LAUREA AMMATTIKORKEAKOULU Uuden edellä

PHP-yksikkötestauksen käyttöönotto yritykseen

Könönen, Ari

2012 Laurea Leppävaara

Laurea-ammattikorkeakoulu
Laurea Leppävaara

PHP-yksikkötestauksen käyttöönotto yritykseen

Könönen, Ari
Tietojenkäsittelyn koulutusohjelma
Opinnäytetyö
Lokakuu, 2012

Laurea-ammattikorkeakoulu
Laurea Leppävaara
Tietojenkäsittelyn koulutusohjelma

Tiivistelmä

Könönen, Ari

PHP-yksikkötestauksen käyttöönotto yritykseen

Vuosi 2012 Sivumäärä 41

Toiminnallisen opinnäytetyöni aiheena oli ottaa käyttöön PHP-yksikkötestaus yrityksessä. Yrityksessä ei aikaisemmin ollut käytössä ohjelmallista testausta, vaan testaaminen tapahtui käsin. Tavoitteenani oli kuunnella yrityksen henkilökunnan toiveita ja toteuttaa juuri heidän tarpeisiinsa sopiva ohjelmallinen testaus. Lisäksi PHP-yksikkötestauksen tarkoituksena oli nopeuttaa ja helpottaa yrityksen ohjelmoijien työtä käyttöönoton jälkeen.

Projekti rakentuu teoreettisesta osuudesta ja käytännön työstä. Kirjallisen tutkielman teoreettisessa osuudessa käsiteltiin ohjelmallista testaamista, testauksen vaihejakoa sekä omassa luvussaan syvällisemmin yksikkötestauksesta. Teoriaosuuden jälkeen tarkasteltiin tarkemmin yksikkötestausta käytännössä ja havainnollistettiin testaamista yksikkötestausien luonnin, testauksen ja käyttämisen kautta. Käytännön työn tein yrityksessä kokopäivätyöni ohella vuoden 2012 aikana.

Opinnäytetyön teoriaosuuden aineisto kerättiin tasapuolisesti sekä kirjallisia että internetlähteitä hyödyntäen. Suomenkielisten lähteiden lisäksi osa oli englanninkielisiä. Käytännön työhön sain ideoita ja toiveita kollegoiltani, mutta myös omakohtainen kokemus ohjelmoinnista sekä testaamisesta hyödynnettiin.

Opinnäytetyön lopussa olen kuvannut parannusprojektin vaiheet yksityiskohtaisemmin ja kuvannut, mitä hyötyä yritykselle on ollut PHP yksikkötestauksen käyttöönotosta. Ohjelmallisen testaamisen käyttöönoton jälkeen yrityksen johto ja ohjelmoijat kommentoivat suunnittelun ja testaamisen nopeutuneen verrattaessa käsintestaukseen. Yrityksen palautteen mukaan ohjelmallinen testaus on säästänyt yrityksen varoja.

Asiasanat Testaus, Yksikkö testaus

Laurea University of Applied Sciences
Laurea Leppävaara
Bachelors's Degree Programme in Business Information Technology

Abstract

Könönen, Ari

PHP Unit testing commissioning for a company

Year 2012 Pages 41

Functional thesis topic was PHP Unit testing commissioning for a company. Company hasn't previously used program powered testing, but instead testing was made by hand. My goal was to listen to the employees of the company and take their wishes and implement them for a software testing. In addition, PHP Unit testing's purpose was to speed up and facilitate the company's programmers' work after the commissioning.

The project builds from the theoretical part and the practical work. The theoretical part of the written thesis includes program powered testing, the testing phases and a separate chapter about Unit Testing. After the theoretical part thesis goes in to more practical part about unit testing practice and illustrated by how the unit testing are created, tested, and used. I did the practical work while I had full-time job in the year 2012.

The theoretical material was collected with a balance of written material and internet sources. Sources used in thesis where in both Finnish and English language. Practical part of the project I got ideas and wishes from my colleagues, but also my personal experience of programming and testing were utilized.

At the end of the thesis I have described the improvement phases of the project. I describe in a more detailed and described way about the benefits that the company has gained from the commissioning of PHP Unit Testing. After commissioning program powered testing for the company, company's management and programmers comment that the design and testing process has been sped up compared to testing by hand. Company has been giving feedback about how program powered testing has saved some of the company's money.

Keywords Testing, Unit testing,

Sisällys

1. Johdanto

Toiminnallisen opinnäytetyöni aiheeksi valitsin PHP-yksikkötestauksen käyttöönoton Yritykseen X. Kiinnostuin kyseisestä aiheesta sen vuoksi, ettei yrityksessä ollut käytössä mitään ohjelmaa testaukseen, vaan kaikki testaus tehtiin käsin. Manuaalisessa testauksessa toki on puolensa - käsin tehtynä testaus on parhaimmillaan nopeaa, mutta toisinaan hyvin hidasta ja aikaa vievää.

Lähtötilanteessa eli vuoden 2012 alussa Yritys X oli ehtinyt tutustua ohjelmalliseen testaukseen vasta vähän. Ehdottaessani heille ideaa ohjelmallisen testauksen käyttöönotosta heidän tarpeisiinsa osana opinnäytetyötäni, vastaanotto oli myönteinen. Tuolloin olin ehtinyt työskennellä Yritys X:n palveluksessa noin vuoden ajan, joten yritys tarjosi mahdollisuuden toteuttaa vaativaa projektia. Opinnäytetyön idean hahmotuttua yritykselle sekä itselleni, osaamiseni ohjelmallisen testauksen osalta ei vielä ollut täydellistä. Olin toki siihen tutustunut ja testausta jonkin verran tehnytkin, mutta juuri uuden oppimisen mahdollisuus projektin edetessä innoitti minua suuresti. Tavoitteenani on oppia mahdollisimman kattavasti ohjelmallisesta testauksesta, sillä se kasvattaa ammatillista osaamistani ja toivon tulevaisuudessa toimivani ohjelmoinnin parissa. Konkreettisen työn tekeminen on itselleni paras oppimisen kanava, sillä olen kinesteettinen oppija. Toisin sanoen itse tekeminen saa itselläni aikaan parhaan oppimistuloksen. Lisäksi ohjelmointi on ollut intohimoni jo vuosia- samaan aikaan vaativaa, mutta onnistuessaan erittäin palkitsevaa.

Opinnäytetyöni alussa käydään läpi ensin yritystä, jonka kanssa olen projektin toteuttanut. Käytän yrityksestä nimeä Yritys X, sillä salassapitovelvollisuuden, ja johdon kanssa tekemäni sopimuksen mukaisesti en saa paljastaa yrityksestä mitään, minkä avulla sen voisi tunnistaa. Testausosionkin olen sen vuoksi kirjoittanut esimerkein, en käyttäen varsinaista tekemääni koodia. Yrityskuvauksen jälkeen perehdytän lukijan lähtötilanteeseen ennen projektia, jonka jälkeen tarkastelun alla on aiheepiiriin keskeisesti liittyvää teoriaa. Valitsin opinnäytetyössä käytettäväksi teoreettiseksi viitekehykseksi yksikkötestauksen. Käsittelen edellämainittua käsitettä myöhemmin omassa luvussaan. Lisäksi omissa osioissaan käsitellään myös testausta ylipäätään sekä siihen liittyviä käsitteitä.

Teoriaosuuden jälkeen käsitellään itse käytännöntyötä, sen eri vaiheita, tuloksia sekä kokemiani vastoinkäymisiä sekä niistä selviytymistä. Tämän jälkeen perustelen päätöksiäni ja tekemiäni valintoja. Aivan lopussa on loppupohdinta siitä, miten olen toiminnallisessa opinnäytetyössäni onnistunut, mitä olen oppinut ja kuinka itse kirjoitusprosessi on sujunut.

2. Yritys X

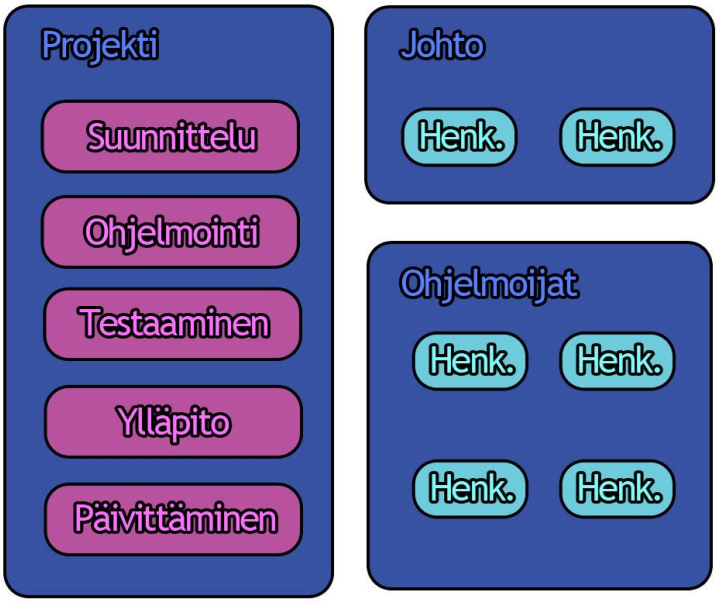
PHP-yksikkötestauksen käyttöönotto toteutettiin yrityksessä nimeltä Yritys X. Yritys haluaa pysyä tuntemattomana ja siksi en voi antaa siitä mitään yksityiskohtaista tietoa. Yrityksessä ei ole aikaisemmin ollut käytössä mitään ohjelmallista testausta, vaan testaus on tehty ohjelmoijien toimesta käsin. Ohjelmointikielenä Yritys X käyttää muun muassa PHP:tä ja tämän vuoksi toteutankin yksikkötestauksen PHP:lle.

Yritys X on pienyritys, jonka henkilöstö koontuu johdon noin kahdesta henkilöstä sekä ohjelmoijista, joita on noin neljä. Organisaation rakenne on koostaan johtuen hyvin yksinkertainen. Sovimme yrityksen johtajan kanssa, että toimin projektin suunnittelijana ja minulla on projektin toteutuksen suhteen varsin vapaat kädet. Toki toivoin yrityksen puolelta toiveita ja ideoita niin johdon kuin ohjelmoijienkin puolelta. Esimieheni toimii yrityksessä toimitusjohtajana, suunnittelijana sekä projektini ajan valvojanani. Vastuu työn etenemisestä sekä huolellisesta toteutuksesta oli minulla itselläni. Virheisiin toki oli varaa, sillä minulla oli tukenani ammattilaisten työyhteisö sekä osaava johtoporras.

2.1. Yrityksen työympäristö

Yrityksen työympäristö muodostuu noin viidestä henkilöstä, joista noin 2 on johtoportaassa ja noin 4 on ohjelmointipuolella. Ohjelmointi jakaantuu yleensä backend- ja frontend-ohjelmointiin, mutta yrityksen suhteellisen pienen koon vuoksi ohjelmoijien täytyy osata sekä backend- että frontend- ohjelmointia. Molempien osa-alueiden osaaminen on hyvin harvinaista ohjelmoijien keskuudessa, sillä yleensä keskitytään vain toiseen.

Alla olevasta kuvasta voi nähdä työympäristön muodostumisen. Työympäristö rakentuu kahdesta erillisestä tekijästä, henkilöstöstä ja projektista.



Kuvio 1: Kuviossa esitetään Yritys X:n työympäristöä

Projektin käsitteeseen sisältyvät seuraavat vaiheet: suunnittelu, ohjelmointi, testaaminen, ylläpito sekä päivittäminen. Suunnitteluvaiheeseen osallistuu henkilöstöstä sekä johto että ohjelmoijat. Tässä kyseisessä vaiheessa suunnitellaan ohjelman osion toteutustapa, käyttöliittymä ja dokumentointi. Ohjelmoinnista yrityksessä vastaavat ainoastaan ohjelmoijat, vaikkakin myös johdolla on riittävä tietotaito tähän osa-alueeseen.

Ohjelmointitapana käytetään Extremeohjelmointia. Yksinomaan ohjelmoijien työnkuvaan kuuluvat myös testaaminen, ylläpito ja päivittäminen. Ohjelmoijien hyvin vapaasta ja itsenäisestä työnkuvasta huolimatta, ohjelmoijat ovat jatkuvassa vastavuoroisessa kanssakäymisessä johtoportaaseen kuuluvien henkilöiden kanssa. Palautekeskusteluja käydään kuukausittain ja ohjelmoijien on annettava näyttöä tekemästään työstä muun muassa dokumentoinnin avulla.

2.2. Hankkeen toimijat ja yhteistyötahot

Hanke toteutettiin yrityksen sisällä eli projektin toimijoihin kuului itseni lisäksi esimieheni sekä välillisesti Yritys X:n muu henkilökunta. Yritys X toteutti projektin itsenäisesti ilman muita yhteistyötahoja.

Projektin valmistuttua tehtävänäni oli perehdyttää yrityksen työntekijät siihen, kuinka testausohjelmointi tehdään. PHP-yksikkötestausta käyttää jatkossa yrityksen jokainen työntekijä eli kyseessä on varsin suuri projekti yritykselle.

2.3. Resurssit ja kustannukset

Seuraavaksi käydään läpi projektissa käytettyjä resursseja sekä mahdollisia kustannuksia. Voisin sanoa olleeni hankkeessa ainoa henkilöresurssi. Koin tämän suurena mahdollisuutena, sillä sain toteuttaa omia ideoitani ja ajatuksiani, tehdä riittävän ajan puitteissa virheitä ja oppia niistä.

Kustannuksia Yritys X:lle ei koitunut muusta kuin palkastani, sillä olin tavallisessa työsuhteessa koko projektin ajan. Toimin yrityksessä ohjelmistosuunnittelijana. Tein siis toiminnallisen opinnäytetyöni käytännön osuutta varsinaisen työni ohella. Kiireisinä päivinä dead linejen lähestyessä palkkatyö meni toki projektin edelle. Käytin projektissa ilmaisia ohjelmia, joten niistä ei koitunut Yritys X:lle erillisiä kustannuksia. Ulkopuolisia rahoittajia ei käytetty, eikä niille ollut tarvettakaan.

2.4. Hankkeen riskit

Hankkeessa suurimpana riskinä oli se, että olin periaatteessa projektin ainoa henkilöresurssi. Väljän aikataulun vuoksi sain onneksi tehdä työtä omaa tahtiani, mutta jos aikataulu olisi ollut hyvin tiukka, sairastumisiin ei olisi ollut varaa. Tämän kaltaisissa hankkeissa tietotekniset asiat ovat toki aina oma riskinsä. Yleisiä tietoteknisiä ongelmia ovat muun muassa tiedostojen tuhoutumiset tai korruptoitumiset. Lisäksi esimerkiksi virukset ja sähkökatkot ovat arkipäivää tietotekniikan alan työpaikoilla.

Yrityksessä on onneksi käytössään tehokkaat ohjelmat ja toimintatavat riskienhallintaan. Projetini oli siis melko riskivapaa. Tietenkin oma terveydentilani oli ainoa, jota ei voinut ennustaa etukäteen tai sairastumisia ennaltaehkäistä.

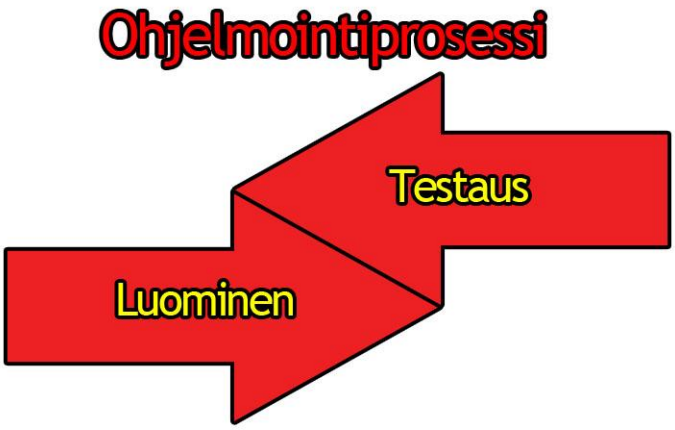
2.5. Projektin dokumentointi

Hoidin yrityksen sisäiset dokumentoinnit itse ja opinnäytetyöni kirjallisen dokumentoinnin myös itsenäisesti. Dokumentoinnissa voin näyttää vain opinnäytetyöni kirjallisen osuuden, koska minulla on salassapitovelvollisuus asioista, mitä yrityksessä teen. Voin ainoastaan läpi käydä, mitä teen ja miksi. Tekemäni testit ovat yrityksen ohjelmientestejä, enkä niitä sen vuoksi voi työssäni esitellä.

Tämän vuoksi olen esitellyt testejä myöhemmin esimerkkien ja periaatteiden avulla omassa luvussaan. Esittelen toki kirjallisessa osuudessa sen, kuinka toteutin testauksen asennuksen ja käyttönoton.

3. Ohjelmointiprosessi yrityksessä

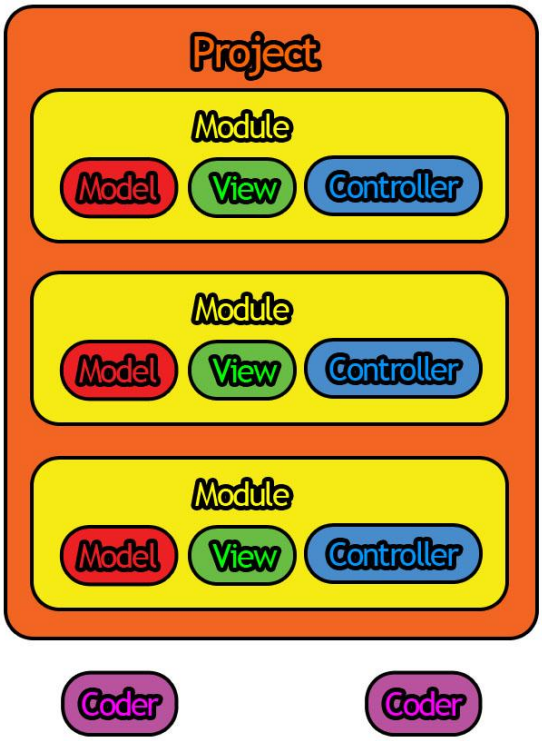
Yritys X:ssä ohjelmointiprosessina käytetään Extremeohjelmointiin mukaista prosessia. Prosessi toimii käytännössä niin, että ensimmäisessä vaiheessa luodaan pieni osa ohjelman osiosta. Tämän jälkeen kyseisen osan toimivuus testataan käytännössä. Ohjelmointiprosessin seuraavat vaiheet mukailevat ensimmäistä vaihetta viimeiseen vaiheeseen saakka. Viimeisessä vaiheessa testataan prosessin edetessä syntyneen kokonaisuuden toimivuus.



Kuvio 2: Kuviossa esitetään ohjelmointiprosessia

3.1. Lähtötilanne: käsintestaus yritys X:ssä

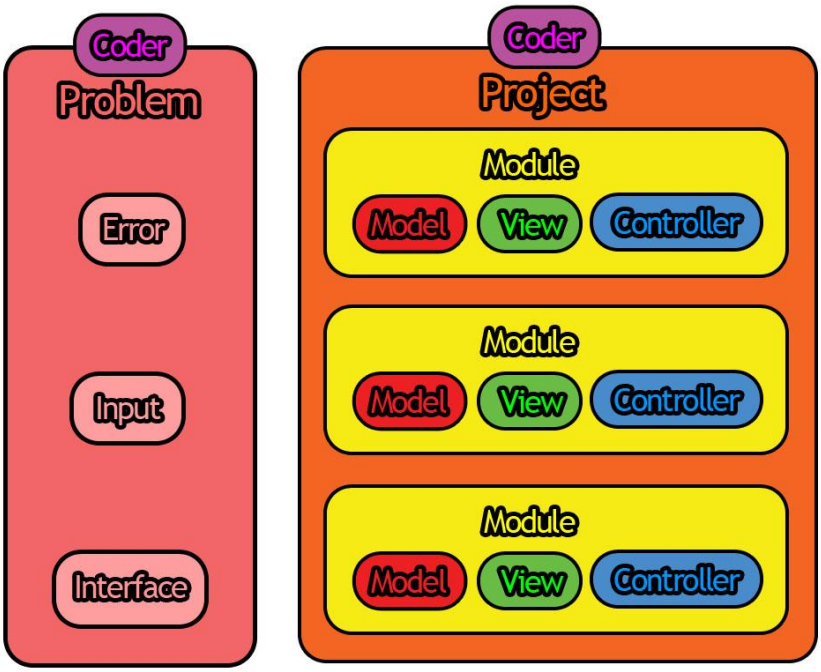
Tässä luvussa esitellään tekemieni kuvien avulla, millainen tilanne valitsemassani yrityksessä on tällä hetkellä testauksen suhteen. Yritys X käyttää testauksessa käsintestausta, eikä ohjelmallista testausta ole aikaisemmin ollut käytössä. Kuvien kieleksi valitsin englannin kielen, sillä testaaminen ja koodaus tapahtuvat englannin kielellä ja koen sen sopivammaksi kuvaamaan projektin lähtötilannetta. Kuvien sisältöä sekä aikaisempaa toimintakäytäntöä avataan kuvien alapuolella olevissa teksteissä.



Kuvio 3: Kuviossa esitetään lähtötilanteen projektia

Toteutusympäristö perustuu projekteihin, jossa yksikkötestausta tullaan käyttämään. Nykyinen järjestelmä valitsemassani pienyrityksessä on ylläolevan kuvan mallinen ja sitä tarkastellaan seuraavaksi.

Projekti koostuu useista osista, joista suurimman osan tuottaa modulet. Modulet koostuvat kolmesta osiosta; Model, View ja Controler, joista käytetään lyhennettä MVC. Ohjelmoijat rakentavat erilaisia moduleita, joista ohjelma koostuu. Modulen ohjelmointi aloitetaan yleensä Controllerista, joka hallitsee modulea. Tämän jälkeen siirrytään View:iin, joka hallitsee modulen ulkonäköasioita. Viimeisenä, muttei vähäisempänä on Model, joka hallitsee datan käsittelyä ja on modulen ydin. (Tony Marston. 2004.)



Kuvio 4: Kuviossa esitetään lähtötilanteen projektia ongelma tilanteessa

Kun nykyisessä projektiympäristössä tapahtuu virhe, ohjelmoija joutuu käymään koko projektin läpi. Yllä olevassa kuvassa näkyy, että ongelmat ovat eri tasoisia ja siksi voivat johtua monesta eri syystä. Ohjelmoija voi saada ongelmatilanteen aikaan useista syistä, mutta ongelmasta riippumatta ne pitää korjata. Yleensä ongelma tilanteen selvittämiseen voi kulua monia tunteja vain siksi, koska ohjelmoijan pitää käydä niin suuri alue läpi käsin. Ohjelman laajuudesta riippuen käsintestaus voi olla erittäin hidasta ja vaativaa. Pienissä ongelmissa käsintestaus on yleensä todella nopeaa, koska ongelman syyt löydetään nopeasti.

3.2. Kehitystarpeiden kuvaus

Ohjelmallinen testaus tuottaa erittäin nopean ja tehokkaan tavan testata ohjelmaa. Tämän hetkinen käsin testaaminen toimii koodia kirjoittaessa, mutta jos ohjelmaan tehdään suurempi muutos, niin tilanne onkin hankalampi. Mittava muutos ohjelmassa saattaa aiheuttaa ongelman, mikä tapahtuu kaikissa sivuissa, missä muutettua osaa käytetään. Ongelmien etsiminen ilman mitään tietoa on hidasta ja välillä erittäin vaikeaa.

Ohjelmallinen testaus vaatii tietysti testausohjelman käyttöön ottamisen. Aluksi pitää valita, minkä ohjelmallisen testausohjelman haluaa asentaa. Valinnan jälkeen ohjelma pitää testata kirjoittamalla testattava testiohjelma.

Ohjelmointi kokee myös kehityksentarpeen ohjelmallisen testauksen jälkeen, koska testausohjelma pystyy testaamaan ainostaan lopputuloksia ilman lisäohjelmointia testattavaan ohjelmaan. Ohjelmoinnin määrän lisäys saattaa kuulostaa huonolta idealta, ohjelmoinnin muutenkin suuren työmäärän vuoksi. Tuskinpa kukaan haluaa pidentää testaamiseen käytettyä aikaa. Vastaus on kuitenkin yksinkertainen: aika, joka kuluu käsin testaukseen on moninkertainen siihen aikaan nähden, mikä kuluu testausohjelman ohjelmointiin. Tämä tarkoitta sitä, että ohjelman osiin pitää ohjelmoida testausta varten säännöt. Näillä säännöillä pystymme katsomaan, että kyseinen kohta tekee juuri sen, mitä kussakin tilanteessa halutaan tapahtuvan.

Kehitystarpeita on paljon kuten haasteitakin, mutta yrityksessä kehitystarpeet nähdään voimavaroina, sillä ne tukevat tulevaisuuden toimintaa sekä tehokasta työskentelyä. Ohjelmallinen testaus tulee olemaan erittäin suuri kehitys yritykseen ja se mahdollistaa nopean testauksen. Ohjelmien ohjelmointiaika tulee lyhenemään tämän ansiosta ja sillä säästetään tulevaisuudessa aikaa ja rahaa.

Automaattinen testaus varmistaa ohjelman halutun toimintatavan ohjelman elinkaaren aikana. Tärkeys huomataan, kun ilman automaattista testausta olevan ohjelman toimintatapa poikkeaa halutusta toimintatavasta. Toimintatavan poikeamisen syyn löytäminen manuaalisesti voi olla erittäin vaikeaa. (Davey Shafik, Lorna Mitchell & Matthew Turland 2011, 243.)

3.3. PHP-yksikkötestauksen valinta yritykselle

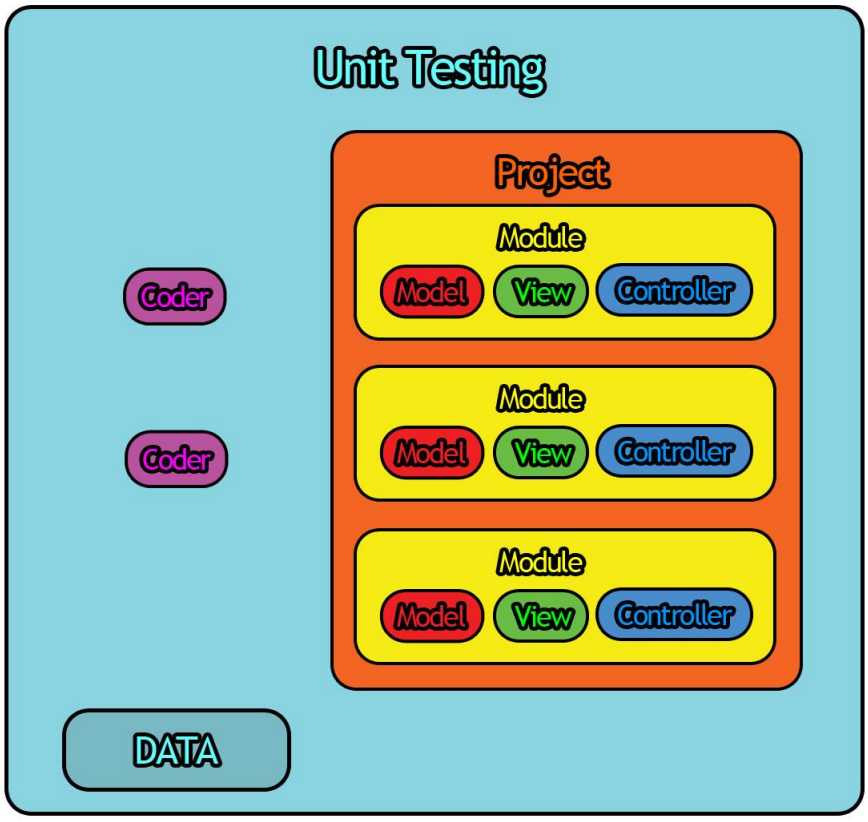
Näin valitsemani aiheen PHP -yksikkötestauksesta tarpeelliseksi kyseiselle yritykselle siksi, koska heillä ei ollut käytössään ohjelmallista testausta. Lisäksi koin ohjelmallisen testauksen nopeuttavan ja helpottavan yrityksessä työskentelevien ohjelmoijien työntekoa.

Ohjelmallisen testauksen ansiosta ohjelmoijien ei tarvitsisi enää käyttää pitkiä aikoja ongelmien selvittämiseen, kun automaattinen testaus suorittaisi sen heidän puolestaan. Aikaisemmin suuren ongelman ilmetessä saattoi käydä niin, että kaikkien ohjelmoijien työnteko keskeytyi vian vuoksi. Muutosten jälkeen ohjelma ei välttämättä toiminut enää oikein ja siitä seurasi väistämättä työntekijöiden toimettomuutta ja odottamista. Edellä mainitun kaltainen tilanne käy toistuessaan yritykselle kalliiksi. Ohjelmallinen testaus puolestaan minimoi tällaiset ongelmatilanteet ja sen hyvinä puolina voidaan nähdä juuri yrityksen varojen säästö, ajan tehokas käyttäminen sekä kaikkien positiivisen asenteen seilyminen. Kehittämistoiminta oli tarpeen, koska yritykseltä puuttui kokonaan ohjelmallinen testaus.

Ohjelmoinnissa testausta pidetään kaikkein ylimmän tason (master level) asiana ja siksi koin projektin tilaisuutena näyttää omaa kehitystäni ohjelmoinnissa. Työ olisi haastava ja samalla myös jännittävä, sillä perusohjelmoinnin lisäksi testausohjelmaa pitää ohjelmoida testaamaan ohjelmointia. Olen nähnyt ohjelmallisen testauksen käyttöönoton alusta alkaen mahdollisuutena kehittyä ja oppia uutta. Toki vakinaisen kokopäivätyön ohella projektissa oli omat haasteensa ja välillä vapaa-ajalle ei juurikaan aikaa jäänyt.

3.4. Tavoitetilanne: Ohjelmallinen testaus Yritys X:ssä

Tehtävänä on luoda ohjelmallinen testaus, jota yritys voi hyödyntää nykyisissä ja tulevissa projekteissa. Tavoitteena on luoda yritykselle tapa testata ohjelmia nopeasti ja tehokkaasti. Ohjelmallinen testaus on tällä hetkellä tehokkain tapa tehdä testausta, koska virheiden ilmetessä, syyn löytäminen on nopeaa.

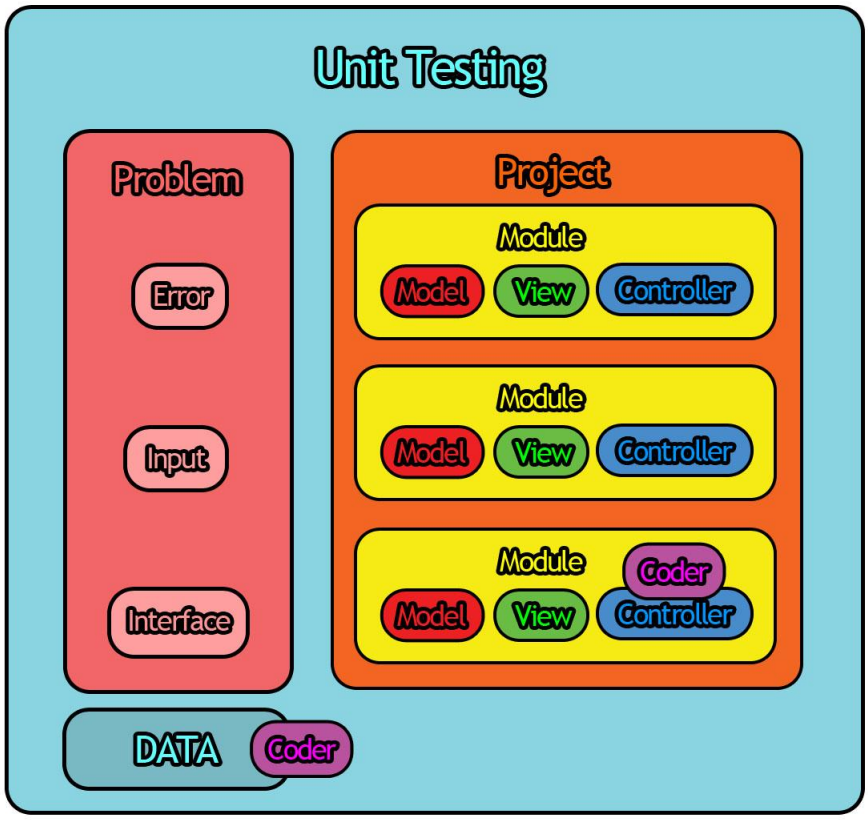


Kuvio 5: Kuviossa esitetään tavoite projektia

Yksikkötestauksen käyttöönotto saavuttaa tietynlaisen turvan projektin ympärille. Turvalla tarkoitan jokaisen osa-alueen toimivuuden varmentamisen yksikkötesteillä. Yllä olevasta kuvasta voi nähdä, että yksikkötestaus on laaja-alainen ja on osa koko projektia.

Yksikkötestaus on vaativa rakentaa, mutta sen todellinen hyöty huomataan ongelmien saapuessa.

Yksikkötestaus vaikuttaa työympäristöön muutamalla tavalla, joista huomattavimmat tekijät käsitellään seuraavaksi. Uuden modulen teko saa uuden vaiheen, joka perustuu Extremeohjelmoinnin ajattelutapaan. Työympäristö perustuu Extremeohjelmointiin ja siksi käyttöönotossa otettiin tämän huomioon. Modulen tekeminen alkaa Controllerista normaalisti, mutta yksikkötestauksen kanssa se alkaakin Controllerin yksikkötestistä. Testi suunnitellaan ja rakennetaan controllerin halutun käyttämisen ja tulosten perusteella. Tämän jälkeen controllerin ohjelmointi voi alkaa. Seuraavat vaiheet menevät tuttuun tapaan. Ensin View ja tämän jälkeen Model, mutta ainoana erona niillekin ohjelmoidaan yksikkötestaus aluksi.



Kuvio 6: Kuviossa esitetään tavoite projektia ongelma tilanteessa

Yllä olevassa kuvassa esitellään ongelman paikantamisen tehokkuus. Yksikkötestauksen käyttö on tehokasta ja nopeaa. Ongelman sattuessa täytyy vain ajaa yksikkötestit ja katsoa yksikkötestistä saatu data/logi, joka tulostaa suoraan miten ja missä ongelma syntyy. Kuvan esimerkissä ongelma on tapahtunut Kolmannen modulen Controllerissa, jonka paikantaminen löytyi nopeasti yksikkötestauksen tulosten avulla.

3.5. Tutkimusmenetelmä

Valitsin tutkimusmenetelmäkseni opinnäytetyössäni kokemuksellisen tutkimusmenetelmän, koska olen mukana suunnittelusta käyttöönottoon sekä käyttämiseen saakka. Työni perustuu kokemuksen ja tekemisen kautta havaittuun ongelmaan. Mielenkiinnosta odotan, mitä muutoksia ohjalmallinen testaus tekee yrityksen aikasemmin käytössä olleille toimintatavoille.

Työn valmistuttua, sitä tullaan kokeilemaan käytännössä, jolloin saadaan itseni lisäksi myös esimieheni sekä kollegoideni palautte ohjelmallisen testauksen käyttämisestä.

4. Ohjelmistojen testaaminen

Tässä luvussa esitellään ohjelmistojen testaamista, sen historiaa sekä testaamisen vaihejakoa. Tämän jälkeen käsitellään perusteellisemmin yksikkötestausta.

Testaukselle löytyy useampia määritelmiä, mutta yksinkertaisesti sanottuna testaus on ohjelman suorittamista virheiden löytämiseksi (Myers, G. 2004, 5-6). Testaus rakentuu neljästä eri työvaiheesta. Näitä vaiheita ovat testauksen suunnittelu, testiympäristön luonti, testin suorittaminen sekä tulosten tarkastelu. Testauksen suunnitteluun voidaan sisällyttää testaussuunnitelma ja testitapaukset. (Haikala, I., Märijärvi, J. 2004, 283.)

4.1. Testaamisen historiaa

Testaamisen alkuaikoina suunnitelmat kirjattiin paperille ja testausprosessi tapahtui myös kirjallisena. Testaaminen toteutettiin usein vasta projektin loppuvaiheessa, eikä erillisiä testaajia ollut, vaan tehtävän suoritti se, kuka ehti. Henkilökohtaisten työasemien käyttöönotto muutti kuitenkin merkittävästi testauksen kehitystä ja voidaankin sanoa, että tämä aloitti testauksessa aivan uuden ajan. Enää testaus ei ollut ainoastaan lopputoimenpide vaan testauksen suunnitteluun ja toteutukseen alettiin käyttää enemmän aikaa. (Dustin, E., Rashka, J., Paul, J. 1999, 5-7.) Tällä vuosituhannella kiinnostus on siirtynyt automaattiseen testaukseen erilaisten apuvälineiden kehittymisen seurauksena.

4.2. Testauksen vaihejako

Testauksen vaihejaon eli testauksen V-mallin mukaan testaustasoja on kolme eli, järjestelmätestaus, integrointitestaus ja moduulitestaus. Moduulitestauksesta käytetään usein myös nimitystä yksikkötestaus. Toisinaan järjestelmätestausta saattaa seurata kenttätestaus tai hyväksymistestaus tai vaihtoehtoisesti molemmat. (Haikala, I., Märijärvi, J. 2004, 288.)

4.1.2 Integrointitestaus

Integrointitestauksessa (integration testing) pienistä palasista (moduulit tai moduuliryhmät) pyritään rakentamaan suurempia kokonaisuuksia sovelluksen osia yhdistelemällä. Tässä testaustyypissä keskeistä on tutkia moduulien rajapintojen toimivuutta (Testaussuunnitelma 2004). Integrointi- ja moduulitestaus liittyvät kiinteästi toisiinsa, sillä integrointitestauksen eteneminen tapahtuu usein vieri vieressä moduulitestauksen kanssa. Integroinnin etenemissuunta on useimmiten ylöspäin suuntautuva eli kokoava (bottom-up). Ylhäältä alaspäin suuntautuvaa integrointia kutsutaan puolestaan jäsentäväksi integroinniksi (top-down). (Haikala, I., Märijärvi, J. 2004, 290.) Menetelmänä voi lisäksi olla kertarysäys, jolloin moduulit liitetään yhdeksi kokonaisuudeksi yhdellä kertaa.

4.2.2 Järjestelmätestaus

Järjestelmätestaukseen sisältyy järjestelmän testaaminen kokonaisuudessaan sekä tulosten vertaaminen itse määrittelydokumentaatioon. Tähän testaustyyppiin lukeutuvat myös niin kutsutut ei toiminnalliset osuudet eli kuormitustestit, käytettävyystestit, luotettavuustestit ja asennustestit. (Haikala, I., Märijärvi, J. 2004, 290-291.)

Järjestelmätestauksen yhteydessä puhutaan monesti käytettävyystestauksesta. Käytettävyystestaus tarkoittaa nimensä mukaisesti pääosin käyttöliittymän testaamista. Testaamista varten tulevista käyttäjistä otetaan pieni otos ja heidän kykyään selvitä erilaisista tehtävistä tarkkaillaan koeolosuhteissa esimerkiksi erillisessä käytettävyyslaboratoriossa. Vaihtoehtoisesti ohjelmistojen käytettävyyden arvioimiseksi voidaan käyttää myös tähän tarkoitukseen erikoistuneita henkilöitä. (Haikala, I., Märijärvi, J. 2004, 291.)

4.3.2 Moduulitestaus

Moduulitestauksessa (module testing) nimensä mukaisesti testataan yksittäisiä moduuleja. Nämä moduulit rakentuvat yleensä noin 100-1000 ohjelmarivistä ja testauksen suorittajina toimivat usein moduulin tekijät yksin tai parin kanssa yhteistyössä.

Testipetien (testbed) avulla voidaan kokeilla moduulien toimivuutta. Testipetiin kuuluu monesti testiajureita ja tynkämoduuleita. Ajureiden avulla moduulin toteuttamien palveluiden kutsuminen sekä tulosten tarkastelu on mahdollista. Tynkämoduulit taasen voivat korvata testattavan moduulin sellaiset osat, joita ei ole olemassa. (Haikala, I., Märijärvi, J. 2004, 289.)

Moduulitestauksessa tavoitteena on löytää virheet ja korjata ne mahdollisimman aikaisessa vaiheessa. Moduulitestaus voidaankin nähdä pienten ohjelmapalasten testaamisena, jossa edetään eteenpäin vasta kun edellinen vaihe toimii ja palaset loksahtavat kohdilleen. (The Software Testing & Quality Engineering Magazine. 2002.)

5. Yksikkötestaus

Yksikkötestaus on moduulitestausta ja käyn yksikkötestauksen tarkemmin läpi omassa luvussaan. Ohjelmoinnissa yksikkötestaus on tapa testata yksittäisiä kohtia ohjelmasta. Kohdat ohjelmasta voivat koostua yhdestä tai useammasta moduulista. Yksikkötestaus näyttää, onko testatut ohjelman osiot käyttövalmiita. Yksikkötestaus palauttaa tulokset tuloksista, jotka voivat olla haluttuja tuloksia tai ei-haluttuja virhetilanteita. Haluttuihin tuloksiin lukeutuu kaikki ne tulokset, joita ohjelman osiolta halutaan. Esimerkiksi ohjelman osio, joka laskee plus laskuja sen tulokseksi halutaan vain numeroita. (Zandstra M. 2008, 391-392.)

Virhetilanteet ovat myös hyvä tutkia, koska ohjelman osio voi saada väärää dataa sisään. Virhetilanne voisi olla plus lasku osiossa, kun numeroiden sijaan lähetetään kirjaimia. Virhetilanteen halutaan olla haitaton, joten yksikkötestauksessa katsotaan mahdollisimman harmiton lopputulema. Olio-pohjaisessa ohjelmoinnissa testejä pystyy tekemään kokonaiselle luokalle tai toiminnolle, jopa yksittäiselle menetelmälle voidaan tehdä testi. Yksikkötestaus tehdään usein ohjelmiston suunnitteluvaiheessa ja sen tekijä on yleensä yksittäinen ohjelmoija.

Yksikkötestaus on parhaimillaan silloin, kun testit voidaan ohjelmoida niin, että testit voidaan ajaa yksin ilman muita ohjelman osia. Tätä pystytään edistämään tekemällä tekaistuja vastauksia eri osiin, jotta toisia osiota ei tarvita testissä. Testin tekijän täytyy osata testin alue kokonaisuudessaan ja siksi yleensä testin tekee ohjelman osan suunnittelija. Testit ovatkin yleensä vain tuloksia ohjelman osiosta.Testien tyylit vaihtelevat ohjelman osion mukaan, koska yksikkötestaus on juurikin yhden palasen testausta. Tämä mahdollistaa nopean virhe raportoinnin, joka tulostaa kohdan missä ja mikä ohjelmassa tuottaa virheen. (Zandstra M. 2008, 403-405.)

5.1. Yksikkötestauksen hyödyt

Yksikkötestauksessa on tavoitteena eristää kaikki ohjelman osat omiin testeihin, mikä mahdollistaa yksittäisten osioiden tarkistuksen. Yksittäiset testit antavat tarkat ja vaativat tulokset siitä, mitä ohjelmalta halutaan. Tämä tuottaa useita ja haluttuja etuja. (Zandstra M. 2008, 394-398.)

5.1.1 Ongelmien löytäminen

Yksikkötestaus tehdään ennen itse ohjelman osan koodia. Ohjelman osa todetaan valmiiksi, kun se läpäisee kaikki tehdyt testit. Extremeohjelmointi hyödyntää tätä käytäntöä, koska testi toimii samalla suunnitteluvaiheena. Samoja testejä ajetaan useasti uudestaan läpi ohjelmoinnin jatkuessa. Tämä mahdollistaa nopean havainnoinnin uusissa virhetilanteissa, joka johtuu uudelleen ohjelmoiduista kohdista tai joissain tilanteissa huonosti tehdystä testistä. Testin avulla nähdään nopeasti missä on ongelma, ja se tekee testien teon kannatavaksi. Aikaa säästyy ongelmien ratkomisessa, sillä testit näyttävät suoraan ongelman sijainnin. Yksikkötestaus ilmoittaa ongelmista jo ennen ohjelman antamista testaajille tai asiakkaille. Tämä tekee yksikkötestauksesta todella hyvän lisän ohjelmointiin. (Zandstra M. 2008, 406-409.)

5.2.1 Muokattavuus

Yksikkötestaus mahdollistaa ohjelman muokkauksen tulevaisuudessa niin, että varmistetaan moduulin toimivuus muokkauksen jälkeen. Tavoitteena on kirjoittaa jokaisesta toiminnosta ja menetelmästä testi, jotta kun virhetilanne tapahtuu se pystytään paikantamaan sekä tunnistamaan nopeasti.

Käyttövalmiit yksikkötestit helpottavat ohjelmoitsijoiden työtä, koska he voivat tarkistaa ohjelman toimintaa työskentelyn ohella.

Jatkuvassa yksikkötestaus ympäristössä ylläpitö on erittäin tehokasta ja sen harjoittaminen on mahdollista koko ohjelmoinnin aikana. Ylläpito pysyy tehokkaana myös ohjelman valmistuttua ja päivitystilanteissa. Syynä tähän on jatkuvan testauksen tapa testata testejä jatkuvasti. Kun ohjelma tallennetaan, jatkuva yksikkötestaus testaa ohjelman ja ilmoittaa, jos virhe on tapahtunut. Jatkuvan testauksen tehokkuus määrittyy siitä, kuinka hyvin testit ovat kirjoittu, mikä määrää samalla myös testien tarkkuuden.

5.3.1 Dokumentointi

Yksikkötestaus soveltuu erittäin hyvin Extremeohjelmointiin, koska se tuottaa omalla tavallaan dokumentoinnin itsestään. Syynä tähän on se, että yksikkötestiin ohjelmoidaan käyttöliittymä, joka näyttää, miten ohjelman osaa käytetään. Tämä toimii erittäin hyvin dokumentointina ja yleensä kertoo ohjelmoijalle enemmän kuin tuhat sanaa. Vaikka yksikkötestaus onkin hyvä tapa dokumentoida, monet ohjelmoijat dokumentoivat myös normaalisti yksikkötestauksen lisäksi. Dokumentointi tehdään usein siis kaksi kertaa.

Normaaliin dokumentointiin verrattuna yksikkötestaus ei vanhene, koska sen on elettävä ohjelman mukana. Tavallinen dokumentointi vanhenee aina silloin, kun ohjelmaa päivitetään. Tämä johtuu siitä, että dokumentointi pitää päivittää vielä erikseen päivityksen jälkeen.

6. Käytännön työn aloitus

Yksikkötestauksen käyttöönotto Yritys X:ssä onnistui yllättävän vaivattomasti, sillä kyseessä on standardi eli se on laajasti tuettu. Alun helppous ei kuitenkaan tarkoita sitä, että kyseessä olisi yksinkertainen prosessi. Tämä johtuu yksikkötestauksen käyttötavasta. Tapaa voisi verrata esimerkiksi ohjelmointikielen käyttöönottoon. Käyttöönottamisen jälkeen kaikki asiat tehdään itse testien ohjelmoinnin kautta.

6.1. Yksikkötestien luonti

Yksikkötestien luontia voidaan käyttää myös suunnitteluvaiheena. Extremeohjelmointi hyödyntää tätä käytäntöä ja on yksi syy, miksi toteutustavaksi otettiin Yksikkötestaus. Yritys X ohjelmoi Extremeohjelmointi:n mukaisesti.

Yksikkötestin ohjelmointi alkaa ohjelmanosion käyttöliittymän suunnittelulla, jos osio on uusi. Vanhassa ohjelmanosassa käyttöliittymä on tietenkin jo olemassa. Käyttöliittymän jälkeen tehdään suunniteltuun käyttöliittymään käyttötestejä. Testit vaihtelevat sallituista haitallisiin ja tämän tarkoituksena on nähdä, toimiiko ohjelma halutulla tavalla kaikissa tilanteissa. Testien kattaessa kaikki käyttämiseen liittyvät tavat, itse suunnitellun ohjelmanosion rakentaminen alkaa.

6.2. Testaus

Testaus tapahtuu käytännössä aina, kun ohjelmanosio valmistuu, saa päivityksen tai rakennusvaiheessa silloin, kun uusi asia on lisätty. Tällä varmistetaan ohjelman modulaarisuus, toimivuus ja yhteensopivuus. Testauksen määrä saattaa kuulostaa suurelta, mutta automaation ansiosta se on melko vaivatonta.

6.3. Käyttäminen

Yksikkötestauksen käyttö voi kuulostaa helpolta, mutta todellisuudessa käyttö on haastavaa, vaikkakin hyvin monipuolista. Suunnittelu ja testien kattavuuden varmistaminen ei ole yksinkertaista. Suunnitteluvaiheessa pitää olla tietoinen, mitä ohjelmanosiolta halutaan. Käyttöliittymästä pitää suunnitella yksinkertainen sekä helppo tapa käyttää ohjelmanosiota. Käyttöliittymän jälkeen pitää miettiä kaikki mahdolliset tilanteet käytössä ja haitallisissa tapauksissa. Molemmat näistä vievät aikaa ja vaativat tarkkaa suunnittelua sekä käsityksen siitä, mitä on tekemässä.

Ohjelmoinnin aikana voidaan tehdä yksikkötestausta helpottavia palautuksia. Esimerkiksi tietyssä virhetilanteessa voidaan palauttaa ilmoitus asiasta. Ilmoituksen avulla voidaan tehdä ylimääräinen testitulos, jonka avulla virhetila paikannetaan nopeasti. Muuten yksikkötestauksen käyttö onkin testien läpi ajamista, jotta nähdään ohjelmanosan toimivuus.

6.4. Tutkimus tulos

Ohjelmoinnin testauksen monista vaihtoehdoista yksikkötestaus osottautui yritykselle parhaaksi testaustavaksi. Monien vaihtoehtojen jälkeen yksikkötestauksen sopivuus Extremeohjelmoinnin kanssa nosti sen ylitse muiden. Yksikkötestaus on myös valittu PHP:n standardiksi, mikä vaikutti myös osaltaan valintaan.

Yksikkötestauksen kokonainen käyttöönotto osoittautui työlääksi ja pitkäksi projektiksi. Aluksi käyttöönotto tapahtui asentamalla. Tämän jälkeen seurasi GUI:n tekeminen yksikkötestauksesta saadusta datasta.

Yksikkötestaus teki testauksesta nopeaa ja vaivatonta. Käsin testaamiseen verrattuna yksikkötestauksen ainoana heikkoutena käsintestaukseen nähden on alun hitaus, mikä jotuu testin teosta. Tätä helpottaa kuitenkin se, että testit korvaavat suunniteluvaiheet. Suunnitteluvaihe vie yleensä tietotekniikaa käyttävissä yrityksissä paljon resursseja.

Huomasin pidemmällä aikavälillä tarkasteltuna yksikkötestauksen päihittävän käsintestauksen testauksen automatisoidun luonteen vuoksi. Käsintestauksessa kaikki ongelman osa-alueet testataan itse, kun taas yksikkötestaus tekee sen automaattisesti.

7. Yksikkötestauksen käyttöönotto

Projektin käyttöönotto on yritys salaisuus ja tästä johtuen siitä ei voi kirjoittaa työhön juuri mitään. Tästä syystä johtuen, seuraavaksi läpikäydään esimerkki testin luonti sekä käyttö. Yksikkötestauksen käyttöönotto on hankalampaa, sillä asennus ei onnistu perinteisellä asennustavalla. Yksikkötestauksen käyttöönotto on lyhyesti listattuna seuraava: tarvittavien osien valitseminen, asentaminen ja lopuksi käyttöönoton suurin vaihe eli testien luonti. Seuraavassa luvussa esitellään käyttöönoton tarpeet, ongelmat ja ratkaisut.

7.1. Ongelmatilanteet yksikkötestauksessa sekä ratkaisut niihin

Yksikkötestauksen käyttöönotossa nousee kaksi ongelmaa vahvasti esille. Ensimmäinen niistä on yksikkötestauksen asentaminen. Prosessi on haastava ja monimutkainen. Toisena vaikeutena voidaan mainita testien luonnin käsittäminen, sillä testien tulee toimia sekä dokumenttina, ylläpitotyökaluna että suunnitelmana.

Yksikkötestauksen ongelmiin on toki vastauksensa, jos niitä osaa etsiä. Asennuksessa ratkaisuna on tarkka manuaalin läpikäynti. Manuaali on vaikealukuinen ja laaja-alainen, mutta vain ymmärtämällä asennuksen tarpeet, siitä voi onnistua ilman ongelmia. Tähän vaaditaan kokemusta testaamisesta sekä ohjelmoinnista ylipäätään. Testien luonnin käsittämisen ongelmaan ratkaisuna on asennusongelman kaltaisesti manuaalin lukeminen. Testeiltä vaaditaan yritysmaailmassa paljon ja tarpeisiin perehtyminen auttaa löytämään oikean tavan toteuttaa testit.

7.2. Ohjelmoijalta vaadittavat ominaisuudet

Yksikkötestaus vaatii laajan osaamisen ohjelmoinnista, sillä jo pelkästään ohjelman asentaminen vaatii vähintään kahden ohjelmointikielen osaamista. Yksikkötestauksen asennuksessa ei käytetä PHP-kieltä.

Toisena tarpeena yksikkötestaus asennuksessa on oliopohjainen asennusympäristö, koska yksikkötestaus perustuu oliopohjaiseen ohjelmiston testaukseen. Tämä tarkoittaa PHP ohjelmoinnissa yhtä vaikeimmista osa-alueista, jota ei suositella opeteltavaksi kuin ainoastaan pitkän ajan ammattilaisille.

Lyhyesti sanottuna yksikkötestauksen hallinta vaatii ohjelmoijalta monipuolista tietopohjaa sekä vankkaa osaamista PHP-ohjelmoinnissa. Lisäksi oliopohjaisen ohjelmointiympäristön tulee olla tuttu.

8. Todettu parannus PHP yksikkötestauksen käyttöönoton jälkeen Yritys X:ssä

Kuvio 7: Kuviossa esitetään suunnitteluvaiheeseen tapahtunutta parannusta

Ennen parannusprojektiani suunnittelu oli vaikeampaa, sillä siihen kului enemmän aikaa ja se vaati suuren määrän dokumentointia. Suunnitteluvaiheeseen nivoutui ennen kiinteänä osana hitaus. Hitaus johtui siitä, että suunnitteluvaihe vaati paljon henkilöstöä sekä muita aikaa muulta työltä. Tämä johti suoraan kokonaisuuden hidastumiseen. Tämän tyyppinen suunnittelu ei tuottanut mitään käytännöllistä hyötyä itse ohjelmaan. Suunnittelu tuotti vain kirjallisen vision siitä, mitä pitäisi tehdä. Yrityksessä käytettiin Extremeohjelmointia, mikä on ristiriidassa käsin toteutettavan ohjelmoinnin kanssa. Suunnittelu piti dokumentoida rajallisen henkilöstön voimin ja dokumentin tekemiseen kului paljon aikaa.

Parannusprojektini jälkeen suunnitteluprosessi on nopeutunut. Suunnittelu tehdään yksikkötestauksen testien luomisen aikana, koska testiin tehdään ohjelman osaa varten tarvittavat asiat. Testin valmistuttua ohjelman osan ohjelmointi voi alkaa, joka on täysin Extremeohjelmoinnin mukainen tapa tehdä suunnittelu. Yksikkötestauksen testi toimii dokumenttina, josta selviää kaikki tarvittava tieto. Dokumentoivat testit korvaavat osan dokumentoinnin tarpeesta.

Kuvio 8: Kuviossa esitetään ohjelmointiin tapahtunutta parannusta

Ennen parannusprojektiani ohjelmointi oli vähän nopeampaa, sillä yksikkötestaus käyttää aiempaa ennemän aikaa testien luonnissa. Testauksen nopeus oli kuitenkin varsin hidasta, sillä käsin testaaminen tarkoittaa koko ohjelman läpi käymistä käsivoimin. Ylläpidettävyys on haastavaa, koska tilanteet ja ongelmat pitää selvittää ja etsiä koko ohjelmasta. Samat haastavuudet tulevat myös päivittämisessä vastaan.

Parannusprojektini jälkeen ohjelmointi on hidastunut, mutta vain testien luomisen ajaksi. Testien luomiseen tuhlattu aika saadaan nopeasti kiinni testauksen nopeuden vuoksi. Ongelmatilanteessa täytyy vain ajaa automaattiset testit läpi ja tulokset näyttävät suoraan ongelman sijainnin ja syyn. Ylläpito on testien avulla helppoa, koska ongelma nähdään testien avulla ja ne pystytään korjaamaan nopeasti. Samat myönteiset seikat tulevat esille päivittämisessäkin. Uuden päivityksen lisääminen sujuu kätevästi, kun päivityksen osan testit luodaan ja näin pidetään huolta osan toimivuudesta.

Kuvio 9: Kuviossa esitetään testaamiseen tapahtunutta parannusta

Ennen parannusprojektiani testaaminen oli tehtävä käsin, joka on välillä erittäin hidasta ja vaikeaa. Käsin testaaminen on myöskin yllättävän kallista, koska virheen etsiminen on pois työnteosta ja se vaatii vähintään yhden ihmisen. Ongelmat voivat, joskus olla hyvinkin syvällä ja vaikeissa paikoissa ohjelmassa. Vaikea ongelma saattaa kuluttaa useita työtunteja.

Parannusprojektini jälkeen testaaminen on nopeaa ja hyvinkin nopeaa, koska testaaminen hoituu automaattisesti Yksikkötestaus ajamisella. Yksikkötestaus säästää yrityksen rahaa, koska se poistaa useiden tuntien käsin testauksen ohjelmallisella automaattisella testauksella.

Kuvio 10: Kuviossa esitetään ylläpitoon tapahtunutta parannusta

Ennen parannusprojektiani ylläpitokin tehtiin käsin. Tämä oli haastavaa, sillä ongelmatilanteessa virhe oli löydettävä nopeasti. Käsintestaus on riskialtista, koska virheelliset korjaukset tai muutokset voivat luoda uusia ongelmia ja uudet ongelmat hidastavat korjausta. Tämän tyyppiset tilanteet käyvät usein kalliiksi yritykselle, koska mitä kauemmin ohjelma on poissa käytössä, sitä kalliimaksi sen käyttämättömyys tulee. Kaiken tämän lisäksi käsin testaus vaatii vähintään yhden ihmisen työpanoksen, mikä on taas pois työtunneista.

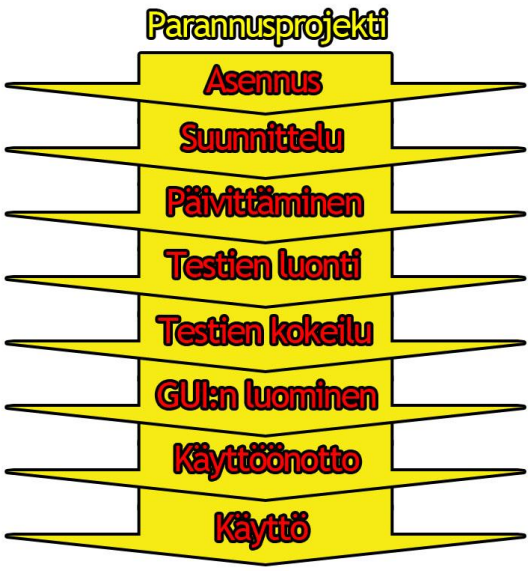
Parannusprojektini jälkeen ylläpito on helpompaa ja testaus on automaattista. Yksikkötestaus on riskittömämpi tapa testata ohjelmaa, koska itse ohjelmaan ei tarvitse tehdä mitään testausta varten. Tämä vähentää kustannuksia, koska testien avulla paikannetaan nopeasti ja tehokkaasti ongelmien syyt ilman ihmisen työtä. Ohjelmallinen testaus mahdollistaa käyttäjäystävällisen tavan ylläpitää ohjelmia.

Kuvio 11: Kuviossa esitetään päivittämiseen tapahtunutta parannusta

Edellä kuvaamieni kuvien tavoin päivitys nopeutui ja helpottui sekä virheiden määrä pieneni verrattuna aikaisempaan tilanteeseen.

9. Parannusprojektin vaiheet

Tässä luvussa kuvataan projektini eri vaiheet kuvan avulla sekä avataan kuvan sisältöä seuraavaksi tekstiosuuksissa.



Kuvio 12: Kuviossa esitetään parannusprojektin vaiheita

Toiminnallisen opinnäytetyöni käytännönosuuden vaiheet tapahtuivat kuvassa näkyvässä järjestyksessä. Parannusprojektini alkoi asennusvaiheesta, jossa asensin työympäristöön Yksikkötestaus Modulen. Vaitiolovelvollisuuteni takia en voi kertoa valitsemani Yksikkötestaus Modulen nimeä, enkä myöskään mitään, mikä voisi antaa vihjeitä yrityksessä käytettyihin asioihin.

Suunnitteluvaiheessa mietin toteutustavan, jota käytettäisiin nykyisessä ja mahdollisissa tulevissa projekteissa. Päivitysvaiheessa puolestaan kävin läpi nykyisen projektin ohjelman valmiita osioita, joihin suunnittelin ja loin yksikkötestauksen testit. Päivitysvaiheen jälkeen vuorossa oli uusien testien luominen. Tässä vaiheessa testien luonti muuttui suunnittelun tapaiseksi luomiseksi. Testit toimivat tässä vaiheessa ohjelman osion suunnitteluna, jossa ilmenee osan tarkoitus ja käyttöliittymä.

Testin valmistutta siirryin testien kokeiluvaiheesseen, jota tehdään pienien ohjelman osioiden valmistuttua. Testit näyttävät positiivisilla tuloksilla, milloin testit ovat valmiita. Testien läpi mennessä ohjelman osa voidaan todeta valmiiksi.

Seuraavaana vuorossa oli tehdä graafinen käyttöliittymä testien tuloksiin. Näiden avulla voitaisiin ajaa testejä nopeasti ja nähdä testien tulokset graafisesti. Esimerkkinä graafisesta käyttöliittymästä testit antavat vihreän värin onnistuessaan ja epäonnistuessaan punaisen värin kyseisille testeille.

Graafisen osuuden jälkeen yksikkötestaus olikin käyttöönottoa vaille valmis. Käyttöönottovaiheessa loputkin yrityksen henkilöt itseni ja esimieheni lisäksi alottivat yksikkötestauksen käytön. Viimeinen vaihe parannusprojektissani on tuotokseni käyttö yrityksen nykyisissä ja tulevissa projekteissa.

9.1. Ratkaisut ja perustelut

Yksikkötestaus oli päätetty yrityksen johdon puolesta ja minun piti tehdä käyttöönotto siitä yritykseen.

Päätöksiini on vaikuttanut eniten yrityksen ja työntekijöiden toiveet. Pyysin projektia aloittaessani työyhteisöä esittämään vapaasti ideoita, joita voisin mahdollisuuksien mukaan työssäni toteuttaa. Sainkin toiveita varsinkin yrityksen ohjelmoijilta eli henkilöiltä, jotka ohjelmallisen testauksen parissa jatkossa toimivat.

Toisena vaikutuksena tekemiini valintoihin ovat olleet ohjelmoinnin ammattilaiset erilaisista asiayhteyksistä kuten keskustelupalstoilta sekä blogeista. Koetin tehdä ratkaisuja niin, että

työympäristöni ja siihen kuuluvat henkilöt hyötyisivät projektistani parhaalla mahdollisella tavalla ja siitä olisi oikeasti hyötyä tulevaisuudessa.

10. Pohdinta

Käytännöllisen osuuden teon eli PHP yksikkötestauksen käyttöönoton Yritys X:ssä koin mielenkiintoiseksi projektiksi. Toki sitä oli välillä raskasta tehdä oman työn ohella, mutta uusien asioiden oppiminen sekä valmiin tuotoksen näkeminen oli palkitsevaa ja vaivan arvoista. Aikataulu oli väljä, mikä oli hyvä asia, sillä välillä yrityksessä oli pitkiäkin ajanjaksoja, jolloin projekti seisoi paikoillaan. Onnistuin käytännön osuudessa omasta mielestäni hyvin ja olen saanut myönteistä palautetta myös esimieheltäni. Kollegani ovat olleet tyytyväisiä automaattisen testauksen hyviin puoliin ja kommentoineet sitä, miten ovat ennen viitsineet kaiken tehdä käsin.

En osannut odottaa, että työtä tulee olemaan näin paljon. Kokopäivätyön, käytännön projektin, opinnäytetyön kirjoittamisprosessin sekä vapaa-ajan yhdistäminen oli ajoittain hyvin hankalaa ja stressaavaa. Välillä tuntui siltä, että on helpompi vain jättää työ kesken, kun työpaikka kuitenkin jo on. Tämän vuoksi epäonnistuin työni kirjaamisessa, sillä en jaksanut panostaa siihen niin paljon kuin alussa kuvittelin. Lisäksi salassapitovelvollisuus rajoitti prosessin kuvailua radikaalisti. Työn kirjoittaminen olisi ollut helpompaa, jos olisin voinut kirjoittaa asioista niiden oikeilla nimillä.

Jos voisin tehdä jotakin toisin, käyttäisin vähemmän aikaa itse käytännön työhön, jotta aikaa jäisi myös kirjoittamiselle. En koe kirjoittamista mielekkääksi, enkä ole siinä erityisen hyvä, joten myös se on tuonut omat haasteensa kirjoitusprosessille. Toisaalta, jos olisin panostanut käytännön osuuteen vähemmän, Yritys X olisi saattanut olla tyytymätön lopputulokseen, eikä ohjelmallista testausta ehkä olisikaan otettu käyttöön. Kuitenkin mielestäni toiminnallinen opinnäytetyö on hyödyllisin silloin, kun siitä oikeasti on jotakin konkreettista käytännön hyötyä. Toiminnallinen työ tulee tehdä tarpeeseen, ei pelkän opinnäytetyön aikaansaamisen vuoksi.

Olen oppinut projektin aikana asioita, jotka tulevat olemaan osa ammattitaitoani lopun elämääni. En vaihtaisi oppimiani taitoja mihinkään, vaikka niiden hinta tulikin opinnäytetyön kirjallisen osuuden hinnalla. Olen pettynyt opinnäytetyöni suppeuteen ja keskeneräisyyteen. Pettymyksestä huolimatta uskon oppineeni enemmän kuin moni muu opinnäytetyönsä aikana.

Lähteet

Dustin, E., Rashka, J., Paul, J. 1999. Automated Software Testing. Introduction, Management and Performance. Addison-Wesley.

Haikala, I., Märijärvi, J., 2004. Ohjelmistotuotanto. Talentum.

Learning to Love Unit Testing. 2002. The Software Testing & Quality Engineering Magazine. Viitattu 28.9.2012.
http://www.stqemagazine.com/featured.asp?stamp=1129125440

Myers, G. 2004. The Art of Software Testing. John Wiley & Sons.

Shafik, D., Mitchell, L. & Turland, M. 2011. PHP Master: Write cutting edge code. SitePoint.

The Model-View-Controller (MVC) Design Pattern for PHP. 2004. Tony Marston. Viitattu 29.9.2012.
http://www.tonymarston.net/php-mysql/model-view-controller.html

Tietotekniikan sovellusprojekti. 2004. Testaussuunnitelma. Viitattu 28.9.2012.
http://sovellusprojektit.it.jyu.fi/kaakkuri/materiaali/testaussuunnitelma.html#_Toc91483622

Zandstra, M. 2008. PHP Objects, Patterns, and Practice, Second Edition. Apress.

Liitteet

11. Yksikkötesti

Seuraavaksi käyn läpi esimerkkien avulla yksikkötestien luonnin ja käyn läpi, mitä testien eri vaiheissa tapahtuu.

11.1. Testin luonti

Seuraavaksi koodia ja jokaisen kohdan väliiin kommentia koodista:

```
<?php
require_once 'PHPUnit.php';
require_once 'EsimerkkiKauppareissu.php';

class Kauppareissu extends PHPUnit
{
    /**
     * @mahdollisuus 1
     */
    public function uusiOstoskori0()
    {
        $this->given('Uusi Kauppareissu')
             ->then(Ostoskori', 0); // Tyhjä
    }
```

Ensimmäisessä mahdollisuudessa esimerkkinä on uuden kauppareissun lähtötilanne, jossa ostoskorin pitää olla tyhjä.

```
/**
 * @ mahdollisuus 2
 */
public function ostoksetPerunaa20()
{
    $this->given('Uusi Kauppareissu')
        ->when('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->and('Osta', 'Peruna', 1)
        ->then('Ostoskorissa', 'Peruna', 20);
}
```

Toisessa mahdollisuudessa tarkistetaan ostamisen tarkkuus, jotta varmistetaan yhteismäärän toimivuus määrästä huolimatta.

```
/**
 * @ mahdollisuus 3
 */
public function ostoksetPeruna10()
{
    $this->given('Uusi Kauppareissu')
        ->when('Osta', 'Porkkana', 5)
        ->and('Osta', 'Peruna', 10)
        ->and('Osta', 'Pihvi', 5)
        ->and('Osta', 'Maito', 5)
        ->and('Osta', 'Salaatti', 5)
        ->then('Ostoskorissa', 'Peruna', 10);
}
```

Kolmannessa mahdollisuudessa tarkistetaan, että muut ostokset eivät vaikuta muiden ostoksien lukumäärään.

```
/**
 * @ mahdollisuus 4
 */
public function vaihdaHintat100()
{
    $this->given('Uusi Kauppareissu')
        ->when('Hinta', 'Porkkana', 100)
        ->and('Hinta', 'Peruna', 100)
        ->and('Hinta', 'Pihvi', 100)
        ->and('Hinta', 'Maito', 100)
        ->and('Hinta', 'Salaatti', 100)
        ->then('Hinnat', array(
            'Porkkana',
            'Peruna',
            'Pihvi',
            'Maito',
            'Salaatti',
        ), 100);
}
```

Neljännessä mahdollisuudessa vaihdetaan haluttujen tuotteiden hintojen arvoksi sata ja lopuksi tarkistetaan hintojen vaihtuneen.

```
/**
 * @ mahdollisuus 5
 */
public function ostoksetVarasto0()
{
    $this->given('Uusi Kauppareissu')
        ->when('Osta', 'Porkkana', 100)
        ->and('Osta', 'Peruna', 100)
        ->and('Osta', 'Pihvi', 100)
        ->and('Osta', 'Maito', 100)
        ->and('Osta', 'Salaatti', 100)
        ->and('Osta', 'Lohi', 100)
        ->and('Osta', 'Muikku', 100)
        ->and('Osta', 'Rahka', 100)
        ->and('Osta', 'Banaani', 100)
        ->and('Osta', 'Karkki', 100)
        ->and('Osta', 'Suklaa', 100)
        ->and('Osta', 'Mehu', 100)
        ->and('Osta', 'Mansikka', 100)
        ->and('Osta', 'Puuro', 100)
        ->and('Osta', 'Leipä', 100)
        ->then('Varasto', 0);
}
```

Viidennessä mahdollisuudessa ostamme kaupan varaston tyhjäksi ja tarkistamme, että varasto on tyhjentynyt oikein.

```
public function runGiven(&$kauppa, $toiminto, $arvo = NULL, $numero = 0)
{
   switch($toiminto)
   {
      case 'Uusi Kauppareissu':
      {
         $kauppa['kauppa']  = new Kauppareissu;
         $kauppa['ostoskori'] = new Ostoskori;
         $kauppa['varasto'] = new Varasto(100);
         break;
      }

      default:
      {
         return $this->notImplemented($toiminto);
      }
   }
}
```

Testien ajaminen onnistuu erilaisilla tapauskohtaisilla tapahtumilla, mitä kutsutaan tulosten saamiseksi. Ylempänä on uuden kauppareissun luominen ja kutsuminen.

```
public function runWhen(&$kauppa, $toiminto, $arvo = NULL, $numero = 0)
{
   switch($toiminto)
   {
      case 'Osta':
      {
         $kauppa['ostoskori']->osta($arvo, $numero);
         $kauppa['varasto']->saldo($arvo, $numero);
         break;
      }
      case 'Hinta':
      {
         $kauppa['kauppa']->hinta->muuta($arvo, $numero);
         break;
      }
```

```
        default:
        {
            return $this->notImplemented($toiminto);
        }
    }
}

public function runAnd(&$kauppa, $toiminto, $arvo = NULL, $numero = 0)
{
    $this->runWhen($kauppa, $toiminto, $arvo, $numero);
}
```

Ylempänä on uuden kauppareissun käyttämiseen tarkoitetut testaukset, joiden avulla pystytään luomaan tuloksia. Esimerkissä on ostamisen toiminto ja hinnan asettamisen toiminto.

```
public function runThen(&$kauppa, $toiminto, $arvo = NULL, $numero = 0)
{
    switch($toiminto)
    {
        case 'Ostoskori':
        {
            $kauppa['ostoskori']->tilanne->onko($arvo);
            break;
        }

        case 'Ostoskorissa':
        {
            $kauppa['ostoskori']->tilanne->tuote($arvo, $numero);
            break;
        }
        case 'Hinnat':
        {
```

```
            if (is_array($toiminto))
            {
               $arvo = array_keys($arvo);
               foreach ($arvo as $tuote);
               {
                  $kauppa['kauppa']->hinta->on($tuote, $numero);
               }
            }
            else
            {
               $kauppa['kauppa']->hinta->on($arvo, $numero);
            }
            break;
         }

         case 'Varasto':
         {
            for ($i = $kauppa['varasto']->tilanne; $i <= $kauppa['varasto']->saldo->aloitus; $i++)
            {
               $kauppa['kauppa']->varasto(0);
            }

            $this->assertEquals($arvo, $kauppa['kauppa']->varasto->tilanne);

            break;
         }

         default:
         {
            return $this->notImplemented($toiminto);
         }
      }
}
?>
```

Lopuksi esimerkki koodissa on lopputuloksien testaus, joka antaa testeistä tulokset. Esimerkissä toimintoina ovat ostoskorin tavaroiden määrän tarkistaminen, tietyn tuotteen määrän tarkistaminen ostoskorista, tuotteiden hintojen tarkistamista ja varaston saldon tarkistaminen.

11.2. Testin tulos

Seuraavaksi esimerkkituloksia testistä:

```
phpunit -- printer PHPUnit_Extensions_ResultPrinter EsimerkkiKauppareissuTulos
PHPUnit 3.7.0 by Sebastian Bergmann.


EsimerkkiKauppareissuTulos

 [x] Ostoskorin koko on 0

   Given Uusi Kauppareissu
    Ostoskorin koko pitäisi olla 0
 [x] Ostoskorissa on 20 perunaa

   Given Uusi Kauppareissu
    When Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
     and Ostaa perunaa 1
    Then Perunaa pitäisi olla 20
```

[x] Ostoskorissa on 10 perunaa

```
  Given Uusi Kauppareissu
  When Ostaa porkkanaa 5
   and Ostaa perunaa 10
   and Ostaa pihviä 5
   and Ostaa maitoa 5
   and Ostaa salaattia 5
  Then Perunaa pitäisi olla 10
```

[x] Tuotteiden hinnat on 100

```
  Given Uusi Kauppareissu
  When Hintaa muutetaan porkkanassa 100
   and Hintaa muutetaan perunassa 100
   and Hintaa muutetaan pihvissä 100
   and Hintaa muutetaan maidossa 100
   and Hintaa muutetaan salaatissa 100
  Then Hintojen pitäisi olla 100
```

[x] Varaston saldo on 0

```
  Given Uusi Kauppareissu
  When Ostaa porkkanaa 100
   and Ostaa perunaa 100
   and Ostaa pihviä 100
   and Ostaa maitoa 100
   and Ostaa salaattia 100
   and Ostaa lohta 100
   and Ostaa muikkua 100
   and Ostaa rahkaa 100
   and Ostaa banaania 100
   and Ostaa karkkia 100
   and Ostaa suklaata 100
   and Ostaa mehua 100
   and Ostaa mansikkaa 100
   and Ostaa puuroa 100
   and Ostaa leipää 100
  Then Varaston saldon pitäisi olla 0
```

Scenarios: 5, Failed: 0, Skipped: 0, Incomplete: 0.

Tästä tekstimuotoisesta tulosteesta näemme testien tapahtumat ja niiden tulokset. Lopusta näemme testeissä tapahtuneet tulokset, jotka ovat tässä esimerkissä onnistuneita. Testien tuloksia pystyy näyttämään halutulla tavalla, koska tulokset tulevat raakana datana ja tästä syystä ne pystytään käsittelemään halutulla tavalla.