojen määrää. Parannukset testauksessa vähentäisivät summaa noin kolmanneksen. (Kasurinen 2013, 11; Thibodeau 2002.)

1.4 Projektin tavoitteet

Projektin tavoitteena oli tuottaa Guild Charter-projektiin englanninkielinen testaussuunnitelma, sekä sen pohjalta edelleen yksikkötestejä niin paljon kuin aika projektissa riittää. Testaussuunnitelma luotiin Google asiakirja-dokumentin muotoon ja yksikkötestit ovat php-ohjelmakoodia sisältäviä testiluokkia. Testien tuloksista tehtiin erilliset raportit Google asiakirja-muodossa.

Edellisten pohjalta osa Guild Charter-ohjelmasta tulee testattua, sekä kehittäjillä on jatkoa varten eväät jatkaa testausta.

2 Tietoperusta

Tässä kappaleessa käsitellään erilaisia opinnäytetyöprojektissa esiintyviä termejä sekä esitellään kehitysympäristössä käytössä olleet viitekehykset.

2.1 Viitekehys

Viitekehys (myös ohjelmistokehys tai sovelluskehys) tarkoittaa ohjelmistotuotetta, jonka tarkoitus on parantaa ja helpottaa ohjelman luomista. Viitekehykset koostuvat ohjelmointirajapinnoista (API eli application programming interface), koodikirjastoista, tukiohjelmista sekä työkaluista. Viitekehyksiä on erilaisia ja eri käyttöön sopivia. Esimerkiksi testausviitekehykset tarjoavat valmiit työkalut testien tekemiseen, ajamiseen ja tulosten raportoimiseen. (Baker, M. 2009; Brown, T. 2006.)

Viitekehyksen käytön etuna on, ettei joitakin osia koodista tarvitse kirjoittaa itse, vaan voi käyttää ennalta kirjoitettuja ja testattuja osia osana omaa ohjelmaa. Viitekehysten avulla voi muodostua parempia ohjelmointitapoja, joka näkyy esimerkiksi ohjelman suorituskyvyn paranemisena. Huonona puolena viitekehyksissä on, että ne ovat joskus todella monimutkaisia ja siten vaikeita oppia ja ottaa käyttöön. (Baker, M. 2009.)

Tässä projektissa esiintyvät seuraavat viitekehykset:

- PHPUnit on saksalaisen Sebastian Bergmannin kehittämä yksikkötestaus-viitekehys PHP-ohjelmointikielelle. Kuten moni muukin yksikkötestaustyökalu, se pohjautuu xUnit-arkkitehtuuriin, jossa käytetään varmennuksia (assertion) varmistamaan, että testattava koodi tekee sitä mitä pitääkin. (PHPUnit, 2013.)
- Zend Framework 2 on avoimen lähdekoodin olio-ohjelmointiviitekehys PHP-kielelle (PHP 5.3 ja uudemmat). Sitä käytetään web-pohjaisissa sovelluksissa, ja se käyttää MVC-arkkitehtuuria sekä olio-ohjelmoinnin periaatteita. Zend Frameworkin kirjasto on laajennettavissa helposti myös ulkopuolisten tahojen omilla koodeilla. (Zend Technologies Ltd 2014.)
- Doctrine on joukko PHP-kirjastoja, jotka helpottavat tietokantakyselyiden ohjelmointia kehitettävään ohjelmaan ORM-tekniikalla. ORM-tekniikka yksinkertaistaa ohjelman ja tietokannan välistä yhteydenpitoa ja tiedon muodon muuntoa, mikä edelleen nopeuttaa ohjelman suorittamista. Doctrinella on oma DQL-kieli (Doctrine Query Language), jolla tietokantakyselyt tehdään. Kyselyt on lyhyempi ohjelmoida verrattuna perinteiseen SQL-kieleen. (Doctrine Team, 2012; Jonssen, C.)

2.2 Testaaminen ja yksikkötestaaminen

Testauksella tarkoitetaan lähes mitä tahansa kokeilemista. Testaaminen voi olla suunnitelmallista tai etsivää, jossa ohjelmaa kokeillaan umpimähkäisesti joillakin arvoilla. Testauksen avulla voidaan osoittaa, että ohjelmassa on virheitä, mutta täydellistä virheettömyyttä ei voida koskaan luvata. Käytännössä ohjelmien toiminnoista pystytään testaamaan vain hyvin pieni osa. Testausta kannattaa kuitenkin tehdä, sillä sen avulla voidaan mitata ohjelman laadullista arvoa sekä lisätä luottamusta ohjelmaa kohtaan. (Graham, van Veenendaal, Evans & Black 2008, 7; Haikala & Mikkonen 2011, 205.)

Myersin ym. (2011, 13–15) mukaan ohjelmistotestaamisen periaatteisiin kuuluu, että ohjelman tuottavan organisaation tai ohjelmoijien ei tulisi testata omia ohjelmiaan. He eivät osaa tarkastella ohjelmaa tarvittavalla kriittisyydellä, ja varsinkin ohjelmoijat tulevat sokeiksi omalle koodilleen. Ohjelmoija on myös saattanut ymmärtää jonkin osan toiminnan eri tavalla kuin on ollut tarkoitus, eikä itse välttämättä havaitse tällaista virhettä. Yritys saattaa taas tuijottaa enemmän aikaa ja rahaa mitä testaamiseen kuluu, sekä päivämäärää, jolloin ohjelma olisi tarkoitus toimittaa tilaajalle.

Yksikkötestaamisen tavoitteena on varmistaa, että komponentit vastaavat ja reagoivat syötteisiin oikein. Yksikkötestaaminen on kaikkein tavallisin testausmuoto, ja sitä käytetään yleisesti kaikissa ohjelmisto-organisaatioissa. Siinä tarkastellaan yhden yksittäisen funktion tai metodin toimintaa tunnettujen syötteiden kanssa, ja verrataan tuloksia odotettuihin tuloksiin. Tätä kutsutaan assertioksi tai vakuutukseksi (assertion), eli vakuutetaan esimerkiksi, että a on yhtä kuin b. Vastauksena väitteeseen on joko tosi (true) tai epätosi (false). Seuraavassa esimerkki yksikkötestistä:

```
function sum($a, $b)
{
 return $a + $b;
}
$this->assertEquals(2, sum(1, 1));
```

Assertioita on useita erilaisia ja eri tilanteisiin sopivia. (Kasurinen 2013, 51. Machek 2014, 48.)

Yleensä yksikkötestit laatii itse ohjelmoija, mutta niiden laadintaan voidaan käyttää myös muuta henkilöä. Koska yksikkötestauksessa yleensä halutaan testata vain yksittäisen funktion toiminta, voidaan testausta helpottamaan rakentaa testikomponentteja tai testi-

tynkiä (mock objects, test stubs), joiden tehtävänä on simuloida järjestelmän osien välistä liikennettä. (Kasurinen 2013, 51–52.)

Machekin (2014) mukaan yksikkötestauksen hyötynä on, että niiden avulla kasvatetaan luottamusta tehtyä koodia kohtaan. Myös mahdollisen refaktoroinnin eli koodin uudelleenkirjoittamisen kannalta on hyvä, että koodia on testattu. Monimutkaisen koodin muuttaminen ilman testausta on kuin miinakenttään astuisi.

2.3 Kehitysympäristö projektissa

Kehitysympäristö oli virtuaalikoneessa, joka oli luotu Vagrantin ja Puppet-konfigurointityökalun avulla. Vagrant on ilmainen ja avoimen lähdekoodin työkalu virtuaalisten kehitysympäristöjen luontiin sekä konfiguroimiseen, jonka on luonut Mitchell Hashimoto vuonna 2010. Vagrant luo kehitysympäristön pitkälti automaattisesti ja nopeasti, kehittäjän tarvitsee vain antaa yksinkertainen käsky siinä kansiossa jonne kehitysympäristö halutaan asentaa. Samaa kehitysympäristöä on myös mahdollista jakaa muille, jolloin esimerkiksi kehitystiimin kaikilla jäsenillä on sama virtuaalinen kehitysympäristö. (HashiCorp, 2013a; HashiCorp, 2013b.)

Kehitysympäristöstä otettiin oma kopio testausta varten. Kehitysympäristö käynnistettiin ja suljettiin komentoriviltä. Testausta varten virtuaalikoneeseen asennettiin mukaan PHPUnit-viitekehys. PHPUnit-komennot ajettiin komentoriviltä.

Itse testejä varten luotiin oma Tests-kansio. Tällainen tehtiin jokaiselle ohjelma-alueelle (Core, Forum, SubCore) erikseen. Tests-kansion juureen luotiin kaksi konfiguraatiotiedostoa, joissa määriteltiin asetuksia PHPUnitille sekä ajettaville testeille. Kansioon perustettiin testeille varsinaisen ohjelman tyyliä muistuttava kansiorakenne, jotta halutut testit on helppo löytää. Toisin sanoen, esimerkiksi Controller-testit tallennettiin Controller-kansioon.

Testit kirjoitettiin PHP-kielellä käyttäen Sublime 2-ohjelmaa. Koska Guild Charter on luotu käyttäen Zend Framework 2 - ja Doctrine - viitekehyksiä, tuli niiden erityispiirteet ottaa huomioon testejä kirjoitettaessa.

3 Toteutus

Tässä luvussa kerrotaan tarkemmin opinnäytetyöprojektin etenemisestä. Siinä käsitellään aiheeseen perehtymistä, ympäristön käyttöönottoa sekä testauksen eri osioiden toteutusta.

3.1 Perehtyminen aiheeseen

Opinnäytetyöprojekti alkoi perehtymisellä PHPUnit-viitekehyksen asennukseen sekä testeihin. Niitä varten piti asentaa oma harjoitteluympäristö.

Composer on riippuvuusmanageri (dependency manager) eli eräänlainen ohjelmistokirjastojen asentaja. Asennus on helppoa, sillä kirjasto asennetaan kirjoittamalla kirjaston nimi ja haluttu versio Composerin asennustiedostoon, ja sen jälkeen ajamalla Composer. Composer osaa samalla kerralla ladata ja asentaa myös kirjaston mahdollisesti vaatimat muut kolmannen osapuolen lisäkirjastot samalla. PEAR on samanlainen työkalu kuin Composer, mutta kirjastot asennetaan kirjoittamalla komentoriville PEAR-käskyn jälkeen asennettavan kirjaston osoite. (Machek 2014, 12, 15–16.)

PHPUnitin asentamisen kanssa oli ongelmia jonkin verran ja niihin meni aikaa. Composer-asennustapa ei alkuun toiminut, koska osa tarvittavista paketeista ei asentunut. Asennus saatiin suoritettua PEARin avulla, joskaan PHPUnit ei enää tue sitä asennustapaa. Myöhemmässä vaiheessa PHPUnit saatiin asennettua Composerillakin. Testaukseen perehtyminen sujui hyvin tämän jälkeen.

Seuraavaksi piti opiskella Zend Framework 2 - alkeita, koska Guild Charter on pääasiassa sillä rakennettu. Uuden harjoitteluympäristön asentamisen kanssa oli ongelmia, ja niiden ratkaisuun meni kauan aikaa. Yhdeksi ongelmaksi paljastui lopulta viallinen asennustiedosto, ja uudella tiedostolla saatiin ympäristö asennettua. Ympäristön asetuksien kanssa meni myös hieman aikaa. Loppujen lopuksi harjoitusympäristö saatiin asennettua. Asiakkaasta oli suuri apu tässä vaiheessa.

Zend Frameworkiin perehtyminen sujui hyvin. Tämän jälkeen kehitysympäristöön asennettiin PHPUnit, jotta testausta voitiin harjoitella Zend Frameworkin kanssa. Tämä vaihe sujui myös hyvin.

3.2 Kehitysympäristön asennus ja Guild Charteriin perehtyminen

Testausta varten Guild Charterista otettiin oma kopio, jotta varsinainen kehitysversio ei sotkeentuisi, mikäli jotain menisi vikaan. Asennuksen kanssa oli jälleen ongelmia, esimerkiksi komentorivillä esiintyi erilaisia virheviestejä. Asennus saatiin lopulta päätökseen, jonkin verran meni vielä aikaa eri tietojen konfigurointiin.

PHPUnitia käynnistettäessä kävi ilmi, että ohjelmassa oleva väliaikaisratkaisu estää PHPUnitin toiminnan. Kyseessä oli web server-ympäristön komento, jota PHPUnit ei tue eikä ymmärrä. Komentoa muutettiin hieman PHPUnitin ymmärtämään muotoon, mutta tämä ratkaisu esti ohjelman toimimisen web-tasolla. Sillä ei koettu olevan niin väliä yksikkötestaamisen kannalta, joten työ jatkui tästä huolimatta.

Tässä vaiheessa tapahtui myös perehtyminen dokumentaatioon. Sitä oli muuten kiitettävästi, mutta valitettavasti vaatimusmäärittely-dokumenttia ei ollut. Vaatimusmäärittely on testauksen kannalta tärkeä, koska se kertoo mitä ohjelman odotetaan tekevän. Tilaajien kanssa sovittiin, että he kirjaavat koodiin jokaisen metodin kohdalle mitä niiden odotetaan tekevän. Lopputulema oli kuitenkin, ettei ihan joka metodissa ollut odotuksia kirjattu. Näiden metodien odotukset olivat siis testaajan arvioitavissa, tosin muutaman metodin suhteen ei täysin auennut mitä niissä tapahtuu.

3.3 Testaussuunnitelma ja testitapausten laadinta

Kasurisen (2013, 116) mukaan testaussuunnitelmalla tarkoitetaan yleisesti projektitasolla suunniteltua dokumenttia, jossa linjataan mitä ohjelmasta testataan, missä vaiheessa ja millä menetelmällä. Organisaatioissa saattaa olla testausstrategia ja/tai testauspolitiikka, johon testaussuunnitelma niissä perustuu. Testaussuunnitelma voi olla ainoastaan nippu yhteisesti sovittuja linjauksia, mutta tavallisesti se on edes jotenkin yhtenäinen dokumentti, joka sisältää testaustoiminnan pääkohdat.

Rubinin ym. (2008, 65) mukaan epämalkainen testaussuunnitelma on virhe, joka kostautuu myöhemmin. Testaussuunnitelma kuitenkin on dokumentti, joka määrittelee testauksen raamit ja koko kehitystiimin on tarkoitus saada siitä kaikki tarvitsemansa testaukseen liittyvä tieto.

ISO/IEC 29119-testausstandardin mukaan testaussuunnitelma sisältää muun muassa seuraavia asioita:

- kuvaus projektista
- kuvaus testattavasta tuotteesta

- testien laajuus eli mitkä osat testataan ja mahdolliset tiedossa olevat ongelmat
- käytetty testausstrategia, eli kuka, milloin, missä ja miten testaa
- aikataulu ja työnjako
- riskikartoitus mikäli tuotteeseen liittyy jotain merkittäviä riskejä
- toimintasuunnitelma miten vaatimusmäärittelyn vaatimukset todennetaan
- henkilöstölistaus testaajista ja kuka tekee mitäkin

(Kasurinen 2013, 117.)

Kasurisen (2013, 117–118) mukaan pienempiin yrityksiin ja projekteihin toisaalta saattaa sopia paremmin projektitasolle suunniteltu SPACE DIRT-menetelmä, jonka kirjaimet muodostavat lyhenteen testaussuunnitelman asioista:

- Scope = Laajuus, mitä testataan ja mitä ei
- People = Ihmiset, heiltä vaadittava koulutus, vastuut ja aikataulu
- Approach = Lähestymistapa, eli testausmenetelmät
- Criteria = Kriteerit eli aloitus-, lopetus-, keskeytys- ja jatkamiskriteerit
- Environment = Ympäristö, testausympäristö
- Deliverables = Tuotokset, mitä testausprosessi tuottaa
- Incidentals = Satunnaiset, erikoisominaisuudet ja poikkeukset joita testaamiseen liittyy, kuka voi muuttaa suunnitelmia
- Risks = Riskit ja niiden torjunta
- Tasks = Tehtävät, jotka kuuluvat testausprosessiin

Tässä projektissa testaussuunnitelma (Liite 1) tehtiin tälle projektille sopivaksi. Suunnitelmaan kirjattiin aluksi lyhyt Guild Charter-projektin esittely, sekä viittaus projektin kansiosta löytyviin ajantasaisiin lisätietoihin. Sen jälkeen on kerrottu vaatimuksista, kuka testaa ja missä ajassa, sekä lyhyt testaukseen vaikuttavien riskien kartoitus. Suunnitelmassa on myös selostettu miten testaaminen tehdään, aikataulu, budjetti, testausympäristö sekä aloitus- ja lopetusehdot.

3.4 Testitapausten laadinta

Testitapaus kuvaa yhtä tapahtumaa, jolla varmistetaan jokin ohjelman toiminnallisuus. Normaalisti testitapaukset kehitellään järjestelmän vaatimusmääritelmän tai arkkitehtuurimallin pohjalta, tavoitteena joko varmentaa että jokin vaadittu toiminnallisuus toteutuu, tai että jokin ominaisuus pysyy muuttumattomana. Testitapausten suunnittelu ja laadinta on tärkeää, koska ohjelmia ei voida testata jokaisen koodirivin osalta. Testitapauksia määritellään koko projektin elinkaaren ajan lisää jotta lopputuotteen laatu saadaan hyväksi. (Kasurinen 2013, 119; Myers ym. 2011, 41.)

Periaatteessa testitapauksen kuvaamiseksi riittää pelkkä kuvaus siitä, mitä testitapauksessa tulee tehdä ja miten järjestelmän tulee tähän reagoida. Tämä riippuu kuitenkin siitä millaista testausta varten tapaukset tehdään. Testitapauksiin voidaan myös kirjata muun

muassa kuka tapauksen on suunnitellut, työvaiheet sekä onko sillä riippuvuuksia muihin testitapauksiin. Testaajan on kuitenkin hyvä tietää mitä järjestelmän pitäisi tehdä jotta hän pystyy arvioimaan toimiiko kaikki oikein. (Kasurinen 2013, 120.)

Myersin ym. (2011, 18) mukaan hyvä testitapaus on sellainen, jonka avulla todennäköisesti löydetään virheitä. Menestyksekäs testitapaus taas löytää virheen, jollaisen olemassaoloa ei ennestään osattu kuvitella. Testitapauksiin täytyy määritellä myös selkeästi millaista lopputulemaa odotetaan sekä mahdolliset syötteet.

Tässä projektissa tehtiin hyvin yksinkertaiset testitapaukset. Testitapauksen nimi sisälsi mitä testataan, esimerkiksi testIndexActionCanBeAccessed = IndexAction-metodiin päästään (ja se vastaa) tai testGetTimezoneForCityReturnsCorrectValue = GetTimezoneForCity-metodi palauttaa odotetun arvon. Lisäksi kirjattiin minkä tiedoston metodia testi testaa, vaatimukset (mikäli niitä oli), koska testitapaus on luotu ja milloin se on mennyt läpi. Lisäksi kirjattiin minkä nimisessä testitiedostossa testitapaus on. Muunlaisia tietoja ei kirjattu, koska ne eivät tuoneet mitään lisäarvoa testiin.

Ohjelmistoprojektien ongelmaksi muodostuu helposti se, että testattavia asioita on paljon enemmän kuin mitä testausorganisaatio pystyy toteuttamaan. Käytännössä tämä tarkoittaa sitä, että luoduista testitapauksista joudutaan osa jättämään pois. Mukaan valittavat testitapaukset saatetaan valita esimerkiksi tilaajan toiveiden, riskien tai laatuvaatimusten mukaan. Testitapausten valinta ja priorisointi voidaan tehdä myös siten, että aloitetaan tärkeimmistä osista ja edetään kohti vähemmän kriittisiä osia. Tällöin ohjelmasta saadaan testattua kaikkein tärkeimmät ominaisuudet mitä saatavilla olevilla resurssimäärillä pystytään testaamaan. Kaikkia testitapauksia ei kuitenkaan koskaan ennätetä toteuttamaan, joten testaussuunnitelmaan kirjataan yleensä lopetusehdot (exit criteria), joissa määritellään milloin testaaminen loppuu. (Kasurinen 2013, 123–124.)

Koska testattava ohjelma on laaja ja testaajia tässä projektissa vain yksi, valittiin ohjelmasta mukaan kolme kriittisintä aluetta (Core, Forum, Subcore), joille testejä tehdään. Näistä edelleen testattaviksi osiksi valittiin controllerit, modelit ja repositoryt. Testauksen lopetusehtona oli aika, eli kun tähän opinnäytetyöprojektiin varatut tunnit loppuvat.

3.5 Testaaminen

Projektissa yksikkötestaaminen aloitettiin controller-luokista, joista testattiin, että ne vastaavat. Näiden testien perusteella voitiin todeta, että reititys on laadittu oikein ja eri metodeihin saa yhteyden. Tämän jälkeen testattiin model-luokkia. Niissä testattiin, että metodit

vastaavat ja palauttavat odotetut syötteet. Repository-luokan osalta testattiin tiedon hakua ja palautusta.

Testaamisen yhteydessä käytettiin paljon testikomponentteja simuloimaan järjestelmän muita osia. Testikomponentteja käytetään siksi, koska halutaan testata kerrallaan vain testattavana olevaa metodia, ja koska muut komponentit saattavat myös aiheuttaa testin aikana virheitä. Välillä käytettiin myös järjestelmässä valmiina olevia luokkia.

3.6 Testitulosten mittaaminen

On olemassa paljon erilaisia menetelmiä ja mittareita, joiden avulla pystytään seuraamaan testauksen etenemistä sekä arvioimaan tehtyjen testaustoimenpiteiden kattavuutta. Hyvä mittari on toistettava, tarkka, vertailukelpoinen ja taloudellinen. Mittarit voivat perustua raakaan testausdataan, työn suunnitteluun, projektin tilaan, kattavuuteen, testituloksiin, testauksen laatuun sekä luotettavuuteen. (Kasurinen 2013, 162.)

Mittarit valitaan projekteihin sen mukaan millaisia tietoja halutaan kerätä. Mittarin keräämien tietojen olisi kuitenkin hyvä muodostua projektiin joka tapauksessa tehdystä työstä. (Kasurinen 2013, 167.)

Tässä projektissa mittaamiseen käytettiin koodikattavuusanalyysia. Testausraporttien avulla esiteltiin, montako prosenttia laadituista testitapauksista on tehty ja millaiset tulokset niistä on saatu. Näitä esitellään tarkemmin kahdessa seuraavassa kappaleessa.

3.7 Testausraportit

Testausraporteilla on tarkoitus antaa tietoa projektin testaustoiminnan sujuvuudesta ylemmälle johdolle sekä kehittäjille. Niiden pohjalta voidaan myös tehdä päätöksiä mahdollisista muutoksista. Raportin tulee olla selkeä luettavuudeltaan ja siitä pitää saada helposti tarvittava tieto. Tärkein tieto on tulokset testauksesta, mutta raportissa voi olla myös lyhyesti testaustavasta sekä suosituksia tulosten pohjalta. Projektin päättyessä raporteista voidaan koostaa loppuraportti, johon kootaan tärkeimmät havainnot ja mittarit projektin etenemisestä. (Kasurinen 2013, 105; Rubin ym. 2008, 273–275.)

Testituloksista laadittiin raportit siten, että kunkin ydinalueen testien tuloksista tehtiin yhteenveto. Raportissa määriteltiin ensin suunniteltujen testitapausten määrä, ja sen jälkeen montako niistä on tehty. Tehtyjen testitapausten osalta määriteltiin, montako testiä menee läpi ja montako ei mene. Lopuksi on luku joka kertoo, montako suunnitelluista testitapauksista on tekemättä. Testausraportit löytyvät liitteestä 2.

3.8 Koodikattavuus

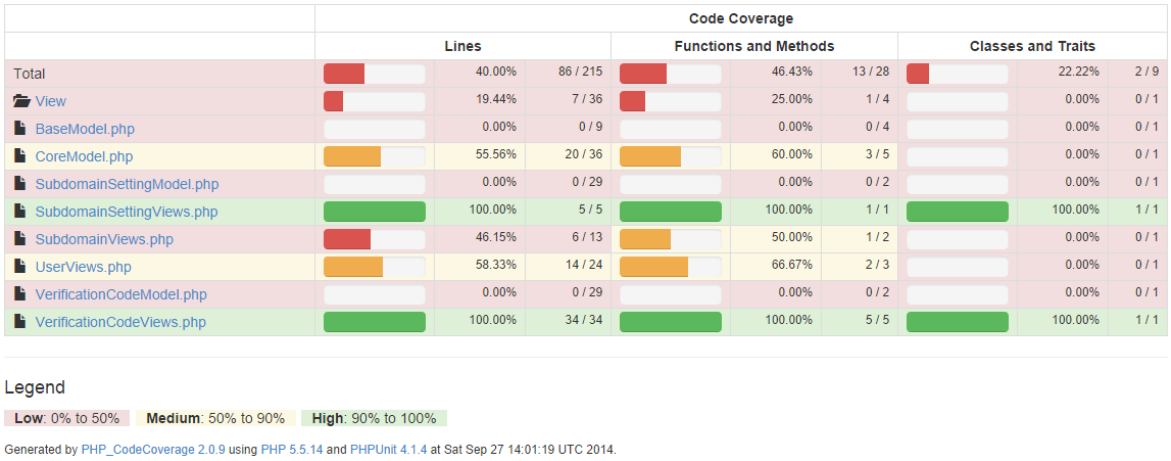
Koodikattavuudella (code coverage) selvitetään, mitkä osat koodista valmiit testit suorittavat sekä mitä osia ei suoriteta. Mitä suurempi kattavuus sitä pienempi on riski, että ohjelmassa esiintyisi virheitä, mikä osaltaan lisää luottamusta ohjelman laatua kohtaan. Koodikattavuutta selvitetään yleensä työkaluilla, joita on markkinoilla useita eri ohjelmointikielille. (Cornett, S. 2014; Graham ym. 2008, 107.)

Koodikattavuus mittaa ensisijaisesti testien laatua, ei ohjelman. Koodikattavuuden avulla pystytään huomaamaan mahdolliset turhat testit, jotka eivät lisää kattavuutta. Joskus yhtenä testaamisen lopetuskriteerinä on saavuttaa tietty prosenttiosuus kattavuuden suhteen. (Cornett, S. 2014.)

Cornettin (2014) mukaan koodikattavuuden arviointiin on useita erilaisia kriteereitä, joita esimerkiksi ovat:

- lausekattavuus (statement coverage): onko jokainen ohjelman lause (rivi) ajettu testissä
- päätöskattavuus (decision coverage): onko jokainen boolean-haara ajettu testissä (esimerkiksi if-lauseet)
- haarakattavuus (path coverage/branch coverage): onko ohjelman funktioiden jokainen haara ajettu testissä.

Tässä projektissa koodikattavuuden arviointi suoritettiin PHPUnitilla. PHPUnit tutkii lausekattavuutta, eli mitkä rivit ohjelmasta ajetaan testeissä. Kuviossa 1, jossa näkyy raportin yleisnäkymä, kerrotaan kuinka monta prosenttia kustakin tiedostosta kattavuus on. Totalluku kertoo kokonaiskattavuuden kaikkien kansiossa olevien tiedostojen osalta, ei koko ohjelman. Tiedoston nimeä klikkaamalla pääsee tarkastelemaan rivikohtaisesti tilannetta. Vihreällä värjätyt rivit on katettu, punaisella värjättyjä ei. Keltaisella on merkitty kuollut koodi, eli koodi jota ei ohjelmaa suorittamalla koskaan ajeta.



	Code Coverage								
	Lines			Functions and Methods			Classes and Traits		
Total		40.00%	86 / 215		46.43%	13 / 28		22.22%	2 / 9
View		19.44%	7 / 36		25.00%	1 / 4		0.00%	0 / 1
BaseModel.php		0.00%	0 / 9		0.00%	0 / 4		0.00%	0 / 1
CoreModel.php		55.56%	20 / 36		60.00%	3 / 5		0.00%	0 / 1
SubdomainSettingModel.php		0.00%	0 / 29		0.00%	0 / 2		0.00%	0 / 1
SubdomainSettingViews.php		100.00%	5 / 5		100.00%	1 / 1		100.00%	1 / 1
SubdomainViews.php		46.15%	6 / 13		50.00%	1 / 2		0.00%	0 / 1
UserViews.php		58.33%	14 / 24		66.67%	2 / 3		0.00%	0 / 1
VerificationCodeModel.php		0.00%	0 / 29		0.00%	0 / 2		0.00%	0 / 1
VerificationCodeViews.php		100.00%	34 / 34		100.00%	5 / 5		100.00%	1 / 1

Kuvio 1. Esimerkki koodikattavuusraportista.

Raportti kertoo myös jokaiselle tiedostolle CRAP-arvon (Change Risk Analysis and Predictions). Mitä suurempi CRAP-arvo on, sitä monimutkaisempaa koodi on. Monimutkainen koodi saattaa aiheuttaa ongelmia esimerkiksi mahdollisten muutosten yhteydessä ja sellaista on usein myös vaikeaa testata. CRAP-arvo pienenee mitä enemmän koodirivejä on testattu, mutta hyvin monimutkaiseen koodiin ei auta edes 100 % koodikattavuus. Monimutkaisiin koodiluokkiin suositellaankin refaktorointia eli koodin uudelleenkirjoittamista tai muokkaamista yksinkertaisemmaksi. (Savoia, A. 2007.)

Raportti ei kuitenkaan ole täysin virheetön. Punaiseksi on saatettu merkitä esimerkiksi rivi, jolla on ainoastaan yksi sulkumerkki. Tällaisia rivejä oli myös joskus merkitty keltaiseksi. Controller-testit, joilla testattiin vastaavatko metodit, eivät yleensä näkyneet raportin tuloksissa mitenkään. Raportti on kuitenkin suuntaa antava, mutta prosenttilukuja ei kannata liikaa tarkastella. Koodikattavuusraportit ovat liitteenä 3 ja niitä käsitellään lyhyesti myös liitteessä 2.

4 Yhteenveto

Projektin tavoitteet saavutettiin hyvin satunnaisista vaikeuksista huolimatta. Koska projektisuunnitelmassa osalle töistä oli arvioitu enemmän tunteja kuin mitä niihin loppupeleissä meni, oli projektissa tällöin hieman pelivaraa.

4.1 Tulokset

Projektissa saatiin aikaan testaussuunnitelma sekä sen pohjalta yksikkötestejä, sekä koodikattavuusraportteja ja testausraportteja. Viimeisin testausraportointi sekä koodikattavuusanalyysin tulokset ovat loppuraportissa, joka on liitteenä 2. Kaikki asetetut tavoitteet toteutuivat pääsääntöisesti.

Projektissa luotiin yhteensä 181 testitapausta, joista Core-alueelle 55, Forum-alueelle 86 ja Subcore-alueelle 40. Näistä toteutettiin lopulta yhteensä 125, joista Core-alueelle 41, Forum-alueelle 54 ja Subcore-alueelle 30. Toteutetuista testeistä kaikki menivät läpi, eli testattavat koodiosat toimivat.

Loppujen testitapausten toteutuksen tiellä oli ongelmia, joita ei saatu projektin aikana ratkaistua. Model-luokkien osalta suuri ongelma oli BaseModel-luokka, joka pysäytti testien ajamisen erilaisin virheviestein. Joissain tapauksissa puuttuvien koodikommenttien vuoksi jäi epäselväksi, mitä metodin on tarkoitus tehdä.

Controller-luokissa uudelleenohjausta testaavat testit epäonnistuivat. Tähän saattaa vaikuttaa kappaleessa 3.2 mainitun väliaikaisratkaisun poistaminen käytöstä. Tarkemmin toteutetuista ja toteuttamattomista testeistä on kerrottu liitteessä 2.

Core-alueella kokonaiskattavuus koodirivien osalta oli 22,73%. Koodikattavuus controller-luokkien osalta oli 2,53 % ja model-luokkien osalta 41,86 %. Koska controller-luokkien toteutuneissa testeissä testattiin metodien vastaamista, niitä ei lueta mukaan kattavuuteen, ja tästä syystä controller-luokkien kattavuus on kaikkien alueiden osalta olematon. Repository-luokan koodirivien kattavuus oli 98,61%.

Yhteenveto Core-alueen koodikattavuudesta on kuvattu kuviossa 2. Kuviossa on esitetty koodirivien lisäksi kattavuus metodien (functions and methods) sekä luokkien (classes and traits) osalta. Prosenttilukujen vieressä on numerot kappalemääränä, esimerkiksi 120 koodiriviä 477:stä on testattu.

	Code Coverage								
	Lines			Functions and Methods			Classes and Traits		
Total		22.73%	173 / 761		44.07%	26 / 59		21.05%	4 / 19
Controller		2.53%	12 / 474		18.18%	4 / 22		0.00%	0 / 7
Model		41.86%	90 / 215		50.00%	14 / 28		22.22%	2 / 9
Repository		98.61%	71 / 72		88.89%	8 / 9		66.67%	2 / 3

Kuvio 2. Yhteenveto Core-alueen koodikattavuudesta.

Forum-alueella kokonaiskattavuus koodirivien suhteen oli 18,55%. Koodikattavuus controller-luokkien osalta oli 0,0 % ja model-luokkien osalta 20,19 %. Repository-luokan koodirivien kattavuus oli 91,57%. Tarkempi yhteenveto Forum-alueen koodikattavuudesta on esitetty kuviossa 3.

	Code Coverage								
	Lines			Functions and Methods			Classes and Traits		
Total		18.55%	271 / 1461		26.19%	22 / 84		25.00%	6 / 24
Controller		0.00%	0 / 748		0.00%	0 / 22		0.00%	0 / 8
Model		20.19%	108 / 535		26.67%	12 / 45		18.18%	2 / 11
Repository		91.57%	163 / 178		58.82%	10 / 17		80.00%	4 / 5

Kuvio 3. Yhteenveto Forum-alueen koodikattavuudesta.

Subcore-alueella kokonaiskattavuus koodirivien osalta oli 25,16%. Controller-luokkien osalta koodikattavuus oli 0,0 % ja model-luokkien osalta 66,33 %. Repository-luokan koodirivien kattavuus oli 88,71%. Tarkempi yhteenveto Subcore-alueen koodikattavuudesta on esitetty kuviossa 4.

	Code Coverage								
	Lines			Functions and Methods			Classes and Traits		
Total		25.16%	120 / 477		39.39%	13 / 33		41.67%	5 / 12
Controller		0.00%	0 / 317		0.00%	0 / 15		0.00%	0 / 4
Model		66.33%	65 / 98		69.23%	9 / 13		60.00%	3 / 5
Repository		88.71%	55 / 62		80.00%	4 / 5		66.67%	2 / 3

Kuvio 4. Yhteenveto Subcore-alueen koodikattavuudesta.

Joissakin tiedostoissa oli hyvinkin korkea CRAP-arvo, joiden suhteen kehittäjien kannattaisi harkita refaktorointia. Korkeimmillaan CRAP-arvo oli eräässä tiedostossa 3906. Missään luotettavissa lähteissä ei kerrottu suositeltavaa lukua arvon suhteen, epävirallisemmissa suositeltiin alle 50. Tarkemmat koodikattavuustulokset ovat luettavissa liitteessä 3.

4.2 Projektin arviointi

Oli hankalaa etukäteen arvioida, minkä verran mihinkin työvaiheeseen menisi aikaa. Aika riitti kuitenkin hyvin satunnaisista hankaluuksista huolimatta. Aikaa riitti myös sellaisiin työvaiheisiin, joita ei alun perin huomattu laskea mukaan aikatauluun tai ollut muuten suunniteltu.

Projektin aikataulu petti syksyn aikana. Syynä tähän olivat sairastuminen, työssäkäynti sekä satunnaiset muut pakolliset menot. Onneksi aikataulua ei alun perinkään laadittu kovin tiukaksi eikä asiakkaalla ollut kiirettä asian kanssa. Projekti kuitenkin eteni koko ajan, etenemisvauhti ainoastaan vaihteli.

Yhteistyö projektin asiakkaiden kanssa oli sujuvaa. Yhteyttä pidettiin myös ohjauskokousten ulkopuolella ja asiakkaiden apua sai aina tarvittaessa. Varsinkin alkuvaiheessa asiakkaiden apu oli todella tarpeen asennuksien yhteydessä. Loppuvaiheessa asiakkaiden apua ei enää tarvittu. Ohjauskokoukset pidettiin Skype-puheluiden välityksellä, mikä toimi erittäin hyvin.

Suurin haaste projektissa oli opetella uusi testausviitekehys ja -tekniikka, ja soveltaa niitä ohjelmaan, joka käyttää muita hankalia ja ennestään tuntemattomia viitekehyksiä. Vaarana oli, että projektissa ei saada mitään aikaan. PHPUnitin opettelu ja Guild Charteriin perehtyminen kuitenkin onnistui, ja projektissa saatiin aikaan se, mitä alun perin oli suunniteltukin.

Opinnäytetyön tekijä oppi projektin aikana laatimaan testejä ja koodikattavuus-raportteja PHPUnitilla, sekä sai kokemusta testaussuunnitelman ja testiraporttien tekemisestä.

Lähteet

Baker, Mike. 2009. What is a Software Framework? And why should you like ‘em? Luettavissa: http://info.cimetrix.com/blog/bid/22339/What-is-a-Software-Framework-And-why-should-you-like-em. Luettu: 28.9.2014.

Brown, Titus. 2006. An Extended Introduction to the nose Unit Testing Framework. Luettavissa: http://ivory.idyll.org/articles/nose-intro.html. Luettu: 28.9.2014.

Cornett, Steve. 2014. Code Coverage Analysis. Luettavissa: http://www.bullseye.com/coverage.html. Luettu: 28.9.2014.

Doctrine Team. 2014. Getting Started With Doctrine. Luettavissa: http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/tutorials/getting-started.html. Luettu 30.9.2014.

Graham, D., van Veenendaal E., Evans I. & Black R. 2008. Foundations of Software Testing. Cencage Learning EMEA. London.

Haikala, I. & Mikkonen, T. 2011. Ohjelmistotuotannon käytännöt. Talentum. Helsinki.

HashiCorp. 2013a. About Vagrant. Luettavissa: https://www.vagrantup.com/about.html. Luettu: 28.9.2014.

HashiCorp. 2013b. Why Vagrant? Luettavissa: https://docs.vagrantup.com/v2/why-vagrant/index.html. Luettu: 28.9.2014.

Jonssen, C. What is Object-Relational Mapping (ORM). Luettavissa: http://www.techopedia.com/definition/24200/object-relational-mapping--orm. Luettu: 2.7.2014.

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

Machek, Z. 2014. PHPUnit Essentials. Packt Publishing Ltd. Birmingham.

Myers, G., Sandler, C. & Badgett, T. 2011. The Art of Software Testing. 3. uudistettu painos. Wiley. Hoboken, New Jersey.

PHPUnit. Luettavissa: http://www.phpunit.de. Luettu: 31.8.2014.

Rubin, J., Chisnell, D. & Spool, J. 2008. Handbook of Usability Testing. 2. uudistettu painos. Wiley. Indianapolis.

Savoia, Alberto. 2007. Pardon My French, But This Code Is C.R.A.P. (2). Luettavissa: http://www.artima.com/weblogs/viewpost.jsp?thread=210575. Luettu: 28.9.2014.

Thibodeau, P. 2002. Study: Buggy software costs users, vendors nearly $60B annually. Luettavissa: http://www.computerworld.com/article/2575560/it-management/study--buggy-software-costs-users--vendors-nearly--60b-annually.html. Luettu: 29.10.2014.

Zend Technologies Ltd. 2014. Luettavissa: http://framework.zend.com/about/. Luettu: 31.8.2014.

Liitteet

Liite 1. Testaussuunnitelma (salainen)

Liite 2. Testauksen loppuraportti (salainen)

Liite 3. Koodikattavuusraportit (salainen)