HAAGA-HELIA
ammattikorkeakoulu

PHP-pohjaisen sovelluksen testaus

Viktor Marttinen

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2010

HAAGA-HELIA
ammattikorkeakoulu

Tiivistelmä

27.10.2010

Tietojenkäsittelyn koulutusohjelma

Tekijät Viktor Marttinen	**Ryhmä tai aloitusvuosi** TIKO06DI
Opinnäytetyön nimi PHP-pohjaisen sovelluksen testaus	**Sivu- ja liitesivumäärä** 46 + 8
Ohjaaja tai ohjaajat Sirpa Marttila	

Tämä opinnäytetyö käsittelee, kuinka nykypäivänä toteutetaan web-sovellusten testausta. Opinnäytetyö tehtiin CASE-tutkimuksena verkkopeli, mikä on koodattu PHP kielellä. Verkkopelille ei ollut laadittu testaussuunnitelmaa eikä toteutettu yksiköntason testausta.

Tämän raportin tavoitteena oli tutkia asioita teoriassa, mitä piti käsitellä ja ratkaista kun suunnitellaan web-sovelluksen testaussuunnitelmaa ja erityisesti yksikkötestaukset, soveltaa niitä CASE-esimerkiin ja analysoida niitä, selvittää mihin täytyisi kiinnittää huomion kun suunnitellaan ja kehitetään yksikkötestaukset PHP ja muussa web-ympäristössä.

Tämän raportin viitekehysosuudessa tarkasteltiin teoriaa joka koskee: testausprosessia osana ohjelmistotuotantoa ja testausprosessin vaiheita. Työssä käsiteltiin testauksen menetelmiä, erityisesti yksikkötestauksia ja käytiin läpi lyhyesti testausten mittarit ja automatisointi. Empiirisessä osassa tutustuttiin XAMPPiin, NetBeansiin ja PHPUnitiin, ja toteutettiin muutama yksikkötestitapaus. Pohdinnassa havaintojen ohella nivottiin yhteen teoria- ja empiirisen osion ja mietittiin johtopäätöksestä.

Aineistoa kerättiin usealla eri tavalla. Tietoa hankittiin kirjallisuudesta, seminaareista ja kursseilta. Tietoa haettiin myös testausympäristöstä, jolla pyrittiin toteuttamaan muutama yksikkötason testitapaus. Menetelmänä valittiin tyypillisimmin ilmiöiden kuvaileminen ja toistaminen testausympäristössä ja niiden myöhempi analyysi ja arviointi.

Kokemuksena voidaan mainita, että web-pohjainen sovellusten testaus eroa tavallista sovellustestausta. Itse yksikkötestit eivät paljon poikkea testausta mikä tahansa muu sovelluksentestausta. Selvittiin että kuin kaikki testauksen tekniikat, yksikkötestaus ei havaitse kaikkia ohjelman virheitä. Yksikkötestaus on tehokkaampaa, jos sitä käytetään muiden testausmenetelmien yhteydessä. Saadakseen hyötyä yksikkötestauksessa on ehdottomasti noudatettava testauksen teknologian sekä laadittu lähdekoodin dokumentaatio koko ohjelmiston kehitysprosessin läpi. Tuotu prosessimalli soveltuu myös muille verkkopeli-projektin osille tai jopa muille PHP-pohjaisille sovelluksille.

Asiasanat
Testaus, verkko-ohjelmointi, PHP

HAAGA-HELIA
ammattikorkeakoulu

Abstract

27.10.2010

Degree Programme in Business Information Technology

Authors Viktor Marttinen	**Group or year of entry** TIKO06DI
The title of thesis PHP-based application testing	**Number of pages and appendices** 46 + 8
Supervisors Sirpa Marttila	

The purpose of this thesis was to clarify how web-based applications are tested today. The study focused on investigating some network games application, which is coded in the PHP language. The network games did not have their own testing plan and their unit-level testing had not been implemented.

The frame of reference of this study examined the theory of testing process in software production and testing phases. The theoretical part discussed testing methods, especially the unit testing, and it briefly went through the testing measurement and automation. In the empirical part, XAMPP, NetBeans and PHPUnit were explored, and a few unit test cases were executed.

Information was gathered from literature, seminars and workshops. Also, information was gained from testing environments, in which a few unit-level test cases were implemented.

The study indicated that testing web-based applications differs from normal application testing. However, unit tests themselves are not much different from any other application tests. It became evident that like all the testing techniques, unit testing alone does not reveal all the errors in the program either. Thus, unit testing is more effective if it is used together with other test methods.

The study concludes that, in order to get benefits from unit testing, it is essential to implement it simultaneously with other testing technologies and provide source code documentation throughout the software development process. The testing process model produced in this study is also applicable to other parts of the network games application and even to other PHP-based applications.

Key words
Testaus, verkko-ohjelmointi, PHP

Sisällys

1 Johdanto 1
1.1 Käsitteet 1
1.2 Tausta 3
1.3 Tavoitteet ja menetelmät 4
1.4 Rajaukset 4
2 Viitekehys 5
2.1 Strateginen Web-ohjelmiston testauksen lähestymistapa 5
2.1.1 Yleiset virheet sisältyviä web-sovelluksen ympäristössä 5
2.1.2 V-malli 6
2.1.3 Yksikkö testaus 7
2.1.4 Integraatio testaus 9
2.1.5 Regressiotestaus 11
2.2 Testauksen taktiikka 12
2.2.1 Mustalaatikko menetelmä 12
2.2.2 Valkoisenlaatikko menetelmä 14
2.3 Testauksen mittarit 19
3 Empiirinen osa 22
3.1 Kohteen esittely 23
3.1.1 Verkkopelin testauksen nykytila 24
3.2 Aineisto 25
3.3 Menetelmät 25
3.3.1 Testausympäristö 27
3.3.2 Yksikkö testausten määrän arviointi, PHP_Depend 29
3.3.3 PHPUnitin asetukset 31
3.3.4 Testitapauksien suunnittelu PHPUnitilla 32
3.4 Testausprosessi 34
3.4.1 Testitapaus 1 35
3.4.2 Testitapaus 2 35
3.4.3 Testitapaus 3 36
3.4.4 Testitapaus 1 (jatku) 39
3.5 Tulokset 40
3.6 Arviointi 41
4 Pohdinta 44

4.1 Yhteenveto 44
4.2 Johtopäätökset 45
4.3 Jatkokehityksen ehdotukset 46
4.4 Oman kokemuksen ehdotukset 46
Lähteet 47
Liitteet 49
Liite 7. Loppuraportti 1
1 Tausta 1
2 Saavutetut tulokset 1
3 Työn eteneminen 2
4 Kustannukset 2
5 Resurssien käyttö 2
6 Kokemukset 3
7 Ehdotus jatkotoimenpiteiksi 3
8 Suositukset toimintatapojen muuttamiseksi 3
Liite 1 Ajoitussuunnitelma 5
Liite 2 Toteumaseuranta 1

1 Johdanto

1.1 Käsitteet

DAO luokka (data access object)

> DAO tarjoaa rajapinnan jonkinlaiseen tietokanaan joka tarjoaa joitakin erityisiä toimia ilman paljastamista tietokannan yksityiskohtia. Data Access Objectin käytön etu on suhteellisesti yksinkertaisen erottaminen toisistaan kaksi tärkeää sovelluksen osia, jotka mahdollisesti pitäisi tietää mitään toisistaan. (Bergmann 2010.)

Luokka (class)

> Luokka on olio-ohjelmoinnin keskeinen käsite. Luokka on olion malli, käyttäjän määrittämä tietotyyppi, joka sisältää muuttujat, ominaisuudet ja menetelmät. (Pressman 2005, 271.)

Metodi (method)

> Menetelmä on proseduuripohjaisen lausuen joukko halutun tuloksen saavuttamiseksi. Se tekee erilaisia toimia eri tietotyyppien kanssa. (Pressman 2005, 546.)

Moduuli (module)

> Moduulit ovat tyypillisesti rakennettu ohjelmaan rajapinnan kautta. Rajapinnassa määritelty elementit ovat vaadittavia muista ohjelman moduuleista. Modulointi on keino joka yksinkertaistaa ohjelman suunnittelua ja kehittämistä. Yleisesti moduuliksi kutsutaan ohjelman osa, esim. luokka tai web-sivu tai n. 1000 rivien koodin kokonaisuus. (Haikala & Märijärvi 2004, 280.)

Olio (object)

Olio on kokonaisuus tietokonejärjestelmän muistiosoitteessa, joka ilmestyä, kun luodaan luokan ajokappale, esim. lähdekoodin ajamisen jälkeen. (Pressman 2005, 213.)

Refaktorointi (refactoring)

Refaktoroinnin tarkoitus on muokkaa lähdekoodia helpompi ymmärtäväksi ilman kosketa sen ulkoista funktionaalisesta käyttäytymisestä. Edut ovat parantuneet koodin luettavuus ja vähäinen monimutkaisuus jotka yhteensä parantavat sen ylläpidettävyyttä. (Pressman 2005, 112.)

Järjestelmätestaus (System testing)

Järjestelmän testaus on itse asiassa useita erilaisia testejä joiden alkuperäinen tarkoitus on täydellisesti testata atk-pohjaisen järjestelmän. Vaikka jokaisella testien eräillä ovat omat tavoitteet, kaikki testaustyöt varmistavat että järjestelmän osat ovat asianmukaisesti integroitu ja oikeasti jaettu omiin toimintoihin. (Lehtimäki 2006, 171–174.)

1.2 Tausta

Tämä opinnäytetyö fokusoi tutkimaan asiaa kuinka nykypäivänä toteutetaan web-sovellusten testausta. Ohjelmistojen testaus on olennainen osa ohjelmistokehitystä. Uudelleenkäytettävien moduulien kehittämisen ja kerrosarkkitehtuurien myötä testauksessa on korostunut virheiden etsimisen ohella ohjelmiston laadunvarmistuksen ja lähdekoodiin tehtävien muutoksien hallinnan tärkeys. Ohjelmistojen testausta voidaan käsitellä eri tasoilla sen mukaan, mitä osaa tai kokonaisuutta siitä testataan. Yksikkötestaus kohdistuu ohjelmiston pienimpiin osiin ja sen avulla ohjelmiston yksittäisten toiminnallisuuksien oikea ja virheetön toiminta voidaan varmistaa. Testausta joudutaan usein suorittamaan uudelleen kehityksen yhteydessä, jolloin puhutaan regressiotestauksesta. Testitapausten automatisointi nopeuttaa testien uudelleensuorittamista.

Tämän työn soveltamiskohdealueeksi valittiin eräs selainpohjainen php-kielellä kirjoitettu verkkopeli ja sen toinen osa. Verkkopeli tuottaa uuden virtuaalisen simulaation kansainvälisen liiketoiminnan opiskelijoita varten yhdistäen hotelli-, ravintola- ja matkailualan. Verkkopelin peli II toimii verkon yli ja antaa verkkopelaajille mahdollisuuden osallistua peleissä. Syy miksi on kirjoitettu tämä työ, on se, että verkkopelille ei ole vielä laadittu testaussuunnitelmaa, vaikka integraatio tason testaus on jo suunniteltu ja toteutettu. Siksi ruvetaan käymään läpi asioita, jotka voivat koskea jonkin verran web-pohjaisten ja erityisesti PHP-kielien sovellusten testausta ja pyritään räätälöimään niitä verkkopelien testaussuunnitelmassa. Verkkopelissä ei ole suunniteltu eikä toteutettu yksikkötasontestaus, siksi tässä raportissa empiirisessä luvussa keskitytään pääosin yksikkötestaukseen, vaikka muut tyyppiset testaukset myös käydään rinnakkaisesti läpi.

Koska tämän työn case esimerkiksi on valittu PHP-pohjainen sovellus, kaikki empiirisessä luvussa käsiteltävät ja etenevät luvut ovat juuri PHP-kielin kirjoitettu testausympäristön sovelluksille, joiden päätyökaluna käytettiin PHPUnitin frameworkki. PHPUnit on PHP-ohjelmointikielellä kehitetty testausympäristö, jolla voidaan suorittaa ohjelmoituja testitapauksia. Sillä on sitten mahdollista testata PHP-kielillä toteutettuja ohjelmia. Opinnäytetyössä keskitytään PHP-kielellä kirjoitettujen Web-ohjelmaosien yksikkötestaamiseen, vaikka käydään läpi myös muita testauksen tyyppejä ja menetelmiä. Tämä voi auttaa tulevaan testaussuunnitelman tekemistä. PHPUnit tarjoaa PHP-kielelle apukirjastoja testitapausten ohjelmointiin sekä ajoympäristön testitapausten suorittamiseen ja raportointiin.

Opinnäytetyö perehdyttää lukijan yksikkötestauksen merkitykseen, tavoitteisiin ja prosesseihin yleisellä tasolla. Lisäksi käsitellään yksikkötestausta, testauksen toteuttamista osana Web-ohjelmistoprosessia. Lukijalta odotetaan perustietämystä olio-ohjelmoinnista ja Web -ohjelmistotuotannosta.

Opinnäytetyön toisessa luvussa tarkastellaan teoriaa joka koskee seuraavia asioita: testausprosessia osana ohjelmistotuotantoa, testausprosessin vaiheita. Käsitellään testauksen menetelmiä, erityisesti yksikkötestauksia ja lyhyesti käydään läpi testausten mittareita ja automatisointia. Opinnäytetyön kolmannessa luvussa perehdytään XAMPP, NetBeans ja PHPUnit, on valittu case esimerkki verkkopeli osa 2 testausympäristössä ja yksikkötestitapauksen ohjelmointiin PHPUnitin apukirjaston avulla. Luvussa 4 opinnäytetyön havaintojen ohella pohditaan PHPUnit-testausympäristön soveltuvuutta web-sovellusten yksikkötestaukseen. Lisäksi käsitellään lyhyesti tämän opinnäytetyön vaihtoehtoiset jatkotoimenpiteet.

1.3 Tavoitteet ja menetelmät

Opinnäytetyön kohdealue on yksikkötestauksen suunnittelu ja toteuttaminen, ja koska tämän työn case esimerkkinä on valittu PHP-pohjainen sovellus, nostetaan esille asioita joiden avulla perehdytään toteuttamaan yksikkötestaus erityisesti tässä valitussa ympäristössä. Tämän raportin tarkoitus on haravoida asioita teoriasta, joita piti käsitellä ja ratkaista, kun suunnitellaan web-sovelluksen testaussuunnitelmaa ja erityisesti yksikkötestauksia. Siksi tämän työn sivutavoitteena on myös laatia verkkopelille testaussuunnitelmaa. Sitten tarkastellaan asiaa testaajan näkökulmasta toteuttamalla muutamia tätä työtä varten testausympäristössä laadittuja testejä, tutkitaan analysoimalla niitä, miten saadaan aikaan yksikön testaus ja etsimällä valittuja testausympäristössä ja itse asiassa PHPn yksikkötestauksessa heikkoja ja vahvoja puolia. Sitten pohdinnassa käsitellään tulosten perusta ja pyritään etsiä erityiset seikat, mihin täytyisi kiinnittää huomio, kun suunnitellaan ja toteutetaan PHPn ympäristössä yksikkötestaukset, sekä itse asiassa PHPn yksikkötestien kannattavuus. Ja lopussa annetaan suosituksia, miten voidaan toteuttaa menestyksellistä testausta PHPn ja muun web-sovellusten kohtiin.

1.4 Rajaukset

Tämän raportin tarkoitus ei ole tuottaa täydellinen yksikkötestaus case valittuna esimerkkinä vaan tuoda muutama aikatauluun rajattu yksikkötestaukset PHP-kielien ohjelmistoprojektissa alusta loppuun ja tutkia miten voidaan toteuttaa yksikkötestauksen suunnittelu niin, että testauksen tulos voi arvioida hyväksyttäväksi.

2 Viitekehys

2.1 Strateginen Web-ohjelmiston testauksen lähestymistapa

Testaus on toimintojenjoukko, joka voidaan suunnitella etukäteen ja tehdä sitä järjestelmällisesti. Tästä syystä ohjelmistojen testauksen malli on toimenpiteen joukko, joihin voidaan sijoittaa tiettyjä testitapausten suunnitteluja sekä testauksen menetelmiä, jotka on määriteltävä ohjelmiston prosessin alussa. Useita ohjelmiston testausmenetelmiä on ehdotettu kirjallisuudessa. Kaikki tarjoavat ohjelmiston kehittäjälle erilaisia testauksen malleja ja seuraavia yleisiä ominaisuuksia:

- Tehdään tehokkaasti ohjelmiston testausta, vaatii joukolta toteuttaa tehokkaita teknisiä tarkistuksia. Näin voisin paljasta enemmän virheitä ennen testauksen aloittamista.
- Testaus alkaa komponentin tasolla ja liikkuu ulospäin yhdentymällä koko tietokonepohjaisen järjestelmää.
- Erilaiset testaustekniikat ovat asianmukaisia eri ajankohdissa ja paikoissa.
- Mielellään jos testaus suoritetaan ohjelmistokehittämisjoukossa, tai jos on suuria hankkeita, riippumattomassa testiryhmässä.

Ohjelmistotestaamisen strategian on pakko ehdota alemman tason testejä jotka tarvitsevat sen varmistamiseksi pieni lähdekoodin segmenttejä pantu täytäntöön oikein, ja korkean tason testejä jotka vahvistavat miten suurten järjestelmän toiminnallisuus vastaa asiakkaan vaatimuksiin. Koska eri vaiheet testausstrategiassa tapahtuvat samana aikana kun määräajan paine alkaa jatkuvasti nousua, edistys on mitattavissa, ja ongelmakohdat ovat havaituttava mahdollisimman pian. Tärkeitä strategisen päämäärät vastaavat seuraaviin kysymyksiin:

- Vahvistus: Olemmeko me rakentamassa tuote oikein?
- Varmistus: Olemmeko me rakentamassa oikea tuote?

(Pressman 2005, 387 – 388.)

2.1.1 Yleiset virheet sisältyviä web-sovelluksen ympäristössä

Virheet havaittu web-sovelluksen testauksessa voidaan luokitella seuraavasti:

1. Koska monet web-sovellusten testeistä paljastettu ongelmia, tavataan asiakkaan puolella (asiakas-palvelin ympäristössä), testaaja näkee virheen oire, ei virheen itse.

2. Koska Web-sovellukset usein toteutetaan erilaisessa kokoonpanoissa ja ympäristössä voi olla vaikeaa tai joskus jopa mahdotonta simuloida virheen ympäristön ulkopuolella jossa virhe oli alkuperäisesti kohdattu.
3. Vaikka jotkut virheet johtuivat virheellisestä suunnittelusta tai väärin HTML:stä (tai muu ohjelmointikielistä), monet virheet voidaan jäljittää web-sovelluksen kokoonpanoon.
4. Koska web-sovellukset toimivat asiakas-palvelin arkkitehtuurissa, virheet ehkä vaikea jäljittää kolmessa eri arkkitehtuurin kerroksessa: asiakas, palvelin ja verkkoon itse.
5. Jotkut virheet johtuivat staattisen toimintaympäristöstä (eli erityisestä kokoonpanosta, jossa testaus toteutetaan), kun taas toiset johtuivat dynaamisen toimintaympäristöstä (eli hetkellisen resurssien latausta tai aikaan liittyvin virheistä).

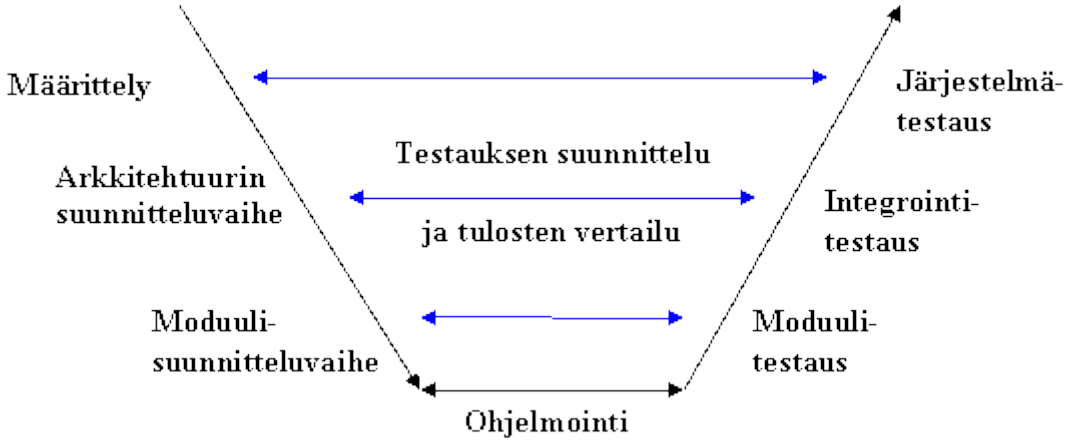
(Pressman 2005, 596 – 597.)

2.1.2 V-malli

Tässä mallissa erilaisia testaustasoja ovat ns. V-mallin mukaisia kuten nimittäin moduulitestaus, integrointitestaus ja järjestelmätestaus. Todellisesti testit ajetaan käännetyssä järjestyksessä suhteellisesti miten ne on suunniteltu, esim. yksikkö testit ovat kirjoitettu vasta loppuun mutta ajetaan ensin. Järjestelmätestausta voi joskus seurata erillinen hyväksymistestaus. Siirryttäessä V-mallin alimmalta tasolta ylöspäin testauksen luonne muuttuu lasilaatikko-testauksesta tavallisesti yhä enemmän mustalaatikkotestaukseksi. (OAMK 2005.)

V-mallin mukaisesti testauksen suunnittelu tapahtuu testaustasoa vastaavalla suunnittelutasolla (Kuvio 1). Määrittelyvaiheessa kuvataan ohjelmiston toiminnot, toteutukselle asetettavat ulkoiset vaatimukset sekä rajoitukset. Järjestelmätestaus suunnitellaan määrittely-vaiheessa. Arkkitehtuurin suunnitteluvaiheessa järjestelmä jaetaan mahdollisimman itsenäisiin, toisistaan riippumattomiin osiin, moduuleihin. Tähän vaiheeseen kuuluu integrointitestauksen suunnittelu. Moduulitestaus suunnitellaan moduulisuunnitteluvaiheessa, jossa jokaisen moduulin sisäinen rakenne suunnitellaan. Testauksesta syntyviä tuloksia verrataan näissä vaiheissa tapahtuneissa suunnitteluissa syntyneisiin dokumentteihin. (OAMK 2005.)

Näiden tasojen lisäksi voidaan suorittaa hyväksymistestaus, joka voi ehkä asiakkaiden ja käyttäjien tekemä testi. Tällä varmistetaan, että järjestelmä on valmis otettavaksi käyttöön. Vaiheen testaustapaukset on hyvä suunnitella asiakkaan kanssa mahdollisimman nopeasti vaatimusmäärittelyn teon jälkeen. (OAMK 2005.)



Kuvio 1. Testauksen V-malli (Kautto 1996)

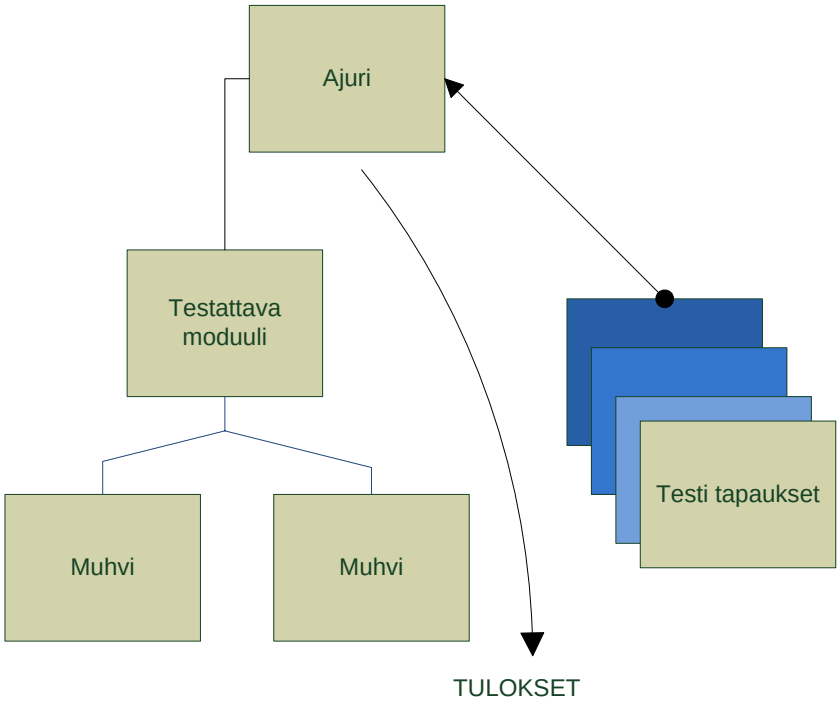
2.1.3 Yksikkö testaus

Kunkin moduulin rajapinnan testaus vaaditaan ennen muiden testien aloittamisesta. Jos tiedot eivät tulevat sisään ja poistuvat kunnolla, kaikki muut kokeet ovat kiistanalaisia. Testitapaukset pitäisi paljastaa muun muassa seuraavia virheitä, kuten:

- vertailu eri tietotyyppiin
- väärä loogisia operaattoreita tai funktiota
- odotuksia tasa-arvosudesta kun virheiden tarkkuuden takia samanarvoisuus tuntuu epätodennäköistä
- virheellinen muuttujien vertailu
- väärin tai olematon silmukan päättyminen
- ei poistu kun erilaiset iteraatiot kohdataan
- väärin muutettu silmukan muuttujat

Reunan testaus eli *boundary testing* on yksi tärkeintä yksikkötestauksen tehtävistä. Ohjelmistot usein epäonnistuivat sen rajojen takia. Testaus usein johtaa virheeksi kun n elementti n:lta matriisilta on käsitelty, jolloin suurin tai pienin sallittu arvo on kohdistettu tai ylittyy. Testitapauksia jotka käyttävät tietorakennetta, valvovat virtausta tai maksimien ja minimien tietojen arvoja alapuolella, tasaan tai hieman yli, ovat hyvin todennäköisesti paljastavat virheitä. Valitettavasti on taipumus sisällyttää virheenkäsittely itseen ohjelmistoon sisällä ja ei koskaan testata sitä. Yksikkötestaus pidetään yleensä integroitu osana koodaus vaiheessa. Yksikkö testien suunnittelu voidaan tehdä ennen koodaus alkua (mieluummin ketterä lähestymistapa) tai vasta silloin

kun lähdekoodi on luotu. Jokainen testitapaus oltaisi yhdistettävä odotettu tulosten joukkoon. Koska moduuli ei ole erillinen ohjelma, *ajuri* (driver) ja / tai *muhvi* (stub) -ohjelmisto on kehitettävä kunkin yksikön testiin. Yksikkö testiympäristö on havainnollistettu seuraavasti (Kuvio 2). Useimmiten ohjelmistojen ajuri ei ole mitään muuta kuin "pääohjelma", joka vastaanottaa testitapausten tiedot ja välittää tällaisia tietoja eteenpäin seuraavalle testattavalle elementille ja lopuksi tulostaa olennaiset tulokset. Muhvilla voi korvata alaisia moduuleja jotka kutsuneet testattavaksi. Muhvit tai "vale-alaohjelma" käyttävät alisteinen moduulien rajapintaa, voivat tehdä vähäisiä tietojen käsittelyjä, tarjoa sisäisen tietojen tarkastus ja lopuksi palaa valvonta moduulille joka niitä on kutsunut.



Kuvio 2. Yksikkötestien ympäristö (Pressman 2005, 397)

Ajurit ja muhvit tarkoittavat lisä kustannuksen menoja. Eli molemmat ovat ohjelmistoja, jotka on kirjoitettava (muodollinen suunnittelu ei ole yleisesti käytössä), mutta ne eivät toimita lopullisen ohjelmiston mukaan. Jos ajurit ja muhvit pitävät yksinkertaisina, todellisia yleiskustannuksia ovat suhteellisesti alhaisia. Valitettavasti monneissa osissa ei ole mahdollista testata riittävällä tavalla vain yksikertaisena ajurina tai muhvina. Tällöin täydellinen yksikkötestaus voidaan lykätä integraatio testien vaiheen saakka joissa ajurit tai muhvit ovat myös tarpeellisia.
Kun moduulissa käsitellään vain yhtä funktiota, testitapausten määrä voidaan vähentää ja virheitä voidaan helpommin ennustaa ja paljasta. (Pressman 2005, 394 – 397.)

2.1.4 Integraatio testaus

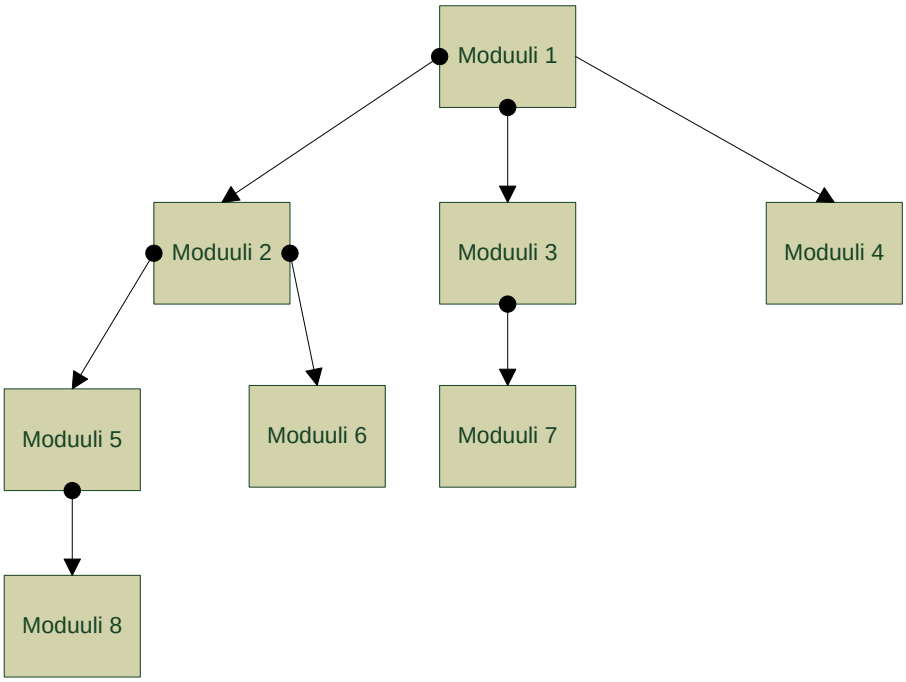
Aloittelija ohjelmistojen maailmassa voi kysyä näennäisesti laillinen kysymys: kun kaikki moduulit ovat testattu yksikössä ja jos ne kaikki toimivat erikseen oikein, miksi tulee epäilys että ne eivät toimi panemassa niitä yhteen? Ongelmana on tietenkin rajapinnoissa. Tietoja saattaa hävitä yli rajapintoja, yhdessä moduulissa voi olla tahaton virhe, joka sisältää haitallinen vaikutuksen toiseen moduuliin tai funktioon, alifunktioon, jolloin ne pistetään yhteen, ei saada tuloksi toivottuja tehtäviä. Integrointitestaus on systemaattinen menetelmä jonka tarkoitus on rakentaa ohjelmistoarkkitehtuurin. Ja samalla kokeiden avulla paljastaa virheitä, jotka liittyvät rajapintoihin. Tavoitteena on ottaa testattava komponentit ja rakentaa ohjelman rakenne, joka vastaa määritelty tavoitteen. (Pressman 2005, 397.)

Ohjelmistokehityksessä useimmiten toteutetaan ohjelmistoa differentiaalitonta (*nonincremental)* integraatiolla eli rakentaa ohjelma niin sanottu "big bang" -lähestymistavalla. Kaikki osat on yhdistetty etukäteen. Koko ohjelma on testattu kokonaisuudessaan. Ja tuloksi yleensä saadaan sotku joka johtaa uuteen virheitten joukkoon havaittavaksi. Korjaus on vaikeaa, koska eristäminen aiheuttaa mutkistaa laajan koon ohjelman takia. Kun nämä virheet korjataan, uusia esiintyvät ja prosessi tuntuu näennäisesti loputonta. (Pressman 2005, 397.)

Differentiaalinen *(Incremental)* yhdentymisen on vastakohta ”big bang:in” lähestymistavan. Tässä tapauksessa ohjelma on rakennettu ja testattu pienneissä moduuleissa joten virheet on helpompi eristää ja korjata. Kun rajapinnat ovat todennäköisesti täysin testattu, voidaan siirtyä järjestelmätestin tasolle. (Pressman 2005, 397 – 398.)

Ylhäältä alas integraatiossa (Top-down integration) on differentiaalinen lähestymistapa ohjelmistoarkkitehtuurin rakentamissa. Moduulit ovat integroitavissa (Kuvio 3) siirtämällä alaspäin ohjaus hierarkian läpi, joka alkaa tärkeässä ohjausyksikössä (pääohjelma). Integraatio suoritetaan viisien askelten sarjalta:

- Ohjausyksikkö käytetään testiajurina. Muhvilla voidaan korvata kaikki alaiset moduulit.
- Riippuen integraation lähestymistavasta alisteiset muhvit korvataan yksi kerrallaan todelliset komponentit.
- Testit suoritettiin joka kerta kun jokainen komponentti on uusintegroitu.
- Päätyttyä kunkin testitapauksen muhvi korvataan todellisen komponentin takaisin.
- Regressiotestaus toteutetaan varmistamaan, että uusia virheitä ei ole esiintyy uudelleen.



Kuvio 3. Ylhäältä alas integraatio (Pressman 2005, 398)

Ylhäältä alaspäin yhdentymisstrategia tarkistaa merkittävät ohjauskohdat tai päätöskohteet varhain testausprosessin vaiheessa. Hyvin suunniteltu ohjelmassa, päätöksenteko tapahtuu ylemmillä tasoilla ja siksi on kohdannut ensimmäisenä. Jos ei ole suurta ohjaus ongelmia, varhainen tunnustaminen on tärkeää. Ylhäältä alaspäin strategia kuulostaa suhteellisesti mutkatonta, mutta käytännössä loogisia ongelmia voisin esiintyä. Yleisin näistä ongelmista syntyy, kun saaminen oikea tulosta testaamisessa alimmaistasoilla vaatii asianmukaista testausta ylemmillä tasoilla. Muhvit korvaavat matalan tason moduulit testauksen alussa, se tekee mahdotonta merkittävää tiedonsiirto ylöspäin ohjelman rakenteen kautta. Testaajalla on kolme vaihtoehtoa:

- Lykätä monia testejä kunnes muhvit korvataan todellisilla moduuleilla.
- Kehittää muhvi, joka suorittaa ainoastaan toimintoja, jotka simuloivat todellisia moduuleja.
- Integroidaan ohjelmisto hierarkia pohjasta ylöspäin (*bottom-up integration*).

(Pressman 2005, 398 – 399.)

Alhaalta ylöspäin integraatio (Bottom-up integration) kuten sen nimenkin kertoi, alkaa rakentamisen ja testaamisen ohjelman atomin tasolla, alhaisilta moduuleilta ylöspäin. Koska komponentit on integroitu alhaalta ylöspäin, integraatio strategia voidaan kuvailla seuraavasti:

- Matalan tason osat yhdistetään klusteriksi (joskus kutsutaan rakenteet), jotka suorittavat tietyn ohjelmiston osatoiminnot.
- Ajuri (testiohjaus ohjelma) on kirjoitettava, ohjataan testitapausten tulo ja lähtö -jonot.
- Koko klusteri on testattava.
- Ajuri on poistettava ja klustereita integroidaan liikkumaan ylöspäin kohti ohjelman rakenteessa.

Koska integraatio liikkuu ylöspäin, tarpeellisuus erilliseen testi ajurien vähenee. Itse asiassa, jos kahden ylimmän tason ohjelman rakenteen on integroitu ylhäältä alaspäin, ajureiden määrä voisin vähentää huomattavasti. (Pressman 2005, 400 – 401.)

2.1.5 Regressiotestaus

Aina kun uusi moduuli on lisättävä ohjelmaan integraatio testaamisen osana, ohjelman sisältö muuttuu. Uudet tietovirran polut ovat aktivoituneet, uudet I/O voivat esiintyneet, ja se johtaa uudeksi ohjauslogiikaksi. Nämä muutokset saattavat aiheuttaa ongelmien toimintoihin, jotka aiemmin voivat toimia moitteetonta. Regressiotestauksen tarkoitus on tarkista korjatut testitapaukset uudelleen ja varmista että muutokset eivät ole lisännyt tahattomia sivuvaikutuksia. Regressiotestaus voidaan suorittaa manuaalisesti, käyttämällä uudelleen korjattu testitapaukset tai automatisoinnin (kaapata/toisto) työkalujen kautta. Kaapata/toisto (*capture / playback*) työkalut antavat ohjelmistosuunnittelijalle kaapata testitapaukset ja tulokset, toista niitä myöhemmin ja vertailla toisiaan. Regressiontestaus sisältää kolme erilaisia luokkien testitapauksia:

- Testin koekappaleen agentti joka suorittaa kaikki ohjelman toiminnot.
- Lisä testit, jotka keskittyvät ohjelmiston toiminnallisuudessa, ja joihin todennäköisesti vaikuttavat muutokset.
- Testit, jotka keskittyvät ohjelmistonosioissa jotka ovat oletettavasti muuttuneet.

Heti kun integraatio testien vaihe sujuu tarpeeksi pitkiin, regression testien määrä voidaan kasvamassa tarpeeksi suureksi. Siksi regression testaus on suunniteltava koskemaan vain niitä testejä, joissa käsitellään yhtä tai useampia virheiden luokkia kunkin suuren ohjelman toiminnossa. On epäkäytännöllistä ja tehotonta suoritta uudelleen kaikki testit kun, muutos on tapahtunut. (Pressman 2005, 401.)

2.2 Testauksen taktiikka

Kun lähdekoodi on luotu, ohjelmisto on testattava, paljastamaan ja korjaamaan niin paljon virheitä kuin mahdollista ennen toimitusta asiakkaalle. Testaajan tehtävänsä on suunnitella useita testitapauksia, joilla suurin todennäköisyyden löydetään virheitä.

Tavanomaisia sovelluksia, testataan kahdesta eri näkökulmasta: sisäisen ohjelmalogiikkaan käytetään "valkoinen laatikko" testitapausten suunnittelun tekniikoita, ohjelmiston vaatimukset käytetään "musta laatikko" testitapausten suunnittelun tekniikoita. Toisen sanoen summataan seuraavasti:

- valkoisen laatikon-testaus on rakenteellinen, sisäinen testaus
- musta laatikon-testaus on toiminnallinen, ulkoinen testaus

Hyvä tapa toteuttaa testausmenetelmiä on vaihda näkökulmaa. Täyty yritä kovasti "murtautua" ohjelmaa keksittämällä testitapaukset kurinalaisesti ja tarkistaa testitapaukset että ne luotu riittävän perusteellisesti. Lisäksi voidaan arvioida testauksen kattavuus ja seurata virheiden havaitsemiseen toiminnot. (Pressman 2005, 420-421.)

2.2.1 Mustalaatikko menetelmä

Musta-laatikko testaus yleisesti edellyttää että testit tehdään ohjelmiston käyttöliittymässä. Se tarkastellaan joitakin olennainen järjestelmän osa, ja se ei ottaa huomioon tai ottaa vain vähän ohjelmansisäisen loogisen rakenne. Mustalaatikon eli blackbox -testaus perustuu testattavan ohjelman tai sen komponentin esim. joku toiminnon siirrännän käyttäytymiseen. Testattavan kohteen oikeellisuutta tarkistetaan vertaamalla saatuja tulosteita haluttuihin tai odotettuihin tulosteisiin. Mustalaatikon menetelmässä ei käytetä testattavan kohteen rakenteesta tai sisällöstä, vaan tutkittaan kohteet jossa tulosteet eroavat syötearvoilta. Seuraavassa on lueteltu muutamia mustalaatikon vaihtoehtoja:

Raja-arvon analyysi (Boundary value analysis) perustuu siihen että testitapausten lukumäärä saataisiin mahdollisimman pieneksi, mutta silti mahdollisimman kattaviksi, on yksi mahdollisuus jakaa syötejoukko ns. reunan arvoanalyysiin. Reunan arvoanalyysissä testitapauksiksi valitaan reunoilla olevia arvoja. Raja-arvoanalyysi perustuu olettamukseen, että jos testattava kohde epäonnistuu joillakin rajojen arvoilla, se yleensä epäonnistuu myös arvoilla, jotka kuuluvat rajo-

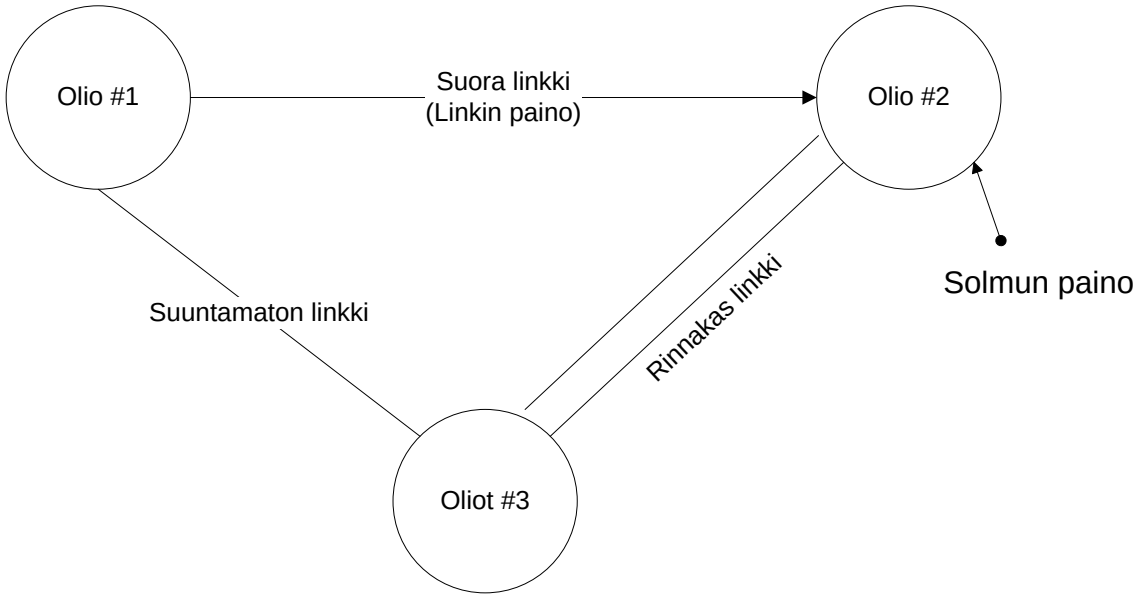
jen sisään. Nämä arvot voivat olla esimerkiksi maksimi tai minimi, lyhyitä tai pitkiä ja niin edelleen. Tällä tavalla voidaan saada testitapausten lukumäärää pienennettyä. (Kautto 1996.)

Ekvivalenssiositus (Equivalence partitioning) on musta laatikon testausmenetelmä joka luokitella ohjelman syöte-jonot luokkiin, joista voidaan johtaa testitapaukset. Ekvivalenssiositus voidaan kuvailla seuraavasti:

- Sekä syötteet että tulosteet jaetaan ekvivalenssiluokkiin toisin sanoen mikä tahansa arvo luokassa edustaa koko luokkaa. Valitaan jokaisesta luokasta yksi arvo testitapaukseen.
- Ekvivalenssiluokan rajat valitaan myös testitapauksiksi.
- Jokainen ekvivalenssiluokan arvo antaa saman testituloksen (toimii oikein tai ei) ja paljastaa virheen samalla todennäköisyydellä.

Ideallinen testitapaus on se joka paljastaa kaikki kuluvien luokkien virheet (esim. kaikkien virheellinen merkkijonojen käsittely), joka muutoin edellyttää monta testitapausta suoritettuna ennen kun yleinen virhe on havaittu. Ekvivalenssiositus pyrkii määrittelemään keino, joka paljastaa uusia virheiden luokkia, ja sen avulla vähentää testitapauksisen kokonaismäärä. Ekvivalenssiosituksen luokka sisältää oikea- tai virheellisen- joukon tilaa kunkin syötejonolle. Yleensä syötejono on joko erityinen numeerinen arvo, arvojen joukko, tai Boolen ehto (esim. K tai E). (Pressman 2005, 437.)

Kuvio-perustuva testauksessa (Graph-Based Testing Method) ohjelmisto testaus alkaa luomalla tärkeintä olioiden kaavioita ja niiden suhteita ja sitten suunnitellaan testien sarjoja, jotka kattavat kuvioita niin, että jokainen kohde ja suhde käydään läpi vähintään kerran ja se voi auttaa virheiden paljastumiseen. Suorittamassa nämä vaiheet, ohjelmoija ensiksi piirtää solmuja jotka edustavat olioita, linkkejä jotka edustavat olioiden suhteita, solmien painot jotka kuvaavat solmien ominaisuutta (esim. erityinen tiedon arvo tai voimien käyttäytyminen) ja linkkien painot jotka kuvaavat linkkien ominaisuuksia. (Kuvio 4.)



Kuvio 4. Kuvio perustuva testaus (Pressman 2005, 436)

Ohjelmistosuunnittelija sitten käy testitapaukset läpi kattamalla kaikki suhteet. Nämä testitapaukset ovat suunniteltu yrittää löytyä virheitä jotka voivat esiintyä jonkin suhteissa. (Pressman 2005, 435.)

2.2.2 Valkoisenlaatikko menetelmä

Valkoisenlaatikko testaus perustu ohjelmistojen nojautumalla sisäisiä yksityiskohtia. Käymällä ohjelman loogisen polkujen läpi ja tutkitaan niiden yhteistyötä, antamalla testitapaukset jotka käyttävät tiettyjä ehdot ja/tai silmukat. Ensi silmäykseltä näyttäisi siltä, että hyvin perusteellinen valkoisenlaatikon testaus johtaisi jopa 100 prosentin oikeaan ohjelmaan. Kaikki mitä tarvittaisi tehdä, on tunnistaa kaikki loogiset polut, toteuttaa testitapaukset jotka käyttävät niitä ja arvioida saatu tulokset, joten luomalla kaikki vaadittava testitapauksia kaatamaan niitä kokonaan. Valitettavasti täydellinen testaus aiheuttaa tiettyjä loogisia ongelmia. Valkoisenlaatikko testaus ei kuitenkaan hylättävä epäkäytännöllinen. Rajoitettu määrä tärkeitä loogisia polkuja voidaan valita ja toteuttaa. Esimerkiksi tärkeää tiedon rakenteita voidaan tarkista oikeellisuuteen. Käyttämällä valkoisenlaatikko testausmenetelmiä, ohjelmistosuunnittelija voi johtaa testitapauksia jotka takaavat seuraavat asiat:

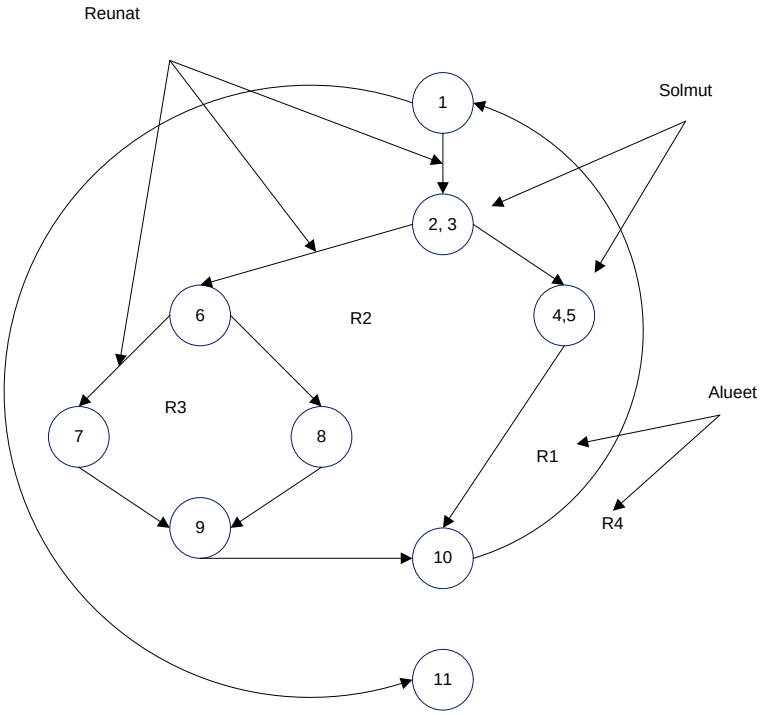
- kaikki riippumattomat polut sisältävä moduulissa käytettävä ainakin kerran
- käyttää kaikkia loogisia päätöksiä niiden oikean ja väärin puolella
- suorittaa kaikki silmukan rajat ja niiden sisäisen toiminnan rajat

– käydään läpi sisäiset tietorakenteet jotta tarkista niiden virheettömyyden.

(Haikala & Märijärvi 2004, 294.)

Yleinen tapa toteutta valkoisen laatikon testausta on polkutestaus.

Polkutestaus (Path testing) on testaustekniikka jonka ensimmäisenä ehdotti Tom McCabe vuonna 1976 (McCabe 1976). Polkutestaus menetelmä antaa suunnittelijalle ansaita loogisen monimutkaisuuden mitat ja käyttää niitä suunniteltu testitapauksessa määrittelemään polun toteuttamista. Ansaittu testitapaukset on vakuutettava että jokainen ohjelman polku suoritettava ainakin kerran testauksen aikana. Tämä ominaisuus voidaan saada eri tekniikalla kuten esimerkiksi *vuokaavion merkintätapa* (Flow path notation*)*. (Haikala & Märijärvi 2004, 295.)



Kuvio 5. Vuokaavion merkintätapa (Pressman 2005, 426)

Vuokaavion merkintätapa käytetään kuvaamaan ohjelman valvonnan rakenne (Kuvio 5). Jokainen ympyrä kutsutaan vuokaavion solmuksi ja edustaa yhtä tai useampaa lausuntoja tai metodeja. Nuolet kutsutaan reunaksi tai linkiksi ja ovat edustavat ohjelman valvonta. Reuna on aina päätyttävä loppunsa solmulla, vaikka solmu ei edustaa mitään menettelyjä tai lausuntoja. Alueet jotka sijaitsevat linkkien välillä kutsutaan alueeksi. Polku täytyy liikkua ainakin yksi reunan pitkin joka ei ole vielä käytettävissä entisen määritetyn polussa. Esimerkin polkujen joukko (Kuvio 5) on:

- polku 1: 1-11
- polku 2: 1-2-3-4-5-10-1-11
- polku 3: 1 -2-3-6-8-9-10-1-11
- polku 4: 1-2-3-6-7-9-10-1-11

Jokainen uusi reitti tuo uuden reunaa. Reitit 1, 2, 3, ja 4 muodostavat vuokaavion perus joukko. Eli jos testit voidaan suunnitella toteuttamalla näitä polkuja pitkin taatuttaan että jokainen kaavion lause suoritetaan ainakin kerran, ja jokainen ehtolause suoritetaan omalla oikealla ja virheellisellä puolella. (Pressman 2005, 425 – 429.)

Mistä tiedetään, kuinka monta polkuja etsimissä? Vastaus voi antaa syklomaattinen arvo (Katso tarkemmin aihe "testauksen mittarit"). Syklomaattinen mutkaisuus, V (G) määritellään kuten:

V (G) = E - N + 2

Missä E on vuokaavion reunojen määrä ja N on solmujen määrä. Eli meidän tapauksessa on 11 reunoja ja 9 solmuja:

V(G) = 11 – 9 + 2 = 4

	1	2	3	4	5	6	7	8	9	10	11
1		a	a								k
2				b	b	c					
3				b	b	c					
4										i	
5										i	
6							e	d			
7									f		
8									g		
9										h	
10	j										
11											

Kuvio 6. Kaavion matriisi (Pressman 2005, 431)

Kattamaan kaikki lauseita ja ehtoja vähintään kerran tarvitse 4 polkuja tai 4 testitapausta.

Menettelylle joka johdetaan kaavio ja jopa määritetään polkujen perusjoukko voi tarvita koneistamista.

Kaavion matriisi (Graph matrices), voi olla varsin hyödyllinen. Kaavio matriisi on neliömatriisi, jonka koko (eli rivien ja sarakkeiden määrä) on yhtä sama kuin solmujen määrä. Jokainen rivi tai sarake vastaa tunnistettu solmuun, ja matriisin arvot vastaavat reunoihin solmujen välillä. Yksinkertainen esimerkki kaavion matriisin (Kuvio 6) viittaa matriisiin, jossa jokainen solmu on yksilöity numeroilla, vaikka jokainen reuna on tunnistettu kirjaimilla. Lisäämällä linkin paino kunkin matriisin soluun, kaavion matriisi voi tulla tehoksi keinoksi arvioittamaan ohjelman valvonnan rakenne (control structure) testauksen aikana. Linkkien paino antavat lisätieto valvonnan rakenteista. Yksinkertaisimmillaan, linkin paino 1 tarkoittaa yhteys on olemassa tai 0 jos yhteyttä ei ole.

Mutta yhteyden painolle voidaan aseta muita tehokkaita ja hyödyllisiä ominaisuuksia:

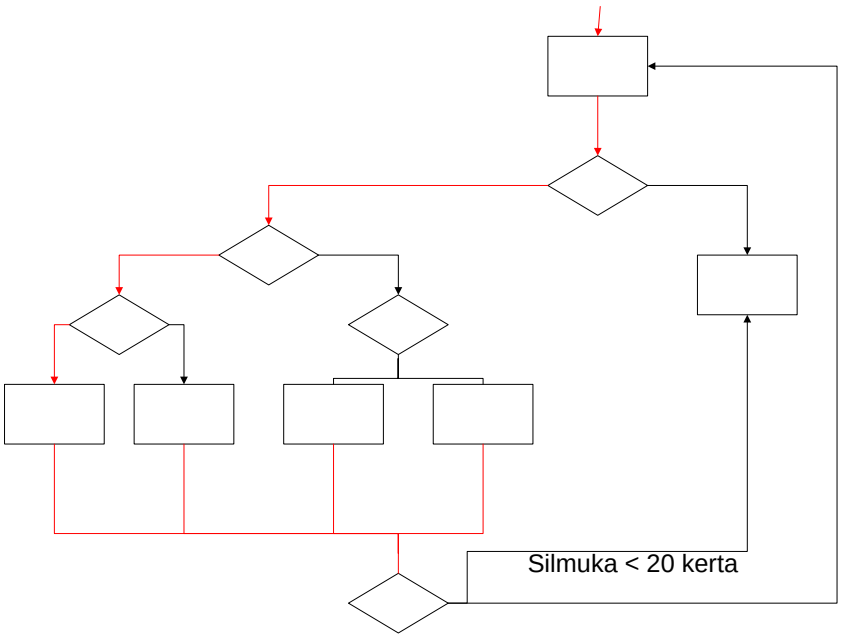
- todennäköisyys, että yhteys (reuna) suoritetaan
- käsittelyaika
- tarvittava muistin määrä
- muita tarvittava resurssia linkkien kulkujen yhteydessä.

(Pressman 2005, 429 – 434.)

Valkoisen laatikon testauksessa käytetään kattavuus-termi. Tämän käsitteen alle tarkoitetaan, kuinka monta vaaditaan testitapauksia kattamaan sovelluksen lähdekoodia valittu kriteerijana 100 %:ksi.

Polkukattavuudessa testitapaukset valitaan siten, että mahdolliset suorituspolut ohjelmakoodissa läpikäytettään kattavasti ja tietty kriteerit täytettään seuraavasti:

- testauspolitiikka määrää kattavuuden
- arvioinnissa käytetään kaikista löytyneiden virheistä osuutta



Kuvio 7. Polkutestauksen kattavuus (UTU 2007)

Esimerkissä (Kuvio 7) on esitetty yhden moduulin kaaviollinen esimerkki. Tällä moduulilla on 10^4 erilaista suorituspolkua! Tämä tarkoitta ohjelmalle tarvitse 10^4 testitapausta kattamaan niitä kaikki. Todella paljon, jos ottaa huomioon että moduuli on joku osa ja useimmiten todella pieni ohjelman kokonaisuudesta. Tämä tuntuu melkein mahdotonta. Kuvassa yksi polku joka on valittu testattavaksi väritetty punaiseksi. Käytetään kontrolloitua polkutestausta, pyritään kattavuuteen muilla määritelmillä. Nykyään on keksitty muutama menetelmiä auttamaan kaatamaan suoritus-polkujen tehokkaasti.

*Lausekattavuus (*statement coverage*)*: jokainen lause kuten ((a>b) && (c==3)) on suoritettava vähintään kerran. Se saa jommankumman arvon, tosi tai epätosi. Tässä menetelmässä on muutama heikkoja puolia:

- ei ota huomioon haarautumisia kontrollirakenteita, joten paljon jää suorittamatta
- ei ota huomioon moduulin sisäisen tilan vaikutusta lauseen suoritukseen

Päätöskattavuus (haarakattavuus, decision coverage): jokainen ehtolauseke kuten ((a>b) && (c==3)) saa vähintään kerran molemmat arvonsa (tosi/epätosi) eli ehdollisen rakenteen molemmat haarat suoritetaan. Yleisesti on vaadittava minimikattavuuteen. Heikkona puolena voidaan olettaa ehtojen kaikkien osia saavat kombinaatioita jotka eivät testaa.

Ehtokattavuus (condition coverage): jokainen ehtolausekkeen osa kuten ((a>b) && (c==3))) saa vähintään kerran molemmin arvonsa (tosi/epätosi). Se ei implikoi välttämättä päätöskattavuutta. Mutta yhdistelmällä sekä päätös- että ehtokattavuus korjataan lausekattavuuden heikkoja puolia. (Haikala & Märijärvi 2004, 294–295.)

Muunnettu ehtokattavuus: kompromissi, joka vaatii vähemmän testitapauksia kuin tavallisessa ehtokattavuudessa. Laaja käytetään ilmailutekniikan ohjelmistossa. Vaaditaan ilmailu standardin RTCA/DO-178B (FAA 2003). Pyritään osoittamaan, että jokainen looginen operandi voi itsenäisesti vaikuttaa päätöksen tulokseen. Jokainen operandi muuttuu arvojen tosi ja epätosi välillä, säilyttäen muut operandit joidenkin kiinteillä arvoilla. Odotettu tulos pitäisi vaihdella kun operandit ovat muuttuneet. Tämä saattaa edellyttää kiinteiden arvojen hakemiselta, jotta näin tapahtuu. (Ross 1998, 104 – 106.)

Mitä on mitata kattavuuden avulla? Tarkoituksena on löytää puutteita testiaineistossa, ei niinkään todistaa testauksen perusteellisuutta. Prosenttilukujen tulkinnassa kannattaa olla varovaisena, sillä suuri kattavuus ei välttämättä tarkoita hyvää testausta ja mitat eivät välttämättä ole keskenään vertailukelpoisia. 100 % kattavuus käytännössä saaminen usein mahdotonta. Yleensä tavoitteena on esimerkiksi 85 % kattavuus (lause- tai päätöskattavuus tai sekä että). Ja vielä täytyy muistaa että kattavuus ei ota huomioon olion tilaa. (UTU 2007.)

2.3 Testauksen mittarit

Riittävä testauksen määrää on vaikea arvioida. Etenkin järjestelmätestauksessa testausta voidaan jatkaa ”kunnes aika ja rahat loppuvat". Testauksen lopettamiselle tulisi aina asettaa hyväksymiskriteerit, jotka määritellään testaussuunnitelmassa. Järjestelmätestauksessa kriteeri voi liittyä esimerkiksi kumulatiiviseen löydettyjen virheiden määrään; kun virhekäyrä tasaantuu, testaus voidaan lopettaa. Varsinkin moduulitestauksessa tarvittavan testauksen määrää voidaan koettaa arvioida ns. mutkikkuusmitoilla ja testauksen riittävyyttä puolestaan ns. kattavuusmitoilla ja virheitä kylvämällä. Vaikeutena on se, että yleensä projektilla on tässä vaiheessa kiinteät resurssit ja aikataulukin on lyöty lukkoon ja kerrottu asiakkaalle. Saattaa olla vaikeaa arvioida projektin kestoa, mikäli testauksen lopetuskriteerinä pidetään vain virhekäyrän tasaantumista, koska tasaantumiseen tarvittava työmäärä ei ole tiedossa etukäteen. (Lehtimäki 2006, 171–174.)

Mutkikkuusmitan avulla voidaan yrittää paikantaa ohjelmistosta paljon testausta vaativat moduulit. Testaus mittarit voidaan jakaa kahteen pääryhmään: mittarit, jotka yrittävät ennustaa

todennäköinen testien määrä tarvittava eri testaustasolla, ja mittarit, jotka keskittyvät testin kattavuuteen jokin tietyllä ohjelman osalla tai moduulilla. Tunnetuimmat mutkikkuusmitat ovat tekijöidensä mukaan nimetyt Halsteadin mitta ja McCaben syklomaattinen numero.

Halsteadin mitta määräytyy seuraavasti. Lasketaan ohjelmassa olevien operaattoreiden ja operandien yhteinen lukumäärä N (operaattoreita ovat varatut sanat ja +, -, jne.). Tämän jälkeen lasketaan ohjelmassa käytettyjen eri operaattoreiden ja operandien yhteinen lukumäärä n. Halsteadin mitta ohjelmalle on tällöin Nlog2n. (Haikala & Märijärvi 2004, 293 – 294.)

McCaben syklomaattinen numero lasketaan yleensä erikseen ohjelman jokaiselle funktiolle. Sen arvo saadaan lisäämällä ykkösen numero funktioon kontrolliverkon haarautumiskohtien lukumäärään. Saatu luku kuvaa funktion kontrolliverkon monimutkaisuutta. Testauksen kannalta tarkastelemalla sen arvoa voi pitää minimimäärä testitapauksia, joilla ko. funktio on testattavissa. (Pressman 2005, 491.)

Syklomaattinen numero on keskeinen lähtökohde komponenttitason polku-testauksessa. Lisäksi syklomaattinen numero voi tulla perusteena joilla päätetään moduulien täydellisen testauksen mahdollisuutta. Moduulit korkean syklomaattinen numeron mukaan ovat todennäköisemmin virheellisemmin kuin moduulit, joiden syklomaattinen numero on pienempi. Tästä syystä testaaja täytyisi uhrata enemmän paljastamaan virheitä ennen kuin moduuli on jo integroitu järjestelmään. Yleisimmin käytetään olevan McCaben syklomaattinen numero. Halsteadin mitta on lähinnä historiallista merkitystä. (Hutcheson 2003, 273.)

Mutta on olemassa myös muita mittoja jotka kertovat lisätieto testauksen tilasta:

Yhteenkuuluvuuden puute (Lack of cohesion in methods, LCOM). Mitä suurempi LCOM on, sitä useampi metodien tilaa on testattava varmistamiseksi, että menetelmät eivät tuota sivuvaikutuksia.

Julkisen pääsy tietojäsenille (Public access to data members ,PAD). Tämä vertailuluku ilmoittaa luokkien tai menetelmien numero, jotka voivat käyttää toisen luokan attribuutit. Korkeat arvot PAD johtaa mahdollisten haittavaikutukseen luokkien joukossa. Testien tarkoitus on varmista että nämä haittavaikutukset ovat paljastuneet ajoissa.

Juurin oleva luokkien määrä (Number of root classes, NOR). Tämä vertailuluku laskee ilmentymisen luokkien lukumäärä. Testitapauksessa kunkin juurin olevalle luokille ja vastaavalle metodille on kehitettävä. NOR numeron kasvaessa, testauksen hyödyllisyys on myös kasvaa.

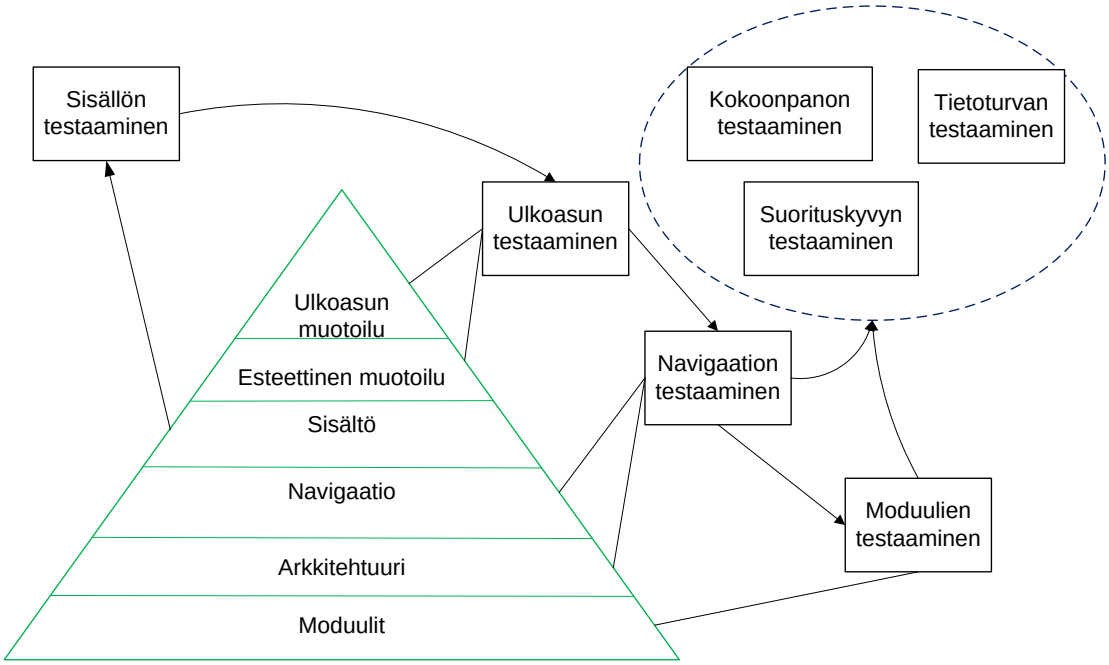
Loogisen veräjien määrä (Fan-in FIN). Fan-in vertailuluvun olio-perinnön hierarkiassa osoittaa minimin periytymisen. Jos FIN > 1 se tarkoittaa, että luokka perii sen attribuutteja ja metodeja useammasta kuin yhdestä juuri luokasta. Eli tämän ominaisuuden olisi vältä aina kun mahdollista.

Lasten lukumäärä (Number of children, NOC) ja perinnön puun syvyys (Depth of the inheritance tree DIT). Jos tämä arvo enemmän kuin 0, juuriluokkien metodit on testattava uudelleen kunkin alaluokalle.

Koodirivien lukumäärä (Lines of Code, LOC) Ajettava koodirivien lukumäärä (ilman kommentteja). Tämä on yksikertainen mittari joka antaa kuva ohjelman koosta. (Pressman 2005, 491 – 492.)

3 Empiirinen osa

Yksikkötestit ovat web-sovellusten testausprosessin avainosa. Vaikka testit yleisesti suoritetaan hyvin tunnettu tietomaailmassa V-mallin mukaan, web-pohjaisissa sovelluksissa on myös oma testauksen erityisyys (Kuvio 8).



Kuvio 8. Web-sovellusten testausprosessi (Pressman 2005, 599)

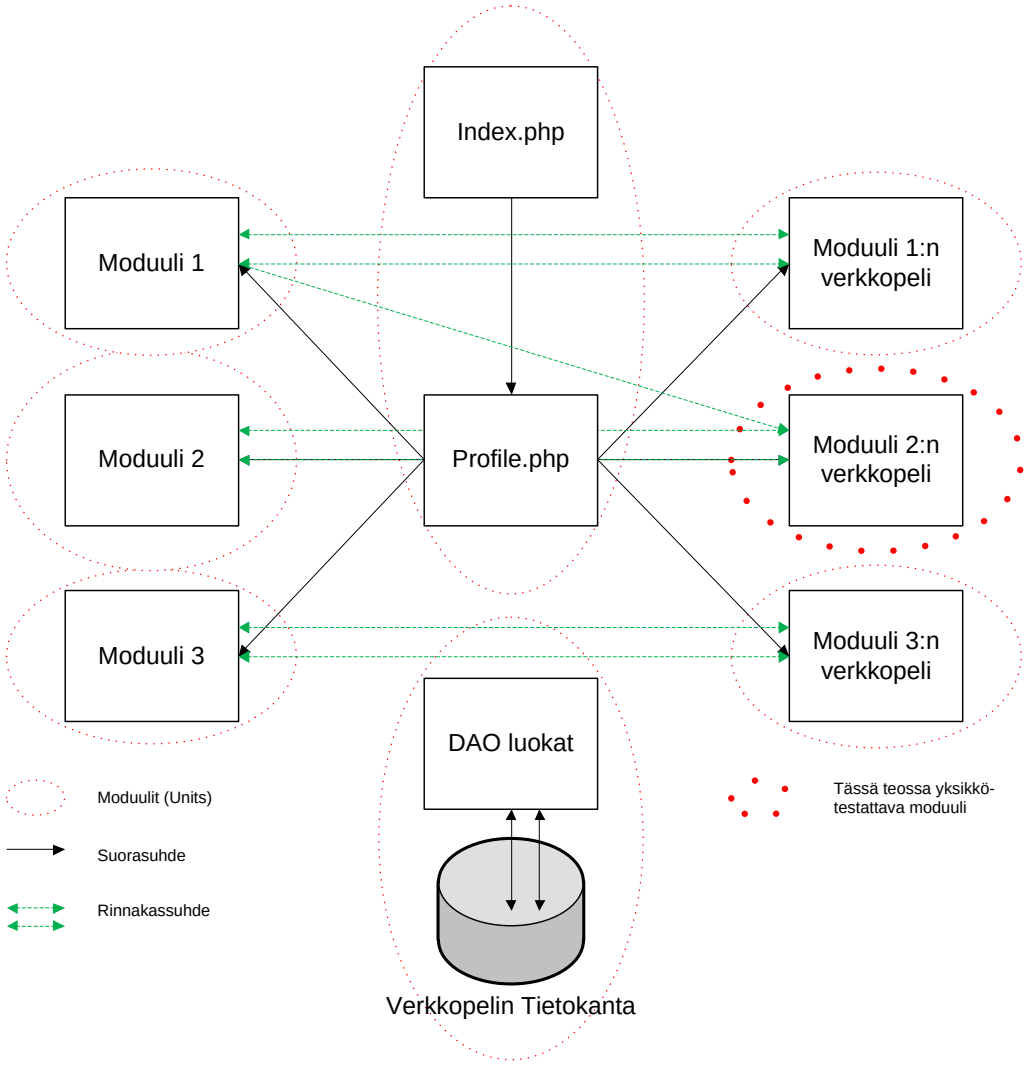
Web-pohjaisen sovelluksessa yksikkötestaus koostu seuraavista osioista: sisällön testaaminen, ulkoasun testaaminen, arkkitehtuurin, navigaation ja moduulien testaaminen. Testaus aloitetaan *sisällön* ja *käyttöliittymän* testaamista (ulkoasun muotoilu, esteellinen muotoilu). Tässä vaiheessa tarkistetaan sisällön kirjoitusvirheitä, ulkoasu ja toiminnot joihin avulla tapahtuu vuorovaikutus loppukäyttäjän kanssa. Sitten jälkeen siirretään *arkkitehtuurin* ja *navigaation* testaamiseen. Tarkistetaan mm. linkkien tai valikkojen toimivuus. Oliko hyvä sovelluksen arkkitehtuuri on valittu (lineaarisen, verkon tai yksinkertaisen hierarkkisen yms.). Ja lopussa siirretään *moduulin* testauksen vaiheeseen. Kun yksi moduuli on testattu, sitä integroidaan muihin moduulien kanssa ja testataan myös *kokoonpano*, *tietoturva* ja *suorituskyky*.

Moduulitestauksessa tarkistetaan web-sovelluksen lähdekoodi ja sen toiminnallisuuden. Jokainen sovelluksen osa on yksikkötestaava. Web-sovelluksessa on kaksi vaihtoehtoa määritellä loogisia yksikköjä (komponentteja):

– Staattisen HTML pohjatussa sovelluksessa yksiköksi voi määritellä web-sivu. Jokainen web-sivu eristää linkkien navigoinnin sisällön, ja elementtien käsiteltyn (lomakkeet, skriptat, pienoissovellukset).
– Dynaamisen HTML pohjatussa sovelluksessa yksiköksi voi määritellä toimiva osa, joka on käännetty suoraan loppukäyttäjälle tai infrastruktuurin komponentille, jonka ansiosta Web-sovellus suorittaa kaikkia ominaisuuksiaan. Useimmiten testit suunnitellaan käyttämällä musta-laatikon menetelmää. Kuitenkin, jos sovelluksen lähdekoodi on monimutkaista, sitten lisäksi käytetään valkoisen-laatikon testejä.

(Pressman 2005, 598–601.)

3.1 Kohteen esittely



Kuvio 9. Verkkopeli-projektin osat

Nykyinen verkkopeli-sovelluksen versio koostuu kolmesta moduulista, joihin kaikkiin sisältyy kolme verkkopeliosioita (Kuvio 9). Kaikki osat ja profiili toimivat MySQL tietokannan kanssa. Verkkopeli perustuu PHPn kielellä, joka mahdollistaa tuotteen kehitystä ja helppoa käyttöönottoa myös kehitysvaiheen jälkeen. Tässä raportissa keskitytään verkkopelin kakkosen moduuliin ja sen suunnittelemiseen.

Tämä moduuli jäljittelee tosielämän hotelli ja ravintola liike-elämän toimintaa, kuten esimerkiksi rekrytointi, investointi ja budjetointi. Pelaamalla peliä ja analysoimalla sen tulosraportteja, opiskelija pystyy ymmärtämään yleisiä liike-elämän toimintoja, jotka tapahtuvat hotelli ja ravintola-alan elinkaaressa ja tulevaisuudessa ja lisäksi miten nämä toimet vaikuttaisivat liiketoiminnan tuloksiin. Tämän pelin tarkoitus on pelata ryhmissä, mutta on mahdollista myös pelata sitä yhdenhengen pelaajana tekoälyisien pelaajien kanssa. Pelaaja voi valita joko hotelli- tai ravintolapelin tai pelata yhdistettyä hotelli- ja ravintolapeliä. Suurin etu verkkopelin osan 2 on monen käyttäjien kilpailutila, koska toimii verkon yli ja antaa mahdollisuuden verkkopelaajille osallistua peliin. Monen käyttäjien kilpailutila tarkoittaa, että jokaisen peliryhmän sisällä samaan aikaan osallistuu tietty pelaajien määrää ja jokaisen käyttäjän toiminta voi vaikuttaa muiden pelaajien yritysten tuloksiin.

3.1.1 Verkkopelin testauksen nykytila

Verkkopelin testauksen vaihe on alkanut huhtikuusta 2010. Työt alkoivat suunniteltaessa integraatiotason testitapauksia mustan laatikon menetelmällä, joista on puhuttu viitekehyksen luvuissa 2.1.4 ja 2.2.1. On toteutettu muutama testauksen iteraatio kohdeyrityksessä ja erässä ammattikorkeakoulun tiloissa. Niihin osallistui yhteensä noin 100 opiskelijaa. Verkkosovelluksen osa 2 tästä ajasta on testattu 3 kertaa. Jokaisessa testauksen iteraatiossa pyrittiin käyttämään maksimi määrä sovelluksen toimintoa ja ominaisuutta, erityisesti huomio kohdistettiin niihin, jotka käyttävät usein verkkoa. Verkkopelin osan 2:ta pelattiin omissa ryhmissä verkon yli. Testauksesta havaittiin noin 50 virheettä ja parannusehdotusta. Jokaisen iteraation jälkeen kaikki löydetyt virheet analysoitiin projektin kehitystiimin sisällä ja määriteltiin määräajat, milloin ko. virheitä on eliminoitava. 2 kertaa verkkosovelluksen kakkosen osalle toteutettiin regressiotestaukset (Luku 2.1.5) joissa korjatut virheet tarkistettiin uudelleen niiden oikeellisuuden sekä sivuvaikutusten kannalta. Kaikki ko. testaukset koskivat sovelluksen integraatiotasoa.

Sovellukselle ei ole laadittu vielä testaussuunnitelmaa eikä yksikkötason testejä. Yksi tämän työn tehtävistä on laatia testaussuunnitelma, joka pohjautuu tässä raportissa tuotuun materiaaliin (Liite 7). Testaussuunnitelmassa määritellään testausympäristö, testauksen dokumentaatio,

ohjelmiston testattavat osat ja ominaisuudet, testien hyväksymisen kriteerit, testien lopettamiskriteerit, testaamiseen liittyvät roolit ja vastuuhenkilöt, aikataulu, testausprosessin ja testaukseen liittyvät riskit sekä riskienhallinta. Testaussuunnitelma laaditaan yleensä ohjelmistoprosessin suunnitteluvaiheessa, ja siihen kuuluu myös testausprosessin kuvaus. Prosessin kuvauksessa määritellään suoritettavat vaiheet ja tehtävät. Prosessin kuvauksessa voidaan kuvata myös prosessin vaiheen tuloksia. Mutta verkkopelin sovellukselle ei ole ennen tehty testaussuunnitelmaa, siksi toteutetaan se jälkikäteen.

Lisäksi tässä opinnäytetyössä seuraavissa luvuissa pyritään suunnitella ja toteuttaa yksikkötason testit, jotka kohdistavat verkkopelin 2:sen moduuliin. Käydään alusta-loppuun kaikki testausprosessin vaiheilta-vaiheen ja testausten päättyessä pyritään analysoimaan tuloksia ja arvioimaan niitä.

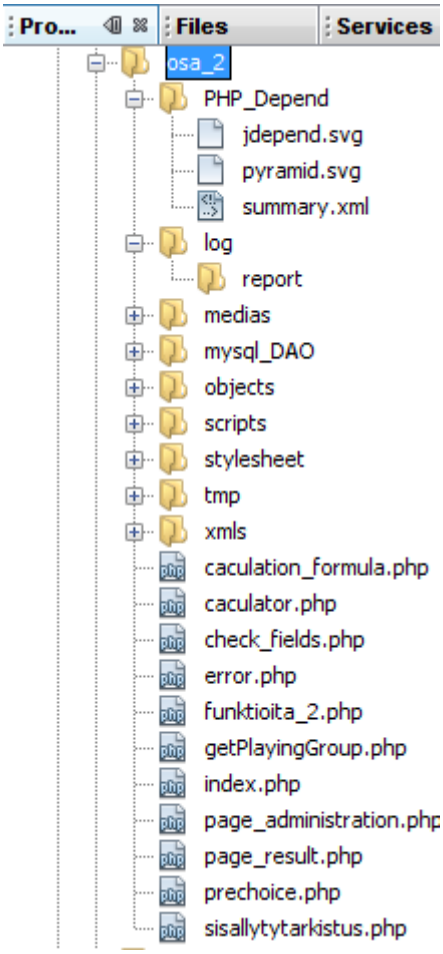
3.2 Aineisto

Aineistoa kerätään useita metodeja käyttämällä. Tietoa hankitaan kirjallisuudesta, seminaareista, internetistä ja kursseilta. Tietoa haetaan myös testausympäristöstä, jossa pyritään toteuttamaan muutama yksikkötason testitapaus.

Tutkimus jakautuu siten, että tutkimuksen tavoitteet on tarkistettava testausympäristön avulla. Ohjelmiston testaamiseen käytetään valkoisen laatikon menetelmää. PHPUnit, Xampp ja NetBeans liittyvät materiaalit löydetään Internetistä. Kaikki testitapaukset on toteutettu työympäristössä sijaitsevalla koneella. Mutta niille ei tarvita erityistä ympäristöä, koska kaikki tässä opinnäytetyössä käytetyt ohjelmistot ovat saatavissa Internetistä ja ladataan vapaaksi. Tutkimusta varten on syvennyttävä testausprosessiin ja tarpeeksi pitkiin menetelmiin niin, että löydetty teoriataustaa voi riittää testitapauksien tekemiseen.

3.3 Menetelmät

Käytettävät tutkimusmenetelmät ovat laadullisia (kvalitatiivisia). Tavoitteena on tyypillisimmin ilmiöiden kuvailu ja toistaminen testausympäristössä. Seuraavana vaiheena tulee analyysi ja arviointi. Analyysin perusteella voidaan joko luoda uutta teoriaa tai parantaa kohdetta. Tutkimusmenetelmän valinnassa täytyy ottaa huomioon tutkimuksen lähtökohdat ja tavoitteet. Tämän opinnäytetyön tutkimuskohteeksi on valittu web-pohjaisten sovellusten yksikkötestaus, joka ajetaan PHPUnitin ajoympäristöön. Tämän aiheen analysointia varten pyritään keräämään aineistoa testausympäristöstä.



Kuvio 10. Verkkopelin osan 2 testauksen ympäristö

Testauskohteena on valittu verkkopelin 2 osa joka on osa verkkopelin projektia (Kuvio 9). Tämä moduuli on valittu ensisijaisesti sen syystä, että PHPUnit testaus vaatii oliopohjaista ohjelmointimenetelmää. Moduulin arkkitehtuuri voidaan kuvailla kuin sekainen. Siellä käytetään sekä oliopohjaisia, että proseduurisia menetellyntapoja. Vuorovaikutus muiden osien kanssa tapahtuu tietokannantasolla ja PHPn SESSION globaali muuttujan ansiosta. Kaikki verkkopeliin 2 osaan liittyvät lähdetiedostot löytyvät kansiosta: osa_2. Se koostuu seuraavista alikansioista (Kuvio 10):

- PHP_Depened: Tämä kansio on tarkoitettu PHP_depenedin tuottamille tiedostoille.
- log: PHPUnitin lokki tiedostojen kansio.
- medias: sisältää kaikkia multimedia-tiedostoja, kuten kuvia, animaatiota yms..
- mysql_DAO: sisältää tietokannassa käytettäviä metodeja ja objekteja
- objects: sisältää kaikki olioluokat
- scripts: sisältää kaikki JavaScript-tiedostot

- stylesheet: sisältää kaikki CSS -tyylisivut.
- xmls: sisältää kaikki pelissä käytetyt XML tiedostot

3.3.1 Testausympäristö

Testausympäristöön voi kuulua kehitettävästä ohjelmistosta riippuen testauspalvelu, testitietokanta sekä muita ohjelmiston testaamisessa tarvittavia ohjelmistoja ja laitteistoja. Testausympäristön olennainen osa on ajoympäristö, joka on testitapaukset suorittava ja testauskerran tulokset raportoiva ohjelmisto. Opinnäytetyössä ajoympäristöksi on valittu NetBeans ja PHPUnit-yhdistelmä. Yhdistelmä tarkoittaa sitä että tavallisen kehitysympäristön (NetBeans IDE 6.x.x) yksikkötestausta varten on lisätty moduuli (PHPUnit). Miksi on valittu juuri tämä kokoonpano? Ensimmäinen syy on se, että opinnäytetekijällä ei ole ollut ennen riittävää kokemusta yksikkötestausten toteuttamisessa, sekä joku erityisistä suositteluista, etenkin valkoisenlaatikon testien tekemisessä. Toinen syy: nämä ohjelmistot ovat täysi vapaita, kuten avoin PHP-ohjelmointikieli. Lisäksi on olemassa eri järjestelmille tarkoitettuja versioita kuten PC, Linux tai Unix. Internetistä voi löytyä runsaasti asiallista dokumentaatiota, käyttöohjeita ja esimerkkejä. NetBeans ja PHPUnit yhdistelmä antaa mahdollisuus aloittelevalle testaajalle nopeasti omaksua yksikkötestien suunnittelu. NetBeans on suhteellisesti uusi ohjelmointiympäristö (IDE), mutta ehti saada hyvän suosion tietomaailmassa sen helppokäytettävyyden ansiosta. NetBeans sopii pieni- tai keski- tasoisille projekteille, joissa käytetään hyväksi ohjelmointiympäristön perusominaisuudet sekä ohjelmisto-joukon yhteistyön työskentely (team), debuggaus ja muut toiminnot.

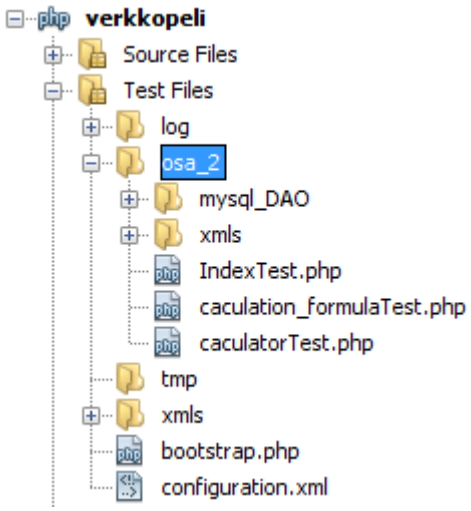
Tällä hetkellä PHPlla on koodattu muutama tarkoitettu web-sovellusten yksikkötestauksien tekemiselle työkalut, ja yksi niistä suosittu on PHPUnit. PHPUnit on osa xUnitin joukosta, jossa toteutuu yksikkötestien tuki melkein kaikille ohjelmointikielille kuten nimittäin Java (JUnite), C#(NUnit) ja muut. Se tukee testauspetiä (Mock-objects), tietokannan testausta, koodin ja testausten mittareita, epätäydellisiä testejä, koodin rungon luomista, automaattista testausta (Selenium RC), koodin kattavuuden-analysointia (tämä PHPUnitin ominaisuus käytetään hyväksi mm. NetBeanissa kun lasketaan testien kattavuus), ja paljon muuta. Tärkeimmät ohjeet löytyy Internetistä (Bergmann 2010). PHPUnit on itsenäinen ja täydellinen työkalu, joka voidaan käyttää itsenäisesti (ilman NEtBeans), käyttämällä testien sarjoja ja konsolia. Se perustuu PHP – kielellä ja siksi se vaatii PHP tulkkia (interpretator). Testiluokkien ja testitapausten laatimisessa käytetään usein hierarkista lähestymistä siten, että yksi testiluokka testaa yhtä luokkaa ja kutakin tila-siirtyvää kohden laaditaan erikoinen testitapaus. PHPUnitin testausympäristö

koostuu ajoympäristöstä ja apukirjastosta. Ajoympäristö toimii riippumatta konsolista, tai kuin meidän esimerkissä NetBeanissa ajettavassa sovelluksessa. Apukirjasto helpottaa testitapausten ohjelmointia. Alkuperäisen PHPUnitin kokokoonpanossa on seuraavia apukirjastoja:

- PHPUnit_Framework_TestCase
- PHPUnit_Extensions_Database_TestCase
- PHPUnit_Extensions_SeleniumTestCase
- PHPUnit_Extensions_TestDecorator

PHPUnit_Framework_TestCase pääapukirjasto tarjoaa testiluokille ohjelmointikehyksen ja helpottaa testiluokkien ja testitapausten kirjoittamisen. Se tarjoaa metodien attribuuttien notaation määrittelyn sekä Assert-luokan vertailujen ja testitapausten hallinnan. PHPUnitin apukirjasto otetaan käyttöön testiluokan alussa lisäämällä PHPUnit_Framework_TestCase kirjasto testiluokan käyttöön komennolla extends PHPUnit_Framework_TestCase.

Testausympäristö tässä opinnäytetyössä on pohjattu Windows Vista koulussa käytettävissä olevalla PC koneella. Siksi alla mainittu testausympäristön asetus muissa ympäristöissä ja käyttöjärjestelmissä saattaa hieman erotella. Yleiset ohjeet opinnäytetyön testausympäristön asentamisesta löytyvät liitteessä 4.



Kuvio 11. Verkkopelin 2 osan testausprojekti

Kun kaikki tarvittavat työkalut ovat ladattu ja asennettu onnistuneesti, on aika luoda uusi projekti NetBeanissa. Koska tässä luvussa keskitytään yksikkötestauksessa ja verkkopelin 2 osa on

jo valmis ja itsenäinen sovellus, pitää luoda uusi projekti käyttämällä olemassa olevaa sovelluksen pohjaa jonka asennusten vaiheista on puhuttu liitteessä 4. Tämän jälkeen testausympäristö on käyttövalmis. Projektin ikkunasta näkee vasta luotu testausprojekti (Kuvio 11).

3.3.2 Yksikkö testausten määrän arviointi, PHP_Depend

Ennen kuin aloitetaan testaus, on pakko arvioida, kuinka monta virhettä odotetaan löytää testistä. Kun asetetaan etukäteen odotusarvo löytyvien virheiden määrälle, on helpompi arvioida testauksen valmiusastetta. Jos löydetään yksikkötestauksessa vähemmän virheitä kuin mitä on odotettu, on syytä tarkistaa testausjärjestelyt ja testauksen kattavuus.

PHP-maailmassa on monta hyödyllistä työkalua, jotka voivat tuottaa tätä arvoa. Tässä tapauksessa valittiin PDepend. PHP_Depend on PHP -ohjelmointikielen pohjoinen työkalu, joka perustuu avoin-lähdekoodiin. Se oli siirretty PHP-maailmalle käyttämällä pohjana jDependin mittaus työkalua, joka mittaa Javan olio-pohjaisen lähdekoodia. Sitä voidaan ladata ja asentaa käsin, mutta parempi ja tehokkaampi on asentaa se jo tutun PEAR luvun 3.3.1 ansiosta. Katso ohjeet tarkemmin (Pichler 2010). Miksi on valittu tämä työkalu? Tämä työkalu tukee täydellisesti NetBeans ja PHPUnit. Esimerkiksi kattamisen raportit, jotka ohjelma tuottaa, tallennetaan XML muodossa, mikä antaa mahdollisuuden käyttää niitä tulevassa testausanalyysissa suoraan NetBeanissa. PHP_Depend on pieni ohjelma, joka suorittaa staattisen koodin analyysin tiettyyn lähteen perusteella. Staattisen koodin analyysi tarkoittaa sitä, että PHP_Depend ensin ottaa lähdekoodin ja jäsentää sen käsiteltävään sisäiseen tietorakenteeseen. Se pystyy generoimaan valtavan määrän tieto-mittareita käyttämällä tietyn koodin osaa. Näillä mittareilla pystyy mittaamaan ohjelmistoprojektin laatua ja ne lisäksi auttavat tunnistaa, missä refaktorointi olisi sovellettava. Ajetaan PHP_Depend testausympäristöön ja havaitaan esille yksikkötestien tarvittava määrä. Avataan konsoli ja syötetään seuraava komento:

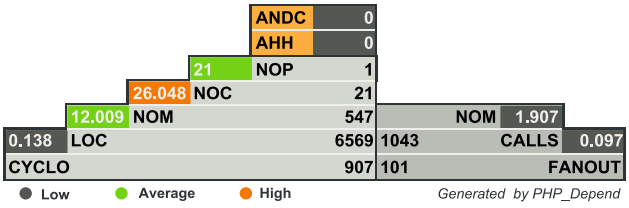
```
pdepend --bad-documentation --summary-xml=PHP_Depend/summary.xml --
jdepend-chart=PHP_Depend/jdepend.svg --overview-
pyramid=PHP_Depend/pyramid.svg C:\xampp\htdocs\verkkopeli\osa_2
```

Työkalu tutkii kaikki notaatio informaatio @-merkkin jälkeen, jotka tavataan lähdekoodissa. Ensimmäinen parametri tarkoittaa että testaus projektissa ei tarvitse niiden tutkimista, koska ne voivat olla epäkäyttöisiä tai puutteellisia. Toinen pdependin parametri märitellee tuottavien tiedostojen suhteellisen polun, ja viimeinen kertoo, missä sijaitsee mittauskohde (osa_2). PHP_Dependin komentorivillä on paljon muita tärkeitä ominaisuuksia, jotka voivat löytyä dokumentaatiosta (Pichler 2010). Niin kuin määriteltiin, kaikki tuottavat mittaus-tiedostot sijoitettiin PHP_Depend kansioon (Kuvio 11). Kansiosta löytyy 2 kaaviota ja 1 XML – tiedosto.

XML-tiedostossa keräytyvät kaikki mitatut arvot XML-hierarkkisessa muodossa, sen ansiosta tuotettiin kaksi mittaus–kaaviota: jdepend.svg ja pyramid.svg.

jDepend kaavio osoittaa paketin ja luokkien riippuvuudet. Se kulkee PHP – luokkien tiedostot ja hakemistot läpi ja luo suunnittelu-laatumittarit jokaiselle luokalle, kuten: luokkien ja rajapintojen määrä, epävakaus ja muut.

Toinen kaavio on pyramidin näköinen (Kuvio 12). Pyramidi-kaaviota käytetään havainnollistamaan täydellisesti ohjelmiston mittareita todella tiivistettynä. Siksi se kerää mittareita perinnön, kytkemisen ja monimutkaisuuden -joukoista, ja liittää ne toisiin.



Kuvio 12. PHP Dependin tuottama pyramidi kaavio

Seuraava lista sisältää kaikki tässä opinnäytetyössä tarvitsevat mittarit, jotka pyramidi-kaavio tuottaa:

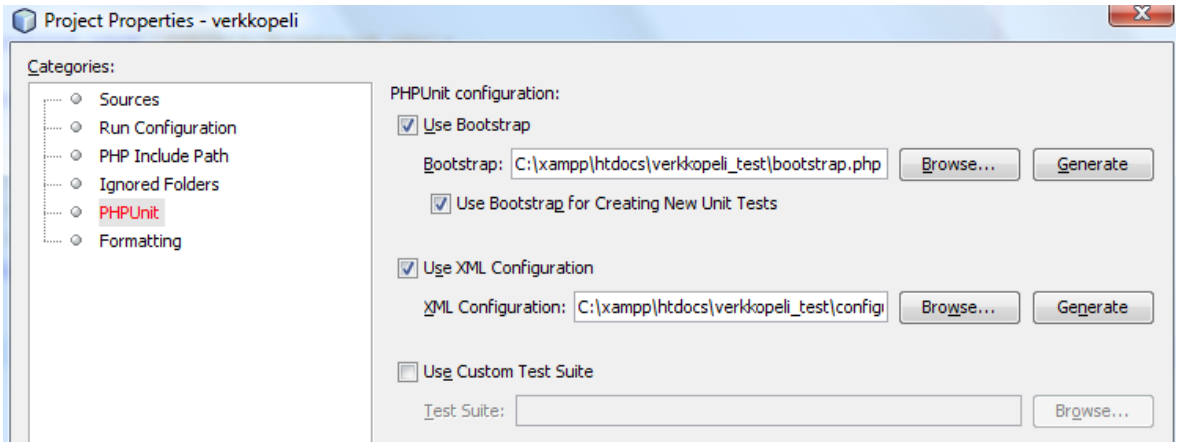
- NOP: pakettien lukumäärä.
- NOC: luokkien määrän mittari laskee julistettujen luokkien määrä.
- NOM: metodien lukumäärä. Laskee kaikki julistetut metodit ja funktiot luokkien sisällä.
- LOC: lähdekoodirivien lukumäärä (ilman kommenttia ja välilyöntiä).
- CYCLO: syklomaattinen numero. Thomas McCaben kehittymä vuonna 1976, joka mittaa ohjelman monimutkaisuutta. Se mittaa riippumatammien polkujen määrää ohjelman lähdekoodissa. Tästä numerosta tarkemmin on kerrottu luvussa 2.3.
- CALLS: tämä vertailuluku laskee, kuinka monta eri metodia ja funktioita on kutsuttu. Se ei laske kaikkia kutsuja, vaan yksi ja samaa metodi tai funktio luokan sisällä on laskettu vain yhden kerran. Tätä laskinta on hyvä vertailla NOM laskimeen.

Pyramidin sivun ulkonemassa sijaitsevat arvot ovat indikaattorit jotka osoittavat keskimäärin kahden vieressä olevan solun arvon (Pichler 2010).

Tässä työssä on tarkoitus toteuttaa yksikkötestaus vurbus_2:n. PHP_Dependin ansiosta. Testausprojektissa on 6596 lähdekoodin riviä ja 547 metodia, eli noin 1 metodi jokaiselle 12 riville. Metodit ja funktiot ovat ajettu 1043 kertaa. Projektissa on 21 luokkaa. Syklomaattinen numero on 907. Niin kuin on kerrottu luvussa 2.9, se kattaa koko lähdekoodin niin, että jokainen lause suoritettaisi vähintään kerran, vaadittaisi 907 polkua tai toisin sanoen 907 testitapausta. Voisi luulla, että 907 on maksimi yksikkötestitapausten määrästä. Tämä työ on hyvin rajattu, siksi on mietittävä perusteellisesti, mitkä kohdat ovat tässä testauskohdassa arvokkaampia ja mitkä niistä ehkä sisältävät avain-virheitä.

3.3.3 PHPUnitin asetukset

PHPUnitissa on 2 asetusten tiedostoa: configuration.xml ja bootstrap.php. Ennen kuin otetaan käyttöön noita tiedostoja, ne on lisättävä NetBeaniin verkkopelin testausprojektiin. Klikkaamalla vasemmalla painikkeella, projektin nimen päällä avataan valikon ikkuna, josta valitaan kohden ominaisuudet (properties). Avatussa ikkunassa klikataan PHPUnit kohtaan (Kuvio 13), ja valitaan sieltä kaksi kohtaa: Use Bootsrap ja Use XML Configuration. Jos tiedostot eivät ole luotu ennen, ne luodaan automaattisesti mallipohjasta. XML-asetuksen tiedostossa (configuration.xml) tallennetaan kaikki asetukset, jotka koskevat ohjelman toiminnallisuutta. Ruudussa näkyy (Liite 1) projektin käytössä oleva configuration.xml esimerkki. Käydään tämä esimerkki läpi. Ensiksi menevät <phpunit> tuntomerkin attribuutit: colors, convertErrorsToExceptions, convertNoticesToExceptions, convertWarningsToExceptions, bootstrap.



Kuvio 13. Verkkopelin osa 2 projektin asetuksen ikkuna

Niillä asetetaan, käytetäänkö vääriä vai ei, huomataan poikkeukset ja missä sijaitsee projektin bootstrap tiedosto. <testsuite> ja <groups> tuntomerkkejä käytetään konsolista kun kokoonpano laaditaan käsin, niille sijoitettaan kaikki testattavat tiedostojen joukot (suite).

Tuntomerkissä <filter> sijaitsee kahta listaa. Ensimmäiseen <blacklist> laitetaan noita tiedostoja ja kansioita jotka halutaan ohittaa. Toiseen <whitelist> laitetaan projektinkansion ulkopuolella olevat tiedostot ja kansiot, jotka halutaan liittää projektiin. Tuntomerkkiin <logging> laitetaan lokkitiedostojen polut ja ominaisuudet. Tuntomerkissä <selenium> sijaitsevat tiedot käytetään automaatio testauksessa, ja tästä nyt ei puhuta. XML-tiedostossa on myös muita attribuutteja joihin voidaan tutustua tarkemmin ohjeesta. (Bergmann 2010.)

Toinen asetuksen tiedosto on bootstrap.php (Liite 2). Tämä tiedosto on erikoinen testausympäristöä varten. Jokaisella projektilla voi olla oma bootstrap. Itse asiassa tämä on tavallinen PHP-tiedosto, missä alustetaan kaikki projektissa käytettävät asetukset. Kaikki asetukset käsitellään tässä tiedostossa, käytetään kaikissa projektiin liittyvissä testitapauksissa. Siksi tähän on sijoitettava esimerkiksi kaikki kirjastot, tietokantaan yhteydet tai kiinteät muuttujat tai alustusasetukset. Kuten tämän työn olevassa esimerkissä alussa valmistellaan alustuksen asetukset niin, kuin virheilmoitusten asetukset ja muistin määrä. Muistin määrä on tärkeä ominaisuus tässä ympäristössä. Mitä suurempi lähdekoodi on, ja mitä runsaammin käytetään metodeja, sitä vaativampi resurssien kulutuksen testitapaus on. Voidaan todeta että PHPUnit on resurssienvaativa ohjelma etenkin, kun se ajetaan NetBeans ympäristöön. Jos muistinmäärä ei ole riittävä, useimmiten se kaatuu. Voidaan suositella, että 128Mb on minimi muistin määrä.

3.3.4 Testitapauksien suunnittelu PHPUnitilla

Ensiksi katsotaan tärkeitä PHPUnitin ominaisuuksia.

- Testiluokan nimi koostuu testattavan luokan nimestä, plus Test-loppuosa.
- Testausta varten useimmiten testausluokka periytyy PHPUnit_Framework_TestCase-apukirjasto.
- Jokainen testi on julkinen metodi, jonka nimi alkaa etuliitteellä "test" kuten näin: public function testPushAndPop().
- Jokaisen testin sisällä, käytetään yhtä Assert-luokan metodia tarkistamaan, mikäli käsittelyn tuloksen on odotettu (enemmän jäljempänä).

Kuten sanottu ylempänä, viimeisessä luetelman kohdassa jokaisessa testitapauksessa käytetään PHPUnitin Assert-luokka. Katsotaan lähemmin sen yleiseestikäytettyjä metodeja ja niiden pääajotuista.

- assertTrue(\$x) Epätosi jos \$x on epätosi
- assertFalse(\$x) Epätosi jos \$x on tosi
- assertNull(\$x) Epätosi jos \$x on voimassa
- assertNotNull(\$x) Epätosi jos \$x ei ole voimassa
- assertIsA(\$x, \$t) Epätosi jos \$x ei ole luokka tai tyyppi \$t
- assertEquals(\$x, \$y) Epätosi jos \$x == \$y on epätosi
- assertNotEquals(\$x, \$y) Epätosi jos \$x == \$y on tosi
- assertIdentical(\$x, \$y) Epätosi jos \$x === \$y on epätosi
- assertNotIdentical(\$x, \$y) Epätosi jos \$x === \$y on tosi
- assertCopy(\$x, \$y) Epätosi jos \$x ja \$y ovat samat muuttujat
- assertWantedPattern(\$p, \$x) Epätosi ellei regex \$p täsmää \$x
- assertNoUnwantedPattern(\$p, \$x) Epätosi jos regex \$p täsmää \$x
- assertNoErrors() Epätosi jos virhe on tapahtunut
- setExpectedException (\$x) Epätosi mikäli PHPn virhe ilmoitus ei saatu.

(Bergmann 2010.)

Tässä mahdolliset Assert -vertailut eivät ole rajoitettu. Kaksi hyvin yksinkertaista tästä joukosta ovat assertFalse () ja assertTrue (). Ne tarkistavat, mikäli tuloksena on epätosi tai tosi vastaavasti. Seuraavaksi tulevat assertEquals () ja sen vastakohta assertNotEquals (). Niiden käytössä on omia nyansseja. Joten kun verrataan liukulukuja, ne voivat ilmoittaa tarkkuutta. Samoin näitä metodeja käytetään DOMDocumentin luokkainstanssien vertauksessa, taulukoissa ja kaikessa olioissa. Olioissa yhdenvertaisuus asetetaan siinä tapauksessa jos luokkien attribuutit sisältävät samat arvot. Myös on syytä mainita assertNull () ja assertNotNull (), jotka tarkistavat mikäli parametrin tietotyyppi on NULL (älä unohda, että PHP:lla tämä on erillinen tietotyyppi).

Katsotaan tilanteet, joissa jotkut testit on pakko ohittaa jostakin syystä. Esimerkiksi silloin, kun testausympäristössä puuttuu jonkinlainen mahdollinen PHP-laajennus, voi merkittä sitä lisäämällä koodiin seuraavaa:

```
$this->markTestSkipped('Extension is not loaded.');
```

Mikäli testi on kirjoitettu koodia varten, mutta koodi ei ole vielä kirjoitettu loppuun, tai se puuttuu kokonaan, voidaan merkitä sen toteuttamattomana:

```
markTestIncomplete()
```

PHPUnitin testiluokka on tavallinen oliopohjainen-luokka. Alussa määritellään luokan attribuutteja. Testiluokalle on mahdollista kirjoittaa konstruktori-tyyppinen metodi, jolla voidaan alustaa testitapausten yhteisiä muuttujia tai olioita tai luoda vain tässä testitapauksessa käytettävä ympäristö. Tällainen metodi merkitään SetUp notaatiolla. Vastaavasti voidaan kirjoittaa destruktori-tyyppinen metodi TearDown notaatiolla, jolla voidaan hallita testauksen suorituksen jälkeen tehtäviä ja toimintoja, esimerkiksi testitietokannan palauttamiseksi oletustila.

Ohjelmistotestaus on kombinatorinen ongelma. Esimerkiksi jokaisen Boolen muuttujan mahdollinen arvo vaatii kaksi testiä: ensimmäinen vaihtoehto kun se on tosi, ja toinen, kun se on vastaavasti epätosi. Tämän vuoksi jokaiselle lähdekoodin riville tarvitaan noin 3-5 riviä testauskoodia. Niin kuin selitettiin luvussa 3.3.2, osa_2:lle tarvitaan 907 testitapausta täysi kattamaan kaikki lähdekoodit päällä, mikä on mahdotonta tässä opinnäytetyössä. Koska tämä työ on kvalitatiivinen eli laadullinen -tyyppinen tutkimus, on tärkeä etsiä uutta teoriaa oppimisessa, saamalla uusia tietoja testausprosessista. Siksi ei ole pakko toteuttaa nämä kaikki testit, vaan pyritään analysoimaan huolellisesti, mitkä osat ovat todella tärkeitä tässä ohjelmassa, mitkä metodit ovat tärkeitä, virheettömyyden näkökylmästä. Valitaan noin 2-4 testitapausta, kehitetään ne tässä työssä ja analysoidaan niiden tulokset luvussa 3.5 ja 3.6.

3.4 Testausprosessi

Kun testiympäristö on luotu ja PHPUnitin toiminnallisuudet ovat tutkittu, siirrytään testitapauksen tekemiseen. Mutta alusta täytyy määritellä, mitä testataan tässä työssä. Niin kun oli määrätty ennen, valittiin 2 – 4 testitapausta. Tutkitaan ohjelman rakenne (Kuvio 10). Ohjelman arkkitehtuurin rakenne on sekava. PHP-kielellä tällainen on mahdollista. Ohjelmat voidaan kirjoittaa sekä proseduurisella, että olio-pohjaisella lähestymistavalla. Ohjelman rakenteessa voidaan huomata kolme isointa ydintä: prechoice.php, check_fields.php calculator.php.

Prechoice.php on kontrolleri, johon kokoontui metodeja, jotka ohjaavat ohjelman ulkoasua, muotoilua, navigaatiota, ja näyttävät sisältöä. Lisäksi siellä alustetaan ohjelman esiasetukset, käyttäjät, ryhmät yms. Prechoice.php on pääkontrolleri ja siksi se on testattava luultavasti viimeisenä, koska se vaatii yksikkötestausten lähestymistapa alhaalta ylöspäin (Katso luku 2.1.4). Lisäksi muotoilu, navigaatio ja ulkoasu, kuten on sanottu luvussa 3, yleisesti web-pohjoisen sovelluk-

sessa suoritetaan integraatiotason testauksessa, siksi tämän kontrollerin testaus jätetään ulkopuolelle tätä teosta.

Check_fields.php on Ajax-kyselyn käsittelyn kontrolleri. Siellä kokoontuivat kaikki metodit jotka vastaavat ohjelman logiikkaa ja toiminnallisuutta. Kaikki kyselyt saapuivat siihen $_GET PHPn globaalisen muuttujan avulla. Sitten nämä kyselyt jaetaan erilaisiksi osiksi ja käsitellään tarkoitetuilla juuri niille metodeilla. Lopussa tulos palautetaan JavaSriptille takaisin AJAX-vastauksena. Tämä on riittävästi mielenkiintoinen kontrolleri yksikkötestauksen näkökylmästä, mutta valitettavasti se ei ole kirjoitettu olio-ohjelmoinnin lähestymistavalla, niin kuin sitä vaati PHPUnit. Siksi tämänkin kontrollerin yksikkötestaukset jätetään ulkopuolille.

Caculator.php kontrollerilla on tärkeä tehtävä (Kuvio 17), se on käytössä lopputuloksien laskemissa. Siellä on vain ainoa metodi *caculateQuarterResult*, joka vastaanottaa tulorivin, käsittelee sitä laskemalla kaikkia arvoja, jotka ovat käytössä neljännesvuosikatsauksessa sekä vuosikertomuksessa. Se hakee xml_reader.php-kontrollerin ansiosta staattisia tietoja erilaisista XML-tiedostoista. Lisäksi se vaatii yhteyttä tietokantaan, josta se hakee muita tarvittavia käsittelyn arvoja, tässä auttaa *caculationDAO.php* luokka. Kun kaikki arvot ovat laskettu, ne on sijoitettava tietokantaan. Myös XML-tiedosto averageResults.xml on päivitettävä xml_writerin kontrollerin avulla. Lopputuloksena kontrolleri palauttaa käsitellyn onnistumistilan.

3.4.1 Testitapaus 1

Keskitytään tässä testitapauksessa caculator.php kontrolleihin (Liite 4). Ensiksi kokeillaan syöttää yksi testausjoukko hotellia varten. Ajamalla sitä saadaan tulokseksi virheellinen ilmoitus: ”Only variables should be passed by reference”. Virhe tapahtui caculateQuarterResult() metodien kutsuessa CaculationDAO tietokannan käsiteltävässä luokassa. Tässä tapauksessa kaikki muut testaustoimet caculator.php luokan kanssa ovat mahdottomia kunnes ei saada korjata tätä vikaa CaculationDAO luokassa.

3.4.2 Testitapaus 2

Kuten selvisi luvusta 3.4.1, Caculation-luokan testaus paljasti vakavan virheen, kutsuttussa aliluokassa caculationDAO.php. Tässä testitapauksessa pyritään selvittämään ongelman tausta caculationDAO-luokan testatessa ja valitsemalla sieltä ne metodit, jotka ovat käytössä caculation luokassa. Sieltä löytyy 3 tällaista (Liite 5): insertHotelResult(),insertRestaurantResult() ja

insertHotelAndRestaurant. Nämä kaikki metodit vastaanottavat 2 parametria: käyttäjän ID, ja $hotelResultin olio.
Koska kaikki yllämainitut metodit vastaanottavat olio-tyyppisen parametrin, niiden simulointiin ei riitä tavallista luku- tai merkkityyppistä syötettä, vaan ne vaativat olioita. Siksi pyritään simuloida olioita luomalla testausmetodeille oliopohjaisia $mock muuttujia. Niiden rakentamista auttavat luokkien konstruktorit, joilla on jo esiasennettu luokkien oletuksen attribuutit. Tässä käytetään assertNotEquals metodia jolla on sama tarkoitus kuin assertEquals, mutta vastakohtainen, se palauttaa merkin tosi jos tulos ei ole samanlainen kuin odotettu. Ajettaessa tämän luokan NetBeanissa tuloksi saadaan jo tuttu entisestä testitapauksesta virhe: "Only variables should be passed by reference". Tämä on luultavasti virheen tausta, joka esti caculation-luokan testi loppuun suoritusta. Pyritään selvittää mistä on ongelman tausta PHPn dokumentaatiosta ja havaitaan että ongelma johtui vanhan metodin käytöstä (Propel 2007). On havaittu että bindParam () DAO – luokan metodi joskus aiheuttaa tämän tyyppisiä ongelmia ja sen sijaan on käytettävä metodia bindValue(). CaculationDAOn lähdekoodin muokkaamisen jälkeen ajetaan testitapaus uudelleen:

```
...................

Time: 1 second, Memory: 8.00Mb

OK (3 tests, 3 assertions)
```

Tulosruudulta käy selväksi että testi on läpäisty onnellisesti ja kaikki 3 kolme metodia on suoritettu.

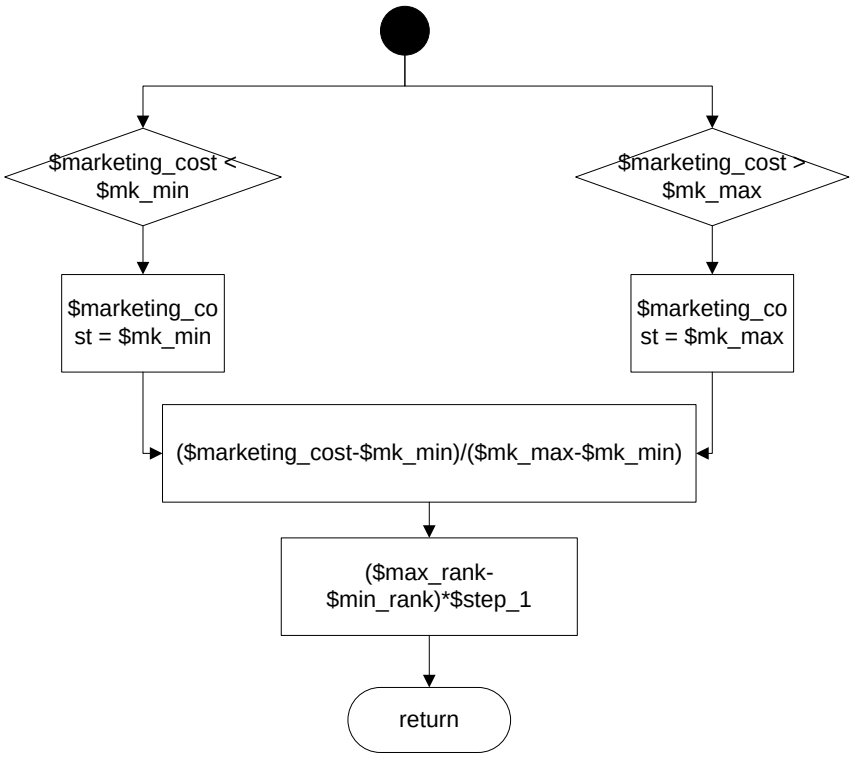
3.4.3 Testitapaus 3

Käymällä läpi Caculator-luokka ja analysoimalla sen sisältöä kävi ilme, että sen lauseissa usein kutsutaan caculation_formula.php luokka johon kokoontui pääosin matemaattisia metodeja. Verkkopelin 2 osa on liiketalouden peli, jossa käytetään paljon matemaattista logiikkaa, siksi on tärkeä testata sen matemaattisia kaavoja ja niiden tuomia tuloksia. Aikataulun rajoituksen takia valitaan kaksi erityyppistä metodia ja kehitettään niiden testausta. Ensisijaiseksi kiinnostus kohdistuu noihin, metodeihin joissa käytetään jako operaatiota. Ne voisivat johtaa vakaviin virheisiin. Tälle kriteerille sopii caculateOccupancy metodi (Liite 6a). Tämä metodi laskee huoneiston täyttöaste, sen sisätuloparametreina vastaanottaa kvartaaleittain huoneiston täyttöasteen ja hotellin huoneiston määrän. Tarkoitus on syöttää laiton parametri ja tutkia ohjelman

toimintaa. Muokataan testCaculateOccupancy testausluokan metodi, lisäämällä sen koodiin seuraavan muutoksen;

```
public function testCaculateOccupancy() {
        $this->assertEquals(-2, $this->object->caculateOccupancy(100, 0)
        );
    }
```

Testin ajettaessa jälkeen ruudulle tulee virheilmoitus (Liite 6a) nollan jaosta. Tämä testi osoitti ongelman moduulin rakenteessa, nimittäin nollan jako ei ole hoidettu tässä metodissa oikealla tavalla, kuten esimerkiksi TRY … CATCH poikkeuksien käsittelytavalla. Ehkä tämä ohjelman ongelma on hoidettu jossain muualla, mutta luotettavuuden näkökylmästä tämä on epävakava tilanne. Testitulosta katsoessa, voidaan ymmärtää, että CaculateOccupancy-metodin testauksessa luodaan luokan instanssi, kutsutaan testattava metodi ennakko-valmistettuna parametreina, ja tarkistetaan, mikäli lasketustoimet on suoritettu oikein sen tulosten palauttaessa. Tähän tarkoitukseen tarvittiin *assertEquals*() metodi, jonka ensimmäisenä pakollisena parametrina määritetään odotusarvo, toisena vastaanotettu tulos ja lopussa tarkistetaan niiden yhtäpitävyys.



Kuvio 14. caculateRank vuokaavio

Seuraavassa testitapauksessa ruvetaan etsimään arvoalueen rajoja (Liite 6b). Testiksi valitaan caculateRank metodi. Se vastaanottaa 5 parametria: marketing_cost, mk_min, mk_max,

max_rank, min_rank. Ne kaikki ovat numeroituja ja sisältävät talouspohjaisia tietoja. Sitten siellä on kaksi ehto-metodia IF ja ELSEIF (Kuvio 14). Liikkumalla kuvalla alaspäin, havaitaan kaksi kaaviota, joissa käsitellään vastaanotettu parametrit ja lopussa saatu tulos palautetaan takaisin kutsujalle. Tarkoitus on testata tämä metodi ehtokattavuudella 100:ksi % ja löytää mahdolliset arvoalueen rajat.

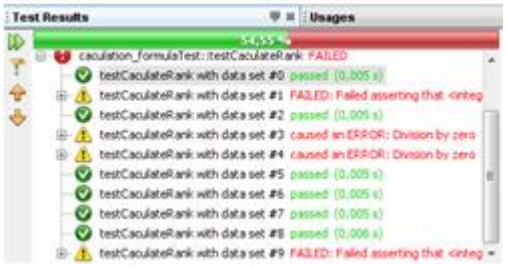
Tähän tehtävän suorittamiseen auttaa *muunnettu ehtokattavuuden* menetelmä josta oli puhuttu luvussa 2.2.2. Tämän menetelmän mukaan jokaisen saadun parametrin pitää vaihtaa tilansa molempiin tilaan tosi ja epätosi. Samaan aikaan muiden parametrien arvot pysytään samana. Ottamalla tämä perusta, suunnitellaan testijoukot tällaisille metodeille. Taulusta näkee, että jokaiselle tarkistettavalle parametrille on väritetty vaalea-sininen tausta. Kaikkien muiden parametrien joukot ovat kiinteät. (Taulukko 1.)

Taulukko 1. Muunnettu ehtokattavuuden menetelmän matriisi

Testitapaus	marketing_cost	mk_min	mk_max	max_rank	min_rank	tulos
1	1	0	1	0	1	
2	0	0	1	0	1	
3	1	1	0	1	0	
4	1	0	0	1	0	
5	0	1	1	0	1	
6	0	1	0	0	1	
7	1	0	1	1	0	
8	1	0	1	0	0	
9	0	1	0	1	1	
10	0	1	0	1	0	

Nyt tulee aika kirjoittaa PHPUnit testi caculateRank metodille (Liite 6b). Otetaan runko testCaculateRank metodia. Tässä metodissa käytetään PHPUnitin hyödyllinen ominaisuus: tiedonantaja (data provider). Tiedonantaja märitellään @dataProvider notaatiolla missä parametrina käytetään metodin nimi, joka välittää haluttu tietoja. Sen sisällä voivat olla erilaisia metodeja ja menetelmiä jotka voivat hakea tietoja erilaisista lähteistä kuten nimittäin tietokantoja, XML – tiedostoja tai niin kuin esimerkissäkin taulukko. Siihen on sijoitettava suunnitellut testauserät,

jotka sitten välitetään testCaculateRank() metodille. Kun testi on valmis, ajetaan se ja tuloksena saadaan seuraava Netbeanin Test Resultsin ikkunan tila (Kuvio 15). Tästä testitapauksesta selvitettiin tilanteita, joissa testCaculateRank metodi kaatuu. Niihin vastattiin käyttötapaukset 2,4,5 ja10. Kaksi niistä 4 ja 5 ovat nollan-jako virheet ja tästä oli jo puhuttu entisessä testissä, kun testattiin caculateOccupancy metodi.

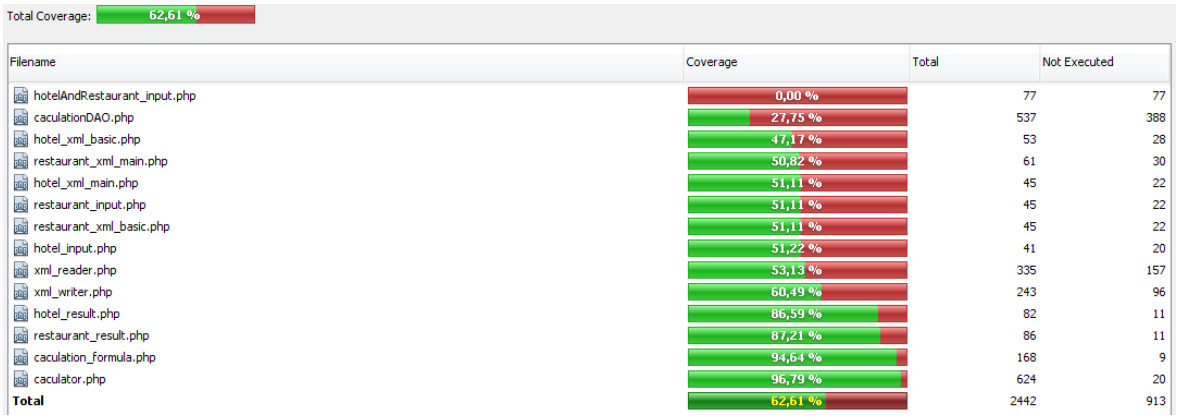


Kuvio 15. NetBeansin Test Results ikkuna toisen testipauksen jälkeen

Lisäksi testistä selvitettiin, että muunnettu ehtokattavuuden menetelmä on todella tehokas, jos tavallisissa menetelmissä testaamiseen kattamalla 100 %:ksi, sellainen metodi tarvitsee 2^n, jossa **n** on operandin (parametrin) määrä. Siinä tapauksessa testitapausten kokonaismäärä tulisi 2^5 = 32, mutta käyttämällä muunnettua ehtokattavuuden saatua 10 testitapausta, mikä on merkittävästi pienempi.

3.4.4 Testitapaus 1 (jatku)

Kun virhe joka estää testin täytäntöönpanoa on eliminoitu, voidaan jatka caculation-luokan testaamista. Kuten näkyy testitapauksesta, testaamaan kaikki 3 ELSE … IF rakenne-polkua vaaditaan 3 testausjoukon. Luvussa 3.4.1 käytetty testCaculateQuarterResult testausmetodi sisältää vain yhden testausjoukko, siksi sitä on vähän muokattava, myös siihen on lisättävä tiedonantaja sama kuin käytetyssä luvussakin 3.4.2 (Liite 4). Kun toinen versio testaus metodia on valmis, voidaan ruveta testaukseen. Tässä käytetään assertNotNull PHPUnitin assert luokan metodi. Se ei vaadi odotustuloa vaan riittää että palautuessa ei tule NULL. Kun testi on suoritettu (Kuvio 16), voidaan arvioida NetBeansin sisäänrakennettua hyvää ominaisuutta, joka käyttämällä PHPUnitin ulkotulevia tietoja rakentelee testien kattavuuden raportin (Code Coverage Report).



Total Coverage: 62,61 %

Filename	Coverage	Total	Not Executed
hotelAndRestaurant_input.php	0,00 %	77	77
caculationDAO.php	27,75 %	537	388
hotel_xml_basic.php	47,17 %	53	28
restaurant_xml_main.php	50,82 %	61	30
hotel_xml_main.php	51,11 %	45	22
restaurant_input.php	51,11 %	45	22
restaurant_xml_basic.php	51,11 %	45	22
hotel_input.php	51,22 %	41	20
xml_reader.php	53,13 %	335	157
xml_writer.php	60,49 %	243	96
hotel_result.php	86,59 %	82	11
restaurant_result.php	87,21 %	86	11
caculation_formula.php	94,64 %	168	9
caculator.php	96,79 %	624	20
Total	62,61 %	2442	913

Kuvio16. Lähdekoodi kattavuuden raportti (Code Coverage Report)

On hyvä, että tässä raportissa kokoontuivat kaikki tiedostot, jotka osallistuvat caculation – luokan testaukseen. Myös sieltä on ilmennetty kaikkien läpi käytettyjen rivien kokonaismäärä. Kun näkee kuvasta, caculator.php on katettu 96,74 %:lla, mikä tarkoitta, ettei kaikki lauseet ole suoritettu loppuun. Tähän poikkeamaan kuuluvat mm. kaikki poikkeuksen metodit jotka on testattava erikseen. Testauksessa myös on testattu kaikki käytetyt olio-luokat ja niihin kuuluvat metodit, lisäksi on testattu caculationDAO ja caculation_formula aliluokat. Tässä tilassa testi-tapaus voidaan katsoa päätettynä.

3.5 Tulokset

Tutkimuksesta selvitettiin, että web-pohjainen sovellusten testaus hiukan erottelee tavallisesta sovellus-testauksesta. Esimerkiksi navigaatio, sen toiminnallisuus ja sisältö testataan ensiksi ja vasta sitten liikkumaan yksikkö, integraatio ja järjestelmä-tason testaus. Mutta itse yksikkötestit suoritetut Verkkopelin 2 osan testausprojektissa eivät paljon poikkea mistä tahansa muusta sovelluksen testauksesta.

Tutkittiin PHPUnitin testausympäristön rakenne ja asentaminen. Havaittiin miten toimivat yhdessä NetBeans ja PHPUnit yhdistelmä ja mitä etuja tämä voisi antaa testaajille. PHP_Depenedin työkalun avulla löydettiin maksimi vaativa testienmäärä kattamaan koko verkkopelin 2 osan 100 %:ksi. Mittauksen jälkeen, opittiin käyttämään pyramidi-kaaviota, joka hakee valtavan määrän avainarvoja ja esittää niitä sopivassa muodossa. Tutustuttiin PHPUnitin kehykseen, sen avain toiminnallisuuteen ja sen pää luokan Assert, jonka funktioita käytettiin kaikissa testitapauksissa. On selvitetty että yksikkötestauksen pohjalla ohjelmoijia pystyy hel-pompi ja nopeampi toteuttaa lähdekoodin parantamisen eli *refactoring*. Ensimmäisessä testita-pauksessa testattiin caculation luokka, johon vuorovaikuttavat muut luokat ja aliluokat. Mutta

testi meni pieliin, siksi päätettiin jäljittää tarkemmin sen suhteet muihin luokkiin ja testata niitä ensisijaisesti, eli valittiin alhaalta ylös testaus menetelmä.

Toiseksi testattiin caculation_formula luokka ja sen funktiot jotka yhteistoimivat caculation luokan kanssa. Siinä tapauksessa kaikki testit osuivat pääosin matematiikan toiminnallisuuteen. Siellä käytettiin uusittua ehtokattavuuden menetelmää, jonka avulla saatiin kattamaan kaikki ehtolauseet 100 %:ksi pienillä testien määrällä.

Kolmannessa testitapauksessa testattiin caculationDAO luokkaa, joka yhteistoimi tietokannan kanssa ja tahdistaa tiedot caculation luokan kanssa. Tästä luokasta johtui ongelmia caculation luokkaa testatessa, mutta niitä onnistui löytymään ja korjaamaan.

Ja viimeisessä testauksessa testataan caculation luokka uudelleen kolmena eri testausjoukkona saadaksemme polkujen kattavuus 100 %:ksi, mutta saatiin 96,79 % sen takia, että ei ole testattu poikkeuksien metodit. Tämän projektin tuloksia analysoimalla selvitettiin että saavutettiin kaiken kaikkia 62 % testien kattavuutta kaikista toteutetuista testitapauksista.

Kaikki testit ovat toteutettu valkoisen laatikon menetelmän mukaan. Yksikkötestaukset auttoivat poistaa epäilyksiä yksittäisiän moduulia kohden. Niitä voidaan käyttää perustana alhaalta ylös testauksessa: ensiksi testataan yksittäiset ohjelman osat, sitten koko ohjelman kokonaisuudessa. Testauksessa on havaittu kaksi sovellusten virheentyyppiä: nollan jako ja "Only variables should be passed by reference" ja annettu ehdotukset niiden korjaamiseen.

3.6 Arviointi

Tässä opinnäytetyössä pyrittiin suunnitella ja toteuttaa yksikkötestit, jotka kohdistavat verkkopelin osan 2 moduuliin. Yksikkötestit voidaan lopettaa siinä vaiheessa, kun suunnitellut testiajot on ajettu hyväksyttäväksi. Loppuen lopuksi suoritettiin 3 testitapausta, joihin osallistui mm. 14 verkkopelin osan 2 tiedostoa ja niiden testaamisessa saatiin 62 % katettuna. Ei ole huono tulos opinnäytetyön sisällä tällaisena pienenä testitapausten määränä, jos lasketaan että normaalissa elämässä projektissa, kohtuullisena pidetään 70 – 85 %, koska yleisesti saavuttaa 100 % katettuna on melkein mahdotonta ja järjetöntä.

Rajoitusten takia ensiksi halutaan testata sovellus ylhäältä alas lähestymistavalla, koska tämä kuin luultiin voi säästää aikaa muihin testitapausten tekemiseen, eli ensin testataan pääluokka, ja sen jälkeen, jos virheitä on havaittu, poistetaan peräkkäin kaikki testausta epäilyt luokat ja

osat mikäli ne ovat virheellisiä. Mutta vasta alussa kohdattiin ongelma jonka takia ei ollut mahdollista jatkaa tuntemattoman virheen takia. Koska jotkut luokat voivat käyttää muita luokkia, yksikkötestaus usein ulottuu myös niihin liittyville. Esimerkiksi ohjelmoija kirjoittaa testausluokkaa ja huomaa, että käytettävä tietokannan luokka on vuorovaikutuksessa tietokannan kanssa. Tämä on virhe, koska normaalisti yksikkötestin ei pitäisi ylittää testattavan luokan rajoja. Tämän seurauksena kehittäjän on kieltäydyttävä tietokantayhteydestä ja sen sijaan on käytettävä omaa mock-oliota. Siksi jouduttiin vaihtamaan lähestymistapaa ja etsimään virheitä alhaalta ylöspäin. Käymällä läpi kaikki metodit ja analysoimalla niitä löydettiin ongelmien kohdat. Jos nämä metodit riippuivat jostain muista ulkopuolella sijaitsevista luokista tai muuttujista, pyrittiin korvaa niitä mock-olioilla, joihin sijoitettiin simuloitavaa tietoa. Simuloitava tieto, tai testausjoukon aineisto otettiin pääosin käytettäviin tietokannasta, tai virheenkorjaus toiminnoista (debugging). Sitten testaamalla niitä saatiin varmuutta, että ne toimivat oikein. Ja vasta silloin testattiin ylhäällä sijaitsevia luokkia.

Testauksessa selvitettiin NetBeans ja PHPUnit yhdistelmän heikkoja ja vahvoja kohteita. NetBeansin vahvuuksiin voi liittää sen ystävällinen käyttöliittymä, kaikki funktiot ja toiminnot sijaitsevat harkitusti harkituissa paikoissansa, etenkin niille käyttäjille, jotka avaavat tätä ohjelmaa ensimmäistä kertaa. Voidaan tehdä yhteenveto että sen käyttöliittymä ei olekaan niin hassu kuin monissa muissa kehitysympäristöissä. Mutta sillä ovat myös omat heikot puolet. Koska tämä on suhteellisesti uusi ohjelmisto, projekti saa ensimmäisen lisenssiansa vasta vuonna 2006 (Wikipedia 2010), se ei ole ollut vielä riittävästi kestävää. Joskus jonkun testitapauksen suorittaessa se voi ehkä mennä jumiin tai kaatua virheellisesti.

PHPUnit on loistava työkalu, mutta siinä havaittiin myös muuttamia ongelmia. Se on suhteellisesti vaativaa resursseja. Tämän työn testausympäristössä jouduttiin nostamaan käytettävissä oleva muistin määrä 256 Megatavuksi, koska ennen tätä sovellus kaatui monta kertaa. Lisäksi heikkouksiin voidaan laskea testausympäristön esiasennon monimutkaisuus. Voitaisiin olla varma, että kukaan ei pystynyt ajamaan testejä ilmaan tehtyä monta kertaa muutosta PHPUnitin asetusten tiedostokseen. Mutta tässä auttaa runsaasti oleva internetissä dokumentaatio. Lisäksi heikkouksiksi voidaan liittää lopullisien testien virheenkorjaus (debugging), koska on tosi vaikea jäljittää ongelman kyse, jos testi jostain syystä meni pieleen. Siinä tapauksessa virheenkorjaus ei toimi oikein monen rajoituksien takia. Ja lopuksi voidaan sanoa, että sillä pysty ajamaan vain oliopohjaisia PHPn lähdekoodeja. Tämä myös liittyy sen heikkouksiin. PHP kieli, itse asiassa on sekainen kieli, sillä on mahdollista kirjoittaa sovelluksia sekä oliopohjaisena että proseduurisena lähestymistavalla. Tämä luultavasti johtuu siitä syystä, että PHPUnit käyttää

alkuperäisesti testaamaan koodattujen Java-pohjaisten sovelluksien JUnitin perustaa. Tästä syystä valitettavasti Check_fields.php ja monen muun tiedoston testaus, jossa kokoontuivat tärkeät funktiot ohjelman toiminnallisuuden näkökylmästä, on jätetty ulkopuolille.
Mutta jopa kaiken kaikkea PHPUnitin heikkouden lukuun ottamatta voitaisiin pitää yhteenvetona se, että PHPUnit joka tapauksessa on hyvä keino testaamaan PHPn kielisija oliopohjaisia luokkia. Jopa tässä hyvin rajoitettu yksikkötestauksessa on löydetty 2 uutta ohjelman virhettä, mikä osoittaa, että PHPUnit on toimiva ja monipuolinen työkalu.

4 Pohdinta

4.1 Yhteenveto

Tämän opinnäytetyön päätavoitteena oli tutkia testausprosessia ja valitsemalla joku yksi niistä, tutkia sen teoriaa tarkemmin ja harjoiteltaessa testausympäristössä, pyrkiä käydä läpi alusta loppuun kaikki valitut testausprosessien vaiheet ja tutkita niiden tärkeitä periaatteita. Tähän oli valittu yksikkötestaus prosessi.

Ohjelmistoprosessin tavoitteena on kehittää tarpeiden ja vaatimusten mukainen, virheetön ja toimiva ohjelmisto. Jotta ohjelman oikea toiminta voidaan varmistaa, se täytyy testata. Testausta voidaan tarkastella sekä osana ohjelmistoprosessia että omana testausprosessinaan. Tässä opinnäytetyössä auttoi PHPUnit ja NetBeans testausympäristö.

PHPUnit on testaustyökalu PHP-kielisen ohjelman testaamiseen. PHPUnitin lähdekoodi ja asennuspaketti ovat Internetissä avoimesti saatavissa. Se on osa niin sanottua xUnit-tuoteperhettä, johon kuuluu yleisimmin ohjelmointikielen testausympäristö. NetBeans ja PHPUnitin ajoympäristö mahdollistaa testitapausten suorittamisen ja tulosten raportoinnin vaivattomasti. PHPUnitin runko (Framework)-apukirjasto helpottaa testitapausten toteuttamista notaatioiden ja Assert-luokan avulla. NetBeans ja PHPUnit-testausympäristön kaltainen työkalu mahdollistaa yksikkötestaamisen järjestämisen pienellä työmäärällä ja on siksi varteenotettava osa ohjelmistokehittäjän kehitystyökaluja. Opinnäytetyössä havainnollistettiin PHPUnitin testitapaukset käyttää olion sallittujen ja virheellisten tilojen testaamisessa käytäntö esimerkkien avulla. Kaikki nämä esimerkit ovat toteutettu valkoisen laatikon periaatteen mukaan.

NetBeans ja PHPUnit-testausympäristön käyttöä on siis mahdollista laajentaa yksikkötestaamisesta koko järjestelmän testaamiseen. Tutkielmassa perehdyttiin lähinnä yksikkötestaukseen PHPUnitin testausympäristössä, joten tutkielmaa voisi jatkaa perehtymällä syvällisemmin integrointi-, järjestelmä- ja regressiotestausvaiheisiin sekä tutkia testausympäristön soveltumista ja hyödyntämistä kyseisissä testaustasoissa.

On todettu, että valkoisen laatikon menetelmä on hyvä ja tehokas tapa havaitsemaan virheitä yksikkötasolla, mutta niin kuin selvitettiin että kaikki testaustekniikat, myös yksikkötestaus ei salli havaita kaikkia ohjelman virheittä. Itse asiassa tämä on mahdotonta jäljittää kaikki mahdolliset ohjelman suorituspolut, lukuun ottamatta hyvin yksinkertaisia tapauksia. Lisäksi kukin moduuli testataan erikseen. Tämä tarkoittaa, että integraatio ja järjestelmä -tason sisältyviä

funktioita ja suoritettavia useissa moduuleissa virheitä eivät paljasta. Lisäksi tällä tekniikalla on turha testata suorituskykyä. Näin yksikkötestaus on tehokkaampaa, jos sitä käytetään muiden testausmenetelmien yhteydessä.

4.2 Johtopäätökset

Saadakseen hyötyä yksikkötestauksesta on ehdottomasti käytävä testauksen teknologian koko ohjelmiston kehitysprosessi läpi. On pakko pitää muistiinpanot kaikista tehdyistä moduuliin muutoksista. Tätä tarkoitusta varten esimerkiksi käytetään versiointihallinnan ohjelmistoa. Tätä esimerkiksi käytetään hyväksi regressio testauksessa. Niinpä, jos uudemman version ohjelmisto ei läpäise testiä läpi, joka meni onnistuneesti aikaisemmin, ei olisi vaikea tarkistaa lähdekoodin versiot ja korjata virheet. Lisäksi täytyy varmistaa jatkuva testien seuranta ja analyysi epäonnistuneista testeistä. Jos jättää tämän huomioon ottamatta, se johtaa lukuisiin testien epäonnistumisiin.

On otettava huomioon, että onnistuneelle testausprosessille voi hyvin auttaa ajoissa laadittu ja standardien mukaan kirjoitettu lähdekoodin dokumentaatio. Sitä voidaan käyttää esimerkiksi testien mittauksessa. Niin kuin on tutkittu tässä, opinnäytetyössä PHPDepend työkalulla voi hyväksi käyttää esidokumentoitua lähdekoodia saavuttaa tarkkaa mittaustulos.

Lisäksi on otettava huomioon, että mielellään on otettava PHP-pohjaisia sovelluksia oliopohjaisen lähestymistavalla mukaan, joten väittä monia ongelmia. Nykyään valtaosa käytettävistä olevista työkaluista ovat oliopohjain-suunnattuja.

Tämän opinnäytetyön perusteella selvitettiin että hyvä testauksen lähestymistapa, on automatisoida kaikki, mikä on mahdollista. Vaikka tässä työssä ei ole puhuttu paljon tästä aiheesta, voidaan tehdä johtopäätös, että automatisointi on tärkeä tekijä, etenkin isoissa projekteissa. Näin voidaan säästää aikaa ja jossakin tapauksissa resursseja. Mutta siinä vaiheessa on aina muistettava, että ihan kaikkea automatisoida varmasti ei tarvitse ja useimmiten se on mahdotonta. Ja päätekijän arvo on kannattavuus. Siksi on hyvin harkittava, kun kyse on automatisoinnista.

Tässä opinnäytetyössä testaus kohtana oli verkkopelin 2:n osa. Mutta testausprosessimallia käyttäessä tässä työssä voitaisiin soveltaa myös muille verkkopeli-projektin osille, tai jopa muille PHP-pohjaisille sovelluksille, koska periaatteet joiden mukaan rakennettiin testejä, ovat standardinmukaisia ja pohjautuvat kokonaan yleiseen testausteoriaan joka on suurin pirteissä yhtä sama erilaisissa ympäristöissä ja ohjelmointiarkkitehtuureissa.

4.3 Jatkokehityksen ehdotukset

Yksikkötestaukset aikana havaittiin niiden tärkeä ominaisuus, ne ovat erityisesti hyödyllisiä regressiotestauksessa. Niiden ansiosta ohjelman lähdekoodi katetaan testaus peitteellä, niin, että jos tehdään muutoksia johonkin ohjelman osaan ja näissä muutoksissa on myös virheellinen sivuvaikutus muihin osiin, siitä selvästi näkyisi testien tulos. Siksi on tärkeä toteuttaa yksikkötason testejä. Mutta yksikkötestien suunnittelu ja teko on aika työläs tehtävä. Lisäksi jokaisessa regressiotestauksen iteraatiossa toistetaan paljon samanlaisia ja joskus turhia toimintoja. Siinä tapauksessa on aina automatisoinnin tarve.

Tehtäessä tätä työtä selvittiin että se vaatii jonkin verran automatisointia. Siksi tämän työn jatkokehityksen ehdotuksena valittaisiin yksikkötestitapausten automatisoinnin PHP-ympäristössä. Tehtäessä tätä työtä selvisi, että esimerkiksi NetBeanissa on jo rakennettu mahdollisuus kehittää automaattisia testejä suunnittelemalla ja toteuttamalla automatisoinnin testaus Seleniumin laajennuksen kautta, siksi jatkokehityksessä voisi ryhtyä juuri tähän kokoonpanoon.

4.4 Oman kokemuksen ehdotukset

Tätä opinnäytetyötä tehtäessä tekijä sai monipuolisen kokemuksen testausprosessien kehittämisestä ja toteuttamisesta. Hän tutkii paljon aineistoa, joka on kohdistettu tähän alueeseen ja sovelsin tätä kokemusta valittuun testausympäristöön. Hän sai lisää seurattavuutta ja hallittavuutta yksikkötason testausprosessiin. Lisäksi hänen kokemukselle lisäsi projektin dokumenttien hallinta. Dokumenttihallinnan kokemus myös hankitettiin testaussuunnitelman tekosta. Testausprosessin motivaation kiinnittymiseen auttoi verkkopelin johdon tietoisuudenlisäämisen testauksen tärkeydestä sekä perusteleminen yksikkötestauksen tarpeellisuutta. Raportissa tutkija kiinnytti huomionsa virheiden havaitsemiseen ja hallintaan. Tekijälle oli apuna testausprosessien mallin käytön yhteydessä suurin piirtein Pressmanin kirja (Pressman 2005) sekä seminaarityön paperit. Testitapauksien suunnittelun ja teon kokemukset hankittiin testausympäristöstä. Yleensä katsoen tekijälle tutkimus auttoi ymmärtämään testausprosessien monimutkaisuutta ja lähestyi niiden merkitystä ohjelmistokehityksen elinkaarissa.

Lähteet

Apache frends. 2010. XAMPP Homepage. Luettavissa: http://www.apachefriends.org/en/xampp.html. Luettu: 17.10.2010.

Bergmann, S. 2010. PHPUnit Homepage. Luettavissa: http://www.phpunit.de. Luettu: 17.10.2010.

FAA. 2003. Federal aviation administration. Software approval guidelines. Luettavissa: http://www.airweb.faa.gov/Regulatory_and_Guidance_Library/rgOrders.nsf/0/640711b7b75dd3d486256d3c006f034f/$FILE/Order8110.49.pdf. Luettu: 18.10.2010.

Haikala, I. & Märijärvi, J. 2004. Ohjelmistotuotanto. 10. painos. Talentum. Helsinki.

Hutcheson, M. 2003. Software testing fundamentals: methods and metrics. Wiley. Indianapolis.

Kautto, T. 1996. Ohjelmistotestaus ja siinä käytettävät työkalut. Luettavissa: http://www.mit.jyu.fi/opiskelu/seminaarit/ohjelmistotekniikka/testaus. Luettu: 18.10.2010

Lehtimäki, T. 2006. Ohjelmistoprojektit käytännössä. Gummerus Kirjapaino Oy. Jyväskylä.

McCabe, T. 1976. A software complexity measure. IEEE Trans. Software engineering. vol. SE-2. December 1976, pp. 308-320.

OAMK. 2005. Software business competence, Ohjelmistotestaus. Luettavissa: http://www.oamk.fi/sbc/testaus/index.htm. Luettu: 18.10.2010.

Oracle Corporation. 2010. NetBeans IDE 6.9 Features. Luettavissa: http://netbeans.org/features/php/index.html. Luettu: 17.10.2010.

Pichler, M. 2010. PHP Depend Downloads. Luettavissa: http://pdepend.org/download.html. Luettu: 17.10.2010.

Pressman, R. 2005. Software engineering: a practitioner's approach. 6th edition. McGraw-Hill. Singapore.

Propel. 2007. Changeset 610, Use bindValue instead of bindParam. Luettavissa: http://www.propelorm.org/changeset/610. Luettu: 18.10.2010.

Rethans, R. 2010a. XDEBUG extension for php. Homepage. Luettavissa: http://www.xdebug.org/index.php. Luettu: 17.10.2010.

Rethans, D. 2010b. Tailored Installation Instructions. Luettavissa: http://xdebug.org/find-binary.php. Luettu: 18.10.2010.

Ross, K. 1998. Practical Guide to Software System Testing. Associates Pty. Ltd. Applecross.

The PHP Group. 2010. PEAR - PHP Extension and Application Repository. Homepage. Luettavissa: http://pear.php.net. Luettu: 17.10.2010.

UTU. 2007. Turun yliopisto. Tuotteen laadunvarmistus: testaus. Luettavissa: http://staff.cs.utu.fi/kurssit/ohjelmistotuotanto/06/Laadunvarmistus_testaus.pdf. Luettu: 18.10.2010.

Wikipedia. 2010. NetBeans. Luettavissa: http://en.wikipedia.org/wiki/NetBeans. Luettu: 18.10.2010.

Liitteet

PROJEKTIN LOPPURAPORTTI

PHP-pohjaisen sovelluksen testaus

Viktor Marttinen

24.10.2010

SISÄLTÖ

1 Tausta 1
2 Saavutetut tulokset 1
3 Työn eteneminen 2
4 Kustannukset 2
5 Resurssien käyttö 2
6 Kokemukset 3
7 Ehdotus jatkotoimenpiteiksi 3
8 Suositukset toimintatapojen muuttamiseksi 3
Liite 1 Ajoitussuunnitelma
Liite 2 Toteumaseuranta

1 Tausta

Tämä tutkimus on tehty tradenomin tutkinnon opinnäytetyön osana. Kurssi on pakollinen kaikille HAAGA-HELIA ammattikorkeakoulun tietojenkäsittelyn koulutusohjelman opiskelijoille. Tavoitteena oli soveltaa tietoja ja taitoja, mitä opittu opiskelun aikana. PHPn sovelluksien testaus ei ole olla mitään opintoja. Opinnoissa oli ohjelmistojen testauksen kurssi (Software QA and Testing - ITP4TF499-2), mistä minä sain motivaatiota kehittä tätä työtä. Lisäksi monet oppiaineet opintoaineet laajensivat minun tietoa ohjelmistotestauksesta. IT-seminaari valmisti minut tekemään tieteellistä kirjoittamista ja tutkimusta. Opinnäytetyön seminaarissa esiteltiin opinnäytetyöprosessi ja erilaisia opinnäytetyön tyyppiä. Noin yhden vuoden kokemus web-sovelluksen ohjelmoinnissa auttoi minulle erittäin paljon. Ammattikorkeakoulun kurssit olivat hyvänä perustana rakentamaan taitoa. Opinnäytetyö oli siltana ammattimaisuuteen. Opinnäytetyön laatiminen oli integroitu työhön. Opinnäytetyön tutkimustyypiksi valittiin empiiriseen tutkimukseen suuntautunut hanke. Kaikki tehtävät tehtiin tämän projektisuunnitelman mukaan.

2 Saavutetut tulokset

Tutkimuksen tavoitteet kuvattiin projektisuunnitelmassa. Huolellinen suunnittelu opinnäytetyön toteuttamisen aikana, aihe ehdotuksen aihe ja projektisuunnitelma antoivat minulle mahdollisuuden toteuttaa tämän työ ajoissa ja saada myönteisiä tuloksia. Soveltamisala pysyi lähes samana aihe ehdotuksen jälkeen koko opinnäytetyön prosessin läpi niin, että alusta alkaen pystyin keskittymään oikeisiin asioihin. Hanke tuottaa odotetut tulokset ajoissa, mikä on hyvä saavutus itseään. Tehtävä tuntui suurelta mutta pikkuhiljaa työ eteni. Tärkeinä oppimisen kannalta tavoitteena oli paneutua yksikkötestaukseen ja PHPUnitiin. Kirjallisuuden katsaus tuottaa tarvittava teoreettisen taustaa aloittamaan testausta testausympäristössä. Empiirinen osa tuotettiin ohjeet työkalujen asentamiseen ja käyttämiseen ja yksikkötason testitapausten laatimiseen. Tehtyjen testitapausten analyysi ja arviointi perustuu näihin tuloksiin. Tulokset teoreettista ja empiiristä osuudesta laitettiin työn testaussuunnitelmaan.

3 Työn eteneminen

Opinnäytetyön laatiminen aloitin toukokuussa 2010, jolloin esitin aihe-ehdotuksen opinnäytetyön ohjaajalle. Lisäksi kesti hyvin kauan saada aihe hyväksyttäväksi. Minun tavoitteeni oli valmistautua tämän vuoden joulukuussa. Kun minä laadin projektin aikataulua, en ollut varmaa kuinka kauan kukin projektin tehtävä tai vaihe kestäisi. Ensiksi minä arvioin tulevan kesän lomapäiviä, sitten määritin tuntien määrä kunkin tehtävään, ja lopussa minä sain keskimäärän työtuntien määrän päivässä. Arviointien jälkeen selvennettiin saamaan projektin valmiiksi ajoissa (viimeinen vuonna 2010 arviointi kokous tapahtuu marraskuussa), tarvitaan noin 5,5 tuntia päivässä Siksi päätin sopia työantajan kanssa vähentää työtunnit puoleen opinnäytetyön teon ajalle. Kiitos johtajalle, että hän antautui tämän mahdollisuuden. Sen jälkeen projekti oli jaoteltava tehtäviin.

Valittuna tutkimusmenetelmänä oli kirjallisuuskatsaus ja empiirisen tutkimus. Kirjallisuuskatsaus alkoi heti aloituskokouksen jälkeen. Osa aineistoa oli kerätty etukäteen. Työnantajan projektista oleva dokumentaatio osoittautui hyödylliseksi ja auttoi paljon opinnäytetyön empiirisen osuuden laadinnassa.

4 Kustannukset

Projektissa ei ollut mitään kustannuksia paitsi käytettyä aikaa.

5 Resurssien käyttö

Kaikki projektin budjetissa määritetty tunnit käytettiin kokonaan. Liitteessä 1 jossa esitetään koko hankkeen aikataulu ja kaikki todelliset ja budjetoidut tunnit. Liitteessä 2 on työajan käyttö kutakin viikkoa kohti. Liitteistä ilmenee että projektin tunnit ylittyivät hieman. Luultavasti väärin ajankäyttö aiheutti tämän ylijäämän. Opinnäytetyön raportinkirjoittamiseen ajankäyttö lähes kolminkertaistui. Viimeinen opinnäytetyön palautusajaksi määritettiin loppukokous, vaikka väliversiot toimitettiin jo kaksi viikkoa ennen.

6 Kokemukset

Projektin teosta opin sain paljon uutta kuten NetBeansin, PHPUnitin ja PDependin käytön. Löysin uuden tavan testata. Minä pystyn ymmärtämään testausprosessien sekä niiden tasojen ja menetelmien tarkoituksen ja voisin ehkä kokeilla itse aloittelevana testaajana jossain yrityksessä. Löysin uuden kehitysympäristön NetBeansin. Työstä saatuja kokemuksia ja johtopäätöksiä sovellettiin työpaikassani, missä usein käytän NetBeans IDE:tä. Se antaa paljon etuja verrattuna muihin kehittymisympäristöihin ja etenkin koodin korjaukseen (debugging) ja refaktoroinnin. Tuttujen asioiden ja työkalujen käyttö on myös parantunut, esimerkiksi PHP-kielen tuntemus ja ohjelmointi sekä oliopohjaisen lähestymistavan periaatteet. Dokumenttien hallinnan periaatteet ovat kehittyneet. Opinnäytetyön aikana laadin testaussuunnitelma kohdeyritykselle sekä muut projektin koskeva asiakirjat. Opinnäytetyön tehtäessä sain arvokasta kokemusta suomen oppimisessa. Minä pystyn tunnistamaan suomen tason eroa, erityisesti kieliopin kannalta.

7 Ehdotus jatkotoimenpiteiksi

Projektin tavoitteena oli tutkia web-sovelluksin testausta PHPn ympäristössä. Tämän työn tavoite on täytetty. Case mallina valittiin eräs verkkopeli. Jatkotoimenpiteeksi ehdotan perehtymään opinnäytetyön avulla muihin verkkopelin osien yksikkötason testaukseen. Tähän voi auttaa tässä projektissa laadittu testaussuunnitelma. Lisäksi testaussuunnitelmaan voisi täydentää järjestelmätestauksen osuudella. Tässä projektissa saavuttu tulos on myös soveltavissa muihin PHP-pohjaisen sovelluksiin.

8 Suositukset toimintatapojen muuttamiseksi

Suosittelen opinnäytetyön aloittamista teko mahdollisimman pian, mahdollisimman jo ennen aihe-ehdotusta ja suunnitella sitä huolellisesti etukäteen. Nykyinen menettely, milloin opiskelija on odotettava aiheen hyväksymistä, ei ole käytännöllistä. Minun tapauksessa tämä vaihe kesti noin kuukauden, jona aikana minä olisin voinut tehdä jo opinnäy-

tetyötä. Toinen asia, mihin haluaisin kiinnittää huomiota, on kokouksien määrä. Kolme projektiryhmän kokousta ei ole riitä. Myös tulevan työn suunnitellussa, suositellen opinnäyteraportin kirjoittamista laittaa koko projektin ajan. Ja viimeiseksi suositellaan varamaan aikaa työn täydentämiseen ja virheen korjaamiseen enemmän, etenkin aikataulun lopussa.

LIITTEET

Liite 1 Ajoitussuunnitelma

punaiset ruudut paikantavat ohjauskokouksia ko. viikolle

turkoosit ruudut tarkoittavat, että työtä tehdään ko. viikolla

harmaat ruudut kuvaavat lomajaksoa

keltaiset kokoavat työvaiheen viikkorivillä

vaalea sininen ja valkoinen vuorottelevat eri kuukausien viikkoja

teht.no.	tehtävä	lopputulos	aloituskriteeri	toteutettu	suunniteltu	22	23	24	25	26	27-31	32	33	34	35	36	37	38	39	40	41	42	43	44
1	Projektin hallinnointi	hyväksytty opinnäytetyö	projektisuunnitelma	57	62																			
1.1	Projekti suunnitelman laatiminen	hyväksytty projektisuunnitelma	hyväksytty aihe-ehdotus	15	12	12																		
1.2	Aloituskokouksen valmistelu	esityslista ja materiaali jaettu	projektisuunnitelma	2	2	2																		
1.3	Aloituskokous	käynnistetty projekti	projektisuunnitelma	2	2	2																		
1.4	Pöytäkirjan laatiminen	pöytäkirja jaettu	kokous on pidetty	3	2	1	1																	
2.1	Edistymisen raportti laatiminen	edistymisraportti	seurantajakso umpeutuu	12	12									6	6									
2.2	ohjauskokouksen valmistelu	esityslista ja materiaali jaettu	testaussuunnitelma	1	2										2									
2.3	Ohjausryhmän kokous	esityslista, materiaali jaettu	sovittu kokousaika	2	2										2									
2.4	Pöytäkirjan laatiminen	pöytäkirjat	kokous on pidetty	2	2										1	1								
3.1	Loppuraportin laatiminen	Projektin loppuraportti	kaikki verkkopelin testaus vaiheet on pidetty	12	20																		10	10
3.2	Päättökokouksen valmistelu	Esityslista, materiaali jaettu	Projektin loppuraportti	2	2																			2
3.3	Päättökokous	Päätetty projekti	Projektin loppuraportti	2	2																			2
3.4	Pöytäkirjan laatiminen	Pöytäkirja	kokous on pidetty	2	2																			2
2	Verkkopelin testaus	hallittu edistyminen	projekti on käynnistetty	401	340																			
1.5	Teoreettisen aineiston keräämi-nen ja kirjoittaminen	opinnäytetyön teoreetti-nen osuus	projektisuunnitelman hyväk-syminen	146	140		15	20	15	20		30	25	15										
1.6	Testauksen suunnittelu ja suunnitelman laatiminen	testaussuunnitelma	teoreettinen osuus	40	50							10	15	15	10									
2.5	Testausaineiston laatiminen	testausaineisto	testaussuunnitelma on hyväk-sytty	10	10										5	5								
2.6	Testauksen toteuttaminen	testauksen tulos	testausaineisto ja suunnitelma	55	90											10	30	30	10	10				

3.1	Opintäytetyönteko	hyväksytty opinnäytetyö	kaikki edellyttävät vaiheet on pidetty	150	60														5	5	10	15	15	10	
			Tunnit yhteensä:	471	400	17	16	20	15	20		40	40	36	26	16	30	35	15	20	15	15	20	16	

Liite 2 Toteumaseuranta

Projektin hallinnointi		
aloitus viikon no.	tehtävä	tunnit
22	Projekti suunnitelman laatiminen	15
22	Aloituskokouksen valmistelu	2
22	Aloituskokous	2
22	Pöytäkirjan laatiminen	3
22	Edistymisen raportti laatiminen	12
23	Teoreettisen aineiston kerääminen ja kirjoittaminen	146
32	Testauksen suunnittelu ja suunnitelman laatiminen	40
35	Testausaineiston laatiminen	10
35	ohjauskokouksen valmistelu	2
35	Ohjausryhmän kokous	2
35	Pöytäkirjan laatiminen	2
34	Edistymisraportin laatiminen	12
38	Testauksien toteuttaminen, testitapaus 1	30
39	Testauksien toteuttaminen, testitapaus 2	15
40	Testauksien toteuttaminen, testitapaus 2	10
38	Opinnäytetyönteko, johdanto	15
39	Opinnäytetyönteko, viitekehys	50
40	Opinnäytetyönteko, empiirinen osuuden teko	50
42	Opinnäytetyönteko, abstraktit, liitteet, lähdeluettelo, määritelmät	15
42	Opinnäytetyönteko, työn täydennys	20

	ja korjaaminen	
43	Loppuraportin laatiminen	12
43	Päättökokouksen valmistelu	2
43	Päättökokous	2
43	Pöytäkirjan laatiminen	2
Yhteensä:		471