Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Arttu Salonen
Jukka Mäkelä

Opinnäytetyö

Testilähtöiseen ohjelmistokehitykseen siirtyminen ohjelmistoprojektissa

Työn ohjaaja Paula Hietala
Työn tilaaja SAKU ry
Tampere 12/2009

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma

Tekijät	Arttu Salonen, Jukka Mäkelä
Työn nimi	Testilähtöiseen ohjelmistokehitykseen siirtyminen ohjelmistoprojektissa
Sivumäärä	55
Valmistumisaika	12/2009
Työn ohjaaja	Paula Hietala

TIIVISTELMÄ

Opinnäytetyön tavoitteena on perehtyä testilähtöiseen ohjelmistokehitykseen liittyviin teorioihin, kuten olioperustaisuuteen, refaktorointiin ja yksikkötestaukseen sekä suunnitteella ja toteuttaa osalle kohdesovelluksen luokista yksikkötestit. Opinnäytetyön tavoitteena on myös palvella mahdollisia tulevia kehittäjiä kohdesovelluksen testilähtöisessä ohjelmistokehityksessä. Kohdesovellus on toisen asteen ammattiin opiskelevien liikunnan ja hyvinvoinnin edistämiseen tarkoitettu Alpo.fi-hyvinvointiportaali.

Kohdesovellusta ei ole alkujaan kehitetty testilähtöisesti eikä sen kehityksessä ole aiemmin käytetty yhtenäistä tapaa. Kuitenkin kohdesovellus on suurimmaksi osaksi olioperustainen, mutta sisältää myös paljon koodia, joka ei sellaisenaan ole yksikkötestattavissa.

Yksikkötestien toteuttamista edelsi testiympäristön luonti, joka sisälsi muutoksia hakemistorakenteeseen, testiympäristön rajaamisen, refaktoroimista sekä uuden toimintaympäristön asettamisen. Opinnäytetyössä eritellään testatut luokat sekä moduulit, ja käydään läpi, mitä refaktorointeja testien kirjoittaminen sisälsi. Lisäksi käsitellään yksikkötestauksessa sovelluksesta löytyneitä virheitä sekä kohdattuja ongelmia.

Yksikkötestauksen käyttöönoton lisäksi pohditaan kohdesovelluksen sellaisia kohtia, joihin tulee kiinnittää huomiota kohdesovelluksen testilähtöisessä ohjelmistokehityksessä.

Avainsanat testilähtöinen, ohjelmistokehitys, yksikkötestaus, refaktorointi

TAMK University of Applied Sciences
Business Information Systems

Writers	Arttu Salonen, Jukka Mäkelä
Thesis	Applying Test-driven Development in a Software project
Pages	55
Graduation time	12/2009
Thesis supervisor	Paula Hietala

ABSTRACT

The purpose of this thesis is to study the theories related to test-driven development, such as object-oriented programming, refactoring and unit testing. The planning and implementing of unit tests for a number of classes of the target application will also be carried out. Another object of the thesis is also to serve possible future developers in the test-driven development of the target application. The target application is Alpo.fi, a portal designed to advance engagement to physical exercises and the well-being of secondary school students.

Test-driven development was not applied in the development of the target application, and the development has been disunited. Regardless of this, the target application has been implemented using object-oriented programming for the most part, though it contains a great deal of code that cannot directly be unit tested.

Before writing unit tests, the testing environment had to be created, which included updating the application directory structure, defining the testing environment, refactoring and adding a new executing environment to the target application. In the thesis the tested classes and modules are specified and an overview of what refactoring was carried is given. Furthermore, the bugs and the problems that came up in unit testing are also covered.

In addition to the retrofitting of unit tests, the issues that should be taken into consideration in the test-driven development of the target application are also discussed.

Keywords test-driven development, phpunit, unit testing, refactoring

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma

Sisällysluettelo

1 Johdanto

Testauksen merkitystä painotetaan usein ohjelmistotuotannon opinnoissa. Kuitenkin testauksen merkittävyyttä voi olla vaikea täydellisesti sisäistää pelkän käsitteellisen havainnollistamisen pohjalta – vasta työskentely projektissa, jossa testaaminen on vähäistä tai puuttuu kokonaan, antaa kunnollisen kokonaiskuvan siitä, miten suurta osaa testaaminen ja johdonmukainen testaussuunnitelma esittävät ohjelmistokehityksessä.

Opinnäytetyössä kuvaamme testilähtöisen ohjelmistokehityksen käyttöönoton Alpo.fi-hyvinvointiportaalissa. Kohdesovellus on toteutettu PHP-ohjelmointikielellä ja pääosin olioperustaisesti. Sitä ei kuitenkaan ole kehitetty testilähtöisesti eikä se siten sisällä ainuttakaan yksikkötestiä. Opinnäytetyössä suunnittelemme yksikkötestauksen käyttöönoton ja kerromme, miten toteutimme yksikkötestit sovellukseen. Valitsimme yksikkötestien toteuttamiseen PHPUnit-yksikkötestaustyökalun, jonka perusteet olivat tulleet tutuiksi ohjelmistotuotannon opinnoissa.

Sovelluksen kehitykselle on ollut luonteenomaista uusien ominaisuuksien jatkuva ja nopea kehitys sekä vähäiset testausresurssit. Ketterän testilähtöisen ohjelmistokehityksen käyttöönoton tarkoituksena on parantaa sovelluksen luotettavuutta ja helpottaa sen ylläpitämistä ja kehitystyötä. Yksikkötestaus ei kuitenkaan korvaa sovellustestausta, jota emme opinnäytetyössä käsittele. Yksikkötestit testaavat ja varmistavat ohjelmakoodin yksikköjen eli luokkien metodien ja funktiokirjastojen funktioiden toimivuuden.

Työharjoittelun aikana kokosimme listan sovelluksen sellaisista ominaisuuksista ja kokonaisuuksista, jotka mielestämme vaativat kehittämistä. Toimeksiantaja valitsi tältä listalta haluamansa kohdat, joita lähdimme toteuttamaan kesätöinä. Yksi listan kohdista, joka ei päässyt kesän työlistalle, oli yksikkötestauksen käyttöönotto kohdesovelluksessa. Koska kyseinen kohta oli mielestämme erittäin tärkeä, päätimme valita sen opinnäytetyömme aiheeksi.

Opinnäytetyössä perehdymme testilähtöiseen ohjelmistokehitykseen ja yksikkötestaukseen. Koska kohdesovellus on melko laaja web-sovellus, pohdimme opinnäytetyössä myös, miten ajamme yksikkötestit. Lisäksi analysoimme yksikkötestien käyttöönoton tulokset: mitä virheitä yksikkötestaus paljasti ja mitä muutoksia kohdesovellukseen tehtiin. Opinnäytetyön myötä sovelluksen kehityksessä on tarkoitus siirtyä testi-

lähtöisyyteen, joten esitämme kattavan kuvauksen teoriasta ja yksikkötestien käyttöönotosta, jota mahdolliset kehittäjät voivat käyttää apuna sovelluksen testilähtöisessä kehittämisessä.

Kohdesovelluksen testaus on ollut puutteellista ja riittämätöntä – se on jäänyt pääosin loppukäyttäjien tehtäväksi. Olemme testanneet kohdesovellusta itse ohjelmointityön jälkeen tai sen aikana tekemällä testidokumentin, johon olemme valinneet testattavan ominaisuuden tärkeimpien tilanteiden testitapauksia. Testidokumentteja ei ole kuitenkaan tehty jokaiselle ominaisuudelle, koska olemme testanneet ominaisuuksia myös pelkästään kokeilemalla ja yrittämällä rikkoa niitä. Näilläkin tavoilla on virheitä löytynyt, mutta kovin suurta varmuutta ominaisuuksien toimimisesta ei näillä keinoilla saavuteta. Testaamiseen tarvitaan siis jokin hallittu ja jatkuvasti mukana oleva menetelmä.

2 Alpo-hyvinvointiportaali

Tässä luvussa tutustutaan opinnäytetyön kohdesovelluksen taustaan ja käytettyihin tekniikoihin.

2.1 Tausta

Opinnäytetyön kohdesovellus on ammattiin opiskelevien liikunta- ja hyvinvointiportaali Alpo.fi (www.alpo.fi). Kohdesovellus on vuonna 2007 aloitettu Tampereen ammattikorkeakoulun tutkimus- ja kehitysprojekti (T&K), jonka asiakkaana on Suomen ammatillisen koulutuksen kulttuuri- ja urheiluliitto SAKU ry. Sovellus on ollut käytössä jo muutaman vuoden, ja projektin käynnistymisestä alkaen sitä on ollut kehittämässä monta työharjoittelijaa Tampereen ammattikorkeakoulusta.

Sovellus on rakennettu toisen asteen ammattiin opiskelevien liikunnan harrastamisen edistämiseksi ja ylläpitämiseksi sekä tueksi ammattiosaajan työkykypassin suorittamiseen. Opetushallitus ja opetusministeriö aloittivat vuonna 2006 ammattiosaajan työkykypassin kehitystyön yhdessä SAKU ry:n kanssa. Työkykypassin tarkoituksena on edistää opiskelijoiden työ- ja toimintakykyä motivoimalla ja innostamalla opiskelijat pitämään huolta terveydestään ja hyvinvoinnistaan jo opintojen aikana. (EDU - Ammattiosaajan työkykypassi 2008.)

Sovellus toimii muun muassa opiskelijoiden ja oppilaitosten henkilökunnan liikuntapäiväkirjana. Sovellukseen voi kirjata liikuntasuorituksiaan ja saada niistä palautetta. Sovellus tukee myös ammattiosaajan työkykypassin suorittamista: se antaa mahdollisuuden kirjata ja tulostaa passisuorituksia. Opiskelijat ja henkilökunnan jäsenet voivat antaa passiohjaajilleen oikeudet tarkastella passiin liittyviä suorituksia. Passiohjaajat voivat myös myöntää ja tulostaa opiskelijoiden passitodistuksia. Sovellus on myös liikunnan, hyvinvoinnin ja työkyvyn edistämisen tietopankki sekä eri käyttäjäryhmiensä (opiskelija, henkilöstö, passiohjaaja ja ylläpitäjä) tiedotuskanava.

Aloitimme työharjoittelun kohdesovelluksen teknisinä kehittäjinä syksyllä 2008. Viisi kuukautta kestäneen harjoittelun jälkeen jatkoimme edelleen projektissa osa-aikaisina, ja lukukauden päättymisen myötä aloitimme jälleen täyspäiväisinä kesätöiden merkeissä. Merkittävämpiä kehittämiämme ominaisuuksia ovat monikielisyyden tuki, jo-

ka sisälsi ruotsinnoksen teknisen toteutuksen, sekä ominaisuuden, joka mahdollistaa tilastojen ja kuntotestien tulosten graafisen esittämisen. Lisäksi kehitimme ominaisuuden, jossa passiohjaaja voi myöntää työkykypassitodistuksia opiskelijoilleen.

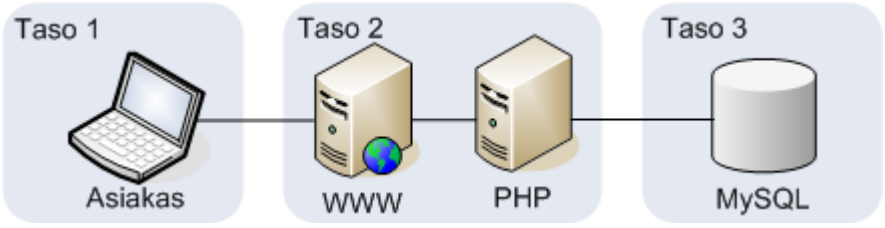
Syksyllä 2009 Alpo-hyvinvointiportaalin käyttäjäksi oli rekisteröitynyt jo noin 1000 ammattiin opiskelevaa opiskelijaa, ja sen kautta oli myönnetty 27 työkykypassia.

2.2 Sovelluksen toimintaperiaate

Kohdesovellus on toteutettu PHP 5 -kielellä, ja sovelluksen tietojen tallentamiseen käytetään MySQL-relaatiotietokantaa. Vuonna 2004 julkaistuun PHP 5:een uusittiin kokonaan kielen oliomalli eli kielen tapa käsitellä olioita, mikä paransi merkittävästi PHP:n tukea olio-ohjelmoinnin tekniikoille. PHP 5:lla voi ohjelmoida sekä olioperustaisesti että perinteisellä proseduraalisella menetelmällä. (PHP: New Object Model – Manual 2009.) PHP-palvelimen muodostama käyttöliittymä eli XHTML-koodi noudattaa XHTML 1.0 Strict -standardia. JavaScript- ja Ajax-tekniikkaa on käytetty elävöittämään käyttöliittymää ja parantamaan sovelluksen käytettävyyttä. Esimerkiksi passiohjaajien tietojen hakeminen tietokannasta alasvetovalikkoon toimii Ajax-tekniikalla. Käyttöliittymän ulkoasun määritykset ovat suurimmaksi osaksi CSS-tyylitiedostoissa.

Kolmitasoarkkitehtuuri

Kohdesovellus on tyypillinen kolmitasoarkkitehtuurin mukainen web-sovellus. Seuraavasta kuvasta (Kuva 2.1) selviää kohdesovelluksen yleisarkkitehtuuri.



Kuva 2.1. Sovelluksen kolmitasoarkkitehtuuri

Kolmitasoarkkitehtuurin tasot ovat käyttöliittymätaso (taso 1), sovelluslogiikkataso (taso 2) ja tietojen varastointiin käytettävä taso (taso 3). Ensimmäisessä tasossa asiakkaan selain näyttää sovelluksen käyttöliittymän asiakkaalle tasolta 2 saadun tiedon perusteella. Ensimmäisessä tasossa asiakas voi olla vuorovaikutuksessa sovelluksen

kanssa: asiakas voi lähettää sovellukseen tietoja ja tarkastella sovelluksen tietoja. Tasossa 2 sijaitsee sovellusta suorittava PHP-palvelin ja WWW-palvelin, joka välittää PHP-palvelimen palauttamat tiedot asiakkaalle. Tässä tasossa sijaitsee sovelluksen toimintalogiikka. Tasossa 3 sijaitsee sovelluslogiikkatason käyttämä tietokanta. (Designing Multi-Tier IIS Applications 2009.)

2.3 Sovelluksen kehityshistoria

Vuonna 2007, jolloin sovelluksen kehitys alkoi, sovellusta ei kehitetty olioperustaisesti, vaan proseduraalisesti sekä tavalla, jossa PHP-tiedostossa XHTML-koodin sekaan upotettiin PHP-koodia. Näin PHP-tiedostoon tuli useita php-tagin aloituksia ja lopetuksia. Lisäksi PHP-tiedostoon ei toteutettu ainuttakaan funktiota eikä käyttöliittymää ja sovelluslogiikkaa erotettu toisistaan.

Proseduraalisessa eli osittavassa ohjelmointitavassa jokin suoritettava korkean tason toiminto pilkotaan tarkempiin alemman tason toimintoihin, jotka voidaan taas jakaa vastaavalla tavalla (Koskimies 2000: 22–23). Proseduraalisessa ohjelmoinnissa ohjelman sisältämä tieto on muuttujissa, joita välitetään parametreina, kun taas olio-ohjelmoinnissa tiedot ja toiminnallisuudet ovat olioilla (Schlossnagle 2004: 38).

Aloittaessamme työharjoittelun sovelluksen parissa syksyllä 2008 oli kohdesovelluksen tietokannan käyttö uusittu olioperustaiseksi. Kehitimme sovellusta aluksi proseduraalisesti ja uusimme funktioita sisältämättömiä PHP-tiedostoja proseduraaliseksi. Kun taidot kehittyivät, aloimme kehittää sovellusta olioperustaisesti ja niin, että erotimme käyttöliittymän sovelluslogiikasta. Opinnäytetyön kirjoitushetkellä sovelluksessa löytyy funktioita sisältämättömiä PHP-tiedostoja sekä proseduraalisesti ja olioperustaisesti toteutettua koodia.

3 Kohti testilähtöistä ohjelmistokehitystä

Ennen testilähtöisen ohjelmistokehityksen käsittelemistä, käydään läpi siihen liittyviä keskeisiä teorioita, joihin kuuluvat olioperustaisuuden peruskäsitteet, refaktorointi ja yksikkötestaus. Lisäksi käsitellään PHPUnit-sovelluskehystä, jota käytetään yksikkötestien toteuttamiseen. Luku päättyy käytännön esimerkkiin esitetyistä teorioista.

3.1 Olioperustaisuuden peruskäsitteitä

Olioperustaisessa ohjelmistokehityksessä kehitettävään sovellukseen mallinnetaan sovellusalueen ja ongelmakentän keskeisiä käsitteitä luokkina (Lecky-Thompson, Nowicki & Myer 2009: 3-4). Luokka määrittelee luokasta muodostettujen olioiden rakenteen eli mitä tietojäseniä (attribuutteja) ja toimintoja (operaatioita) olioilla on. Olio on luokan ajon aikainen ilmentymä. (Koskimies 2000: 37–38.)

Esimerkiksi pankkiautomaattia simuloivaan sovellukseen voitaisiin toteuttaa seuraavat luokat: Tili, Tilitapahtuma, Asiakas, Pankkikortti, Kortinlukija, Näppäimistö, Näyttö ja Rahaluukku. Koska käsitteillä on reaalimaailmassa ominaisuuksia, esimerkiksi asiakkaalla on nimi, osoite ja puhelinnumero, ne voidaan yleensä suoraan asettaa Asiakas-luokan attribuuteiksi eli tietojäseniksi. Myös käsitteeseen liittyvät toiminnot voidaan usein mallintaa suoraan luokan käyttäytymistä kuvaaviksi operaatioiksi. Esimerkiksi Kortinlukija-luokalla voisi olla operaatiot lueTiedot(), työnnäUlos(), pidätä() ja onkoKorttiAsetettu(). (Bjork 2004.)

Kapselointi tarkoittaa olion attribuuttien ja operaatioiden suojaamista toisilta olioilta. Kapseloinnissa olio tarjoaa pelkästään rajapinnan, jonka kautta toiset oliot pystyvät käyttämään kyseistä oliota. Olio ei paljasta ulospäin sitä, miten toiminnot on toteutettu. Sen sijaan olio näyttää ulospäin toiminnot, jotka olio pystyy tekemään. Kapseloinnissa olion sisältämät tiedot ja toiminnot pakataan suojatuksi kokonaisuudeksi. (Koskimies 2000: 30, 49.)

Kapseloinnin periaatteiden mukaan mikään olion attribuutti ei saa olla julkinen (public). Jos olion attribuutti on julkinen, toiset oliot pääsevät lukemaan ja muokkaamaan olion attribuuttia ilman, että kyseinen olio tietää siitä mitään. Tämä rikkoo tiedon ja toimintojen muodostamaa kokonaisuutta sekä vähentää sovelluksen modulaarisuutta.

Julkisten attribuuttien sijaan määritellään attribuutit yksityisiksi (private), ja tehdään attribuutin arvon asettava (set) ja palauttava (get) metodi. Tällöin attribuutin arvo asetetaan ja palautetaan yhdessä paikassa eikä eri puolilla sovellusta. (Fowler, Beck, Brant, Opdyke & Roberts 1999: 206.)

Periytyminen tarkoittaa mekanismia, jossa luokan piirteet eli attribuutit ja operaatiot siirtyvät toiselle luokalle. Periytymisellä voidaan toteuttaa sovellukseen yläkäsitteitä omaavat käsitteet. Esimerkiksi jos luokat Auto ja Polkupyörä periytyvät luokasta Kulkuneuvo, luokan Kulkuneuvon tietojäsenet ja toiminnot siirtyvät aliluokille Auto ja Polkupyörä. Periytyminen edistää ohjelmakoodin uudelleenkäyttöä, koska aliluokissa ei toteuteta periytyneitä piirteitä uudestaan. (Koskimies 2000: 68–69.)

Yliluokkien yksityiset piirteet periytyvät aliluokkiin, mutta aliluokat eivät saa itse viitata näihin periytyneisiin piirteisiin. Jos periytyneet piirteet halutaan antaa aliluokkien käyttöön, on ne määriteltävä protected-määreellä. (Koskimies 2000: 75–76.)

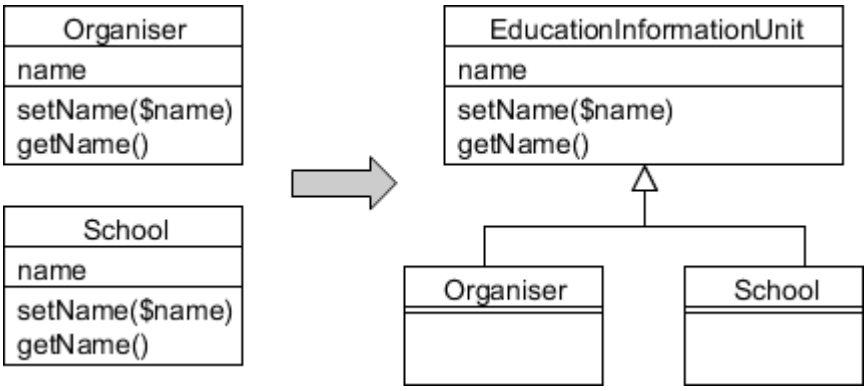
Periytymistä on välttämätöntä käyttää, kun halutaan luoda olio luokasta, jonka yliluokka on abstrakti. Abstraktista luokasta ei voi luoda oliota. (Koskimies 2000: 90–91.) Abstrakteilla luokilla voidaan mallintaa abstraktit käsitteet, joilla ei ole suoraan ilmentymää reaalimaailmassa, esimerkiksi ajateltaessa kulkuneuvoa ajatellaan aina jotakin konkreettista käsitettä, kuten autoa tai polkupyörää. Siten Kulkuneuvo-luokka mallinnettaisiin abstraktiksi, ja konkreettiset Auto- ja Polkupyörä-luokka periytyisivät Kulkuneuvo-luokasta.

3.2 Refaktorointi

3.2.1 Refaktoroinnin tarkoitus ja tavoitteet

Refaktorointi (refactoring) tarkoittaa ohjelmiston koodin rakenteen parantamista sekä koodin rakenteeseen tehtävää yksinkertaista ja lyhyttä muutosta. Refaktoroinnilla on siis kaksi merkitystä: yksittäinen muutos ja muutoksien kokonaisuus. Refaktoroinnissa ei muuteta koodin tuottamaa lopputulosta, vaan parannetaan sitä, miten lopputulokseen päästään. Jotta ohjelmisto pysyisi jokaisen muutoksen jälkeen toimintakykyisenä, ohjelmistoa refaktoroidaan hallitusti ja systemaattisesti yksi muutos kerrallaan. Yksittäinen refaktorointi johtaa usein myös useampiin refaktorointeihin. (Fowler ym. 1999: 53–54.)

Martin Fowler on koonnut listan refaktoroinneista (catalog of refactorings), joissa käydään läpi esimerkkien avulla, miten jokin tietty koodin rakenteessa oleva puute voidaan korjata. Esimerkiksi (Kuva 3.1), jos kahdella tai useammalla aliluokalla on yhteinen piirre, se voidaan siirtää luokkahierarkiassa ylemmälle tasolle eli yliluokkaan (Pull Up Field, Pull Up Method). Jos jokin metodi on liian pitkä eli se tekee useita eri asioita, metodin eri tehtävät jaetaan omiin metodeihinsa (Extract Method). (Fowler ym. 1999: 320, 322.)



Kuva 3.1. Luokan piirteiden nostaminen yliluokkaan

Refaktoroimalla voidaan eheyttää koodin rakennetta ja säilyttää sen laatu. Koodin rakenne heikkenee ajan myötä varsinkin, jos ohjelmistoon toteutetaan uusia ominaisuuksia ilman ymmärrystä koodin rakenteesta tai jos ominaisuuksia kehitetään vain täyttämään sen toiminnalliset tavoitteet ottamatta lainkaan huomioon koodin laatua. Ohjelmiston kehityksen aikana on refaktoroitava jatkuvasti, jottei koodin rakenne rappeudu. (Fowler ym. 1999: 55.)

Refaktoroinnin tuloksena koodin rakenne paranee ja koodista tulee ymmärrettävämpää sekä luettavampaa. Refaktorointi helpottaa ohjelmavirheiden löytämistä ja nopeuttaa ohjelmointia. (Fowler ym. 1999: 56–57.) Vaikka refaktoroinnista on paljon hyötyä, on myös tilanteita, joissa refaktorointia ei kannata käyttää. Esimerkiksi jos koodin rakenne puuttuu tai on erittäin huono, kannattaa koodi refaktoroinnin sijaan kirjoittaa uudelleen alusta asti. Refaktorointi lisää tuottavuutta, joten sitä ei ole syytä hylätä ajan puutteen vuoksi. Lähellä ohjelmiston kehitystyön päättymistä ei kuitenkaan kannata refaktoroida, koska tällöin saavutettu tuottavuus tulee liian myöhään. (Fowler ym. 1999: 66.)

Ennen refaktorointia toteutetaan kohteena olevan koodin toiminnan varmistavat yksikkötestit, jotta estetään uusien ohjelmointivirheiden syntyminen refaktoroinnin aikana. Yksikkötestien toteuttamiseen kannattaa panostaa ja käyttää aikaa, koska testien ajaminen kertoo, mitkä asiat koodissa toimivat ja mitkä eivät. Yksikkötestien ansiosta ohjelmavirheiden etsimiseen kuluu huomattavasti vähemmän aikaa. Yksikkötestien käytöllä varmistetaan, ettei koodin lopputulos muutu. (Fowler ym. 1999: 7, 89.)

3.2.2 Koodin heikkouksia

Koodia tarkastelemalla voi huomata monia tunnusmerkkejä, jotka kertovat huonoista ohjelmointikäytännöistä ja koodin rakenteessa olevista puutteista. Tunnusmerkkejä ovat muun muassa saman koodin toistuminen useassa paikassa (duplicate code), pitkä metodi (long method) ja suuri luokka (large class), jonka vastuulla on monta eri tehtävää. (Fowler ym. 1999: 76–78.)

Toisto (Duplication)

Toiston esiintyminen koodissa on koodin rakenteen merkittävin heikkous. Toisto tarkoittaa saman koodin rakenteen (code structure) tai lausekkeen (expression) esiintymistä kahteen tai useampaan kertaan samassa ohjelmistossa. Esimerkiksi sama lauseke voi sijaita saman luokan kahdessa eri metodissa tai yhdessä metodissa voi toistua tietty rakenne, esimerkiksi `for`-silmukka. Aliluokissa voi myös toistua yhteinen piirre, joka refaktoroinnissa siirretään yliluokkaan. (Fowler ym. 1999: 76.)

Pitkä metodi (Long Method)

Pitkä metodi tarkoittaa monta eri asiaa tekevää metodia. Metodien on oltava lyhyitä, koska mitä pitempi metodi on, sitä hankalampi se on lukea ja ymmärtää. Metodin nimen pitää olla myös mahdollisimman kuvaava, jotta ohjelmoijan ei tarvitse katsoa metodin sisältöä tietääkseen, mitä metodi tekee. Pitkien metodien sisältämät kommentit viestivät usein siitä, että kommenttien yhteydessä oleva koodi voidaan purkaa omaksi metodikseen. (Fowler ym. 1999: 76–77.)

Metodin on tehtävä vain yksi asia mahdollisimman hyvin. Metodissa on pysyttävä vain yhdessä abstraktiotasossa, mikä tarkoittaa esimerkiksi, että samassa metodissa ei saa kutsua sekä ohjelmointikielen palveluja (metodeja tai funktioita) että sovelluksen palveluja (Esimerkki 3.1). Metodissa on kutsuttava joko pelkästään ohjelmointikielen

palveluja tai sovelluksen omia palveluja (Esimerkki 3.2). (Langr 2002: Enlightened Java Style.)

```
1 function printReport()
2 {
3     printHeader();
4     printBody();
5     echo "Reported $ " . getWeek() .
6          " Total:  " . getTotal();
7 }
```

Esimerkki 3.1. Funktio, joka toimii kahdessa abstraktiotasossa

```
1 function printReport()
2 {
3     printHeader();
4     printBody();
5     printTotal();
6 }
```

Esimerkki 3.2. Yhden abstraktiotason funktio

Seuraavassa on esimerkki osasta kohdesovelluksen pitkästä noin sadan rivin mittaisesta metodista, jossa on lukuisia eri abstraktiotasoja. Esimerkki 3.3 konkretisoi hankalasti luettavan ja ymmärrettävän koodin. Tällaisesta metodista on myös vaikea etsiä ja korjata ohjelmointivirheitä. Metodissa olevat kommentit viestivät niistä kohdista, joista metodi voitaisiin jakaa toisiin metodeihin.

```
 1 public function splitUsersByLastname ( $userArray )
 2 {
 3     if ( count($userArray) > 0 )
 4     {
 5         for ($i="A"; $i != "AA"; $i++)
 6         {
 7             $alphabet[$i] = array();
 8         }
 9
10         $alphabet["Å"] = array();
11         $alphabet["Ä"] = array();
12         $alphabet["Ö"] = array();
13         $alphabet[PAGE_OTHER_CHARACTERS_TEXT] = array();
14
15         $others = 0;
16
17         foreach ( $alphabet as $key => $value )
18         {
19             // Decode unicode characters and turn them into
20             // htmlentities.
21             $key = strtoupper(htmlentities(utf8_decode($key)));
22
23             foreach ( $userArray as $user )
24             {
25                 // Extract the first character of the last name
26                 // of the user and manipulate it into comparable form
27                 // with the key.
28                 $lastname = $user->getLastname();
29                 ...
```

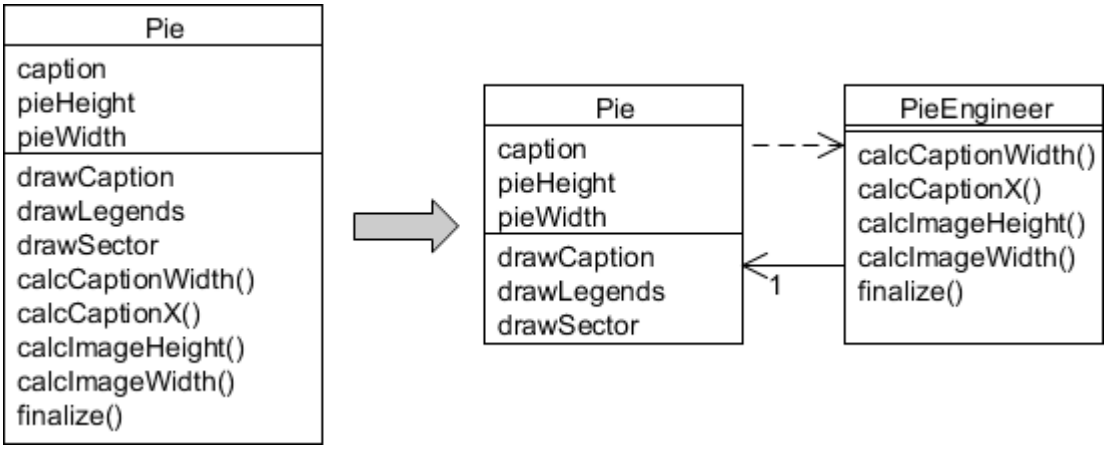
Esimerkki 3.3. Osa kohdesovelluksen noin sadan rivin mittaisesta metodista

Paikalliset metodin muuttujat saattavat aiheuttaa ongelmia ja jopa estää kokonaan metodin purkamisen toisiksi metodeiksi saman luokan sisällä. Tällaisessa tilanteessa voidaan tehdä uusi luokka, johon kyseinen metodi siirretään ja jonka tietojäseniksi asetetaan siirrettävän metodin paikalliset muuttujat (Replace Method with Method Object). (Fowler ym. 1999: 135.) Metodin siirtäminen uuteen luokkaan liittyy läheisesti luokan vastuiden siirtämiseen, jota seuraavassa käsitellään.

Suuri luokka (Large Class)

Luokan on oltava selkeä abstraktio, ja sen on vastattava vain muutamasta asiasta. Suuri luokka vastaa liian monesta tehtävästä sekä tyypillisesti sisältää paljon attribuutteja, metodeja ja koodin toistoa, mikä vaikeuttaa luokan ymmärtämistä. Tällainen luokka voidaan refaktoroida esimerkiksi purkamalla luokasta toinen luokka (Extract Class) tai aliluokka (Extract Subclass). Lisäksi luokasta voidaan poistaa koodin toistoa ja pilkkoa pitkiä metodeja useiksi lyhyiksi metodeiksi. (Fowler ym. 1999: 78, 149.)

Luokkaa purettaessa päätetään, mitkä osat luokasta siirretään toiseen luokkaan. Yhteiset etu- ja loppuliitteet attribuuttien ja metodien nimissä viestivät usein toisiinsa liittyvistä piirteistä, jotka yleensä voidaan siirtää omaan luokkaansa. (Fowler ym. 1999: 149.) Esimerkiksi (Kuva 3.2) kohdesovelluksen ympyräkaavioiden luomiseen tarkoitettu luokka Pie vastasi ennen refaktorointia sekä ympyräkaavion sisäisistä laskutoimituksista että ympyräkaavion piirtämisestä. Laskutoimituksia vastanneiden metodien nimissä oli melkein kaikilla calc-etuliite, mikä auttoi löytämään siirrettäviä metodeja. Refaktoroinnissa teimme uuden luokan PieEngineer, jolle siirsimme vastuun ympyräkaavion sisäisten laskutoimitusten laskemisesta. Pie luokalle jäi vastuu ympyräkaavion piirtämisestä.



Kuva 3.2. Pie-luokan laskutoimituksen vastuun siirtäminen PieEngineer-luokalle

Pie-luokan toiminta riippuu PieEngineer-luokasta, koska se tekee ympyräkaavion kannalta välttämättömiä laskutoimituksia. Pie-luokka ei toimi ilman PieEngineer-luokkaa. Kun Pie-luokasta luodaan olio, luodaan myös PieEngineer, jolle välitetään viite Pie-luokan olioon. PieEngineer-olioon liittyy siis aina korkeintaan yksi Pie-olio.

3.3 Yksikkötestaus

3.3.1 Yksikkötestauksen periaate

Yksikkötestauksessa on kysymys nimensä mukaisesti sovelluksen tai järjestelmän erillisten yksiköiden yksittäisestä testaamisesta (Link & Fröhlich 2002: 5). Paitsi kytkeytyen tiukasti testit ajavaan sovelluskehykseen, yksikkötestaus on myös modulaarinen ajattelutapa, joka suunnittelussa vaatii kokonaisuuksien pilkkomista pienempiin osiin.

Yksikkötestaus tehdään käyttämällä sovelluskehyksiä (unit test framework), joita on kehitetty lähes jokaiselle ohjelmointikielelle (Osherove 2009: 4). Yksikkötestaukseen käytettävä sovelluskehys on ohjelmisto, joka sisältää yksikkötestauksessa tarvittavat perusratkaisut, kuten yksikkötestiluokat ja assert-metodit (Koskimies 2000: 261). Termi yksikkö (unit) on alun perin tarkoittanut pelkästään funktiota tai proseduuria, mutta olioperustaisen ohjelmointikehityksen yhteydessä sen merkitys voi vaihdella luokasta metodiin (Link & Fröhlich 2002: 5). Kokonaisuus, jonka yksi yksikkö kattaa, on kuitenkin ohjelmoijan tulkinnan varassa (Burback 1998). On myös tärkeää huomata, ettei yksiköksi voida mieltää automaattisesti jokaista luokan metodia, vaan yksikkö voi myös olla julkinen rajapinta useaan yksityiseen metodiin. Shore ja Warden summaavat, että yksikkötestien ajaminen on nopeaa, ja mikäli näin ei ole, testit eivät ole yksikkötestejä lainkaan (Shore & Warder 2007: 298). He myös ohjeistavat, että kaikkien testien ajamiseen saisi maksimissaan kulua kymmenen sekuntia.

Vaikka yksikkötestausta varten on olemassa lukuisia eri sovelluskehyksiä monilla eri ohjelmointikielillä, ovat yksikkötestauksen perusperiaatteet samat. Yksikkötestauksessa testin varsinaisena ideana on luoda testausasetelma, jossa yksikköä voidaan käyttää, ja verrata lopputulosta odotettuun lopputulokseen. Testausasetelma (fixture) on Kent Beckin Simple SmallTalk Testing: With Patterns -dokumentin mukaan vapaasti suomennettuna ”yksittäinen konfiguraatio, jonka käytös ei ole arvaamatonta”. Testausasetelmalla testataan yksittäisen testitapauksen (test case) onnistuminen (check). Testisarjalla (test suite) testataan useita testitapauksia. (Beck 2009.)

Yksikkötestin varsinainen suoritus tapahtuu seuraavia askeleita noudattaen:

1. testausasetelman luominen (set up)
2. väittämien tutkiminen (assert)
3. testausasetelman sulkeminen (tear down)

Testausasetelman luomisessa luodaan maailma, jossa testi toimii. Maailma voi tarkoittaa esimerkiksi olion luomista testattavasta luokasta ja sen tilan asettamista halutuksi. Useimmiten olion tilan muutokset tapahtuvat kuitenkin ennen väittämien tutkimista, koska yksi yksikkötesti testaa aina vain yhtä luokan toimintoa. Testin jälkeen maailma suljetaan esimerkiksi vapauttamalla varatut resurssit. Alaluvussa 3.4.1 käydään läpi esimerkki yksikkötestin ajamisesta.

Keskeinen asia testien ajamisessa on, että testausasetelma luodaan ja suljetaan jokaisen testin kohdalla. Tämä mahdollistaa testattavan olion tilan muuttamisen testitapauksen mukaan, ja takaa sen, että testitapaukset ovat toisistaan riippumattomia testausasetelman ja testidatan suhteen.

Yksikkötestit ovat myös työkalu ohjelmakoodin bugien korjaamisessa. Kun sovelluksessa havaitaan bugi, kirjoitetaan uusi yksikkötesti, joka epäonnistuu bugin takia. Tämän jälkeen bugia voidaan lähteä korjaamaan, ja kun testi onnistuu, voidaan myös bugi todeta korjatuksi (Shore & Warden 2007: 163).

3.3.2 Yksikkötestauksen työkalut

xUnit

Kent Beck toteutti yhdessä Erich Gamman kanssa Java-ohjelmointikielen yksikkötestaukseen tarkoitetun JUnit-sovelluskehyksen (Clark 2006). JUnit oli jatkumoa Beckin Smalltalk-kielellä toteutettuihin sovelluskehyksiin, jotka noudattelivat hänen Test-driven development: by example -kirjassaan hahmoteltuja periaatteita ja käytäntöjä automatisoitua yksikkötestausta varten. JUnit saavutti suurta suosiota ja tunnettavuutta ohjelmistokehittäjien keskuudessa. Lisäksi se loi pohjan sovelluskehysten käyttämisestä yksikkötestauksessa ja levitti myös tietoa ketterien menetelmien ja testilähtöisen ohjelmistokehityksen käytöstä (Fowler 2009). JUnitista on sittemmin käännetty yksikkötestaussovelluskehys miltei jokaiselle ohjelmointikielelle, ja näitä sovelluskehyksiä kutsutaan yhteisellä nimellä xUnit-sovelluskehykset (Fowler 2009).

Test doubles

Yksikkötestauksessa tulee eteen tilanteita, joissa testaaminen on vaikeaa, koska testauksen alainen yksikkö riippuu komponenteista, joita ei voida käyttää testausympäristössä, tai koska yksikön testaaminen vaatii enemmän sen toiminnan hallintaa kuin mitä testausasetelman luomisessa voidaan tavallisesti saavuttaa. Tällaisissa tilanteissa, joissa yksikön toimintaan liittyvää komponenttia ei voida tai haluta käyttää, se voidaan korvata testissä sijaisella (test double) (Meszaros 2008). Käytännössä tämä tarkoittaa sitä, että testattavan yksikön käyttöön tarjotaan jokin komponentti, joka sisältää yksikön käyttämät palvelut, mutta jonka toiminnan ohjelmoija on etukäteen säätänyt. Termi test double tulee elokuvaamisesta, jossa vaarallisessa tai vaikeassa kohtauksessa elokuvan päätähden tilalle tulee sijaisnäyttelijä (stunt double). Test doubles -suunnittelumalleihin lukeutuvat muun muassa stub ja mock.

Stub eli vapaasti suomennettuna tynkä on malli, jossa testattavan yksikön kohdalla yksikön toimintaan vaikuttava komponentti (useimmiten olion metodin logiikka) korvataan ”tyngällä” niin, ettei se sisällä mitään muuta kuin valmiiksi konfiguroidun palautusarvon. Näin testauksen alla oleva yksikkö voidaan pakottaa valintoihin, joihin se ei muuten välttämättä ajautuisi (Meszaros 2008).

Mock eli vapaasti suomennettuna jäljitelmä on malli, jossa voidaan paitsi konfiguroida palautusarvot kuten stub-mallissa, mutta myös tarkastella yksikön suorituksen epäsuoria tuloksia, kuten kutsuttuja metodeja, niiden lukumäärää ja järjestystä (Meszaros 2008).

Mock- ja stub-olioita käytetään korvaamaan testattavan yksikön ne osat, jotka kommunikoivat testausasetelman ulkopuolisen maailman kanssa (Shore & Warder 2007: 298). Teknisesti mockin ja stubin välillä ei ole välttämättä lainkaan eroja, mutta niiden ero on tavassa, jolla niitä käytetään. Siinä missä mock-oliot nojaavat aina yksikön käyttäytymiseen, voidaan stub-mallissa tehdä myös näin tai sitten vain pidättäytyä tietojen välittämiseen testattavalle yksikölle (Fowler 2007).

3.4 PHPUnit

PHPUnit on Sebastian Bergmanin kehittämä xUnit-sovelluskehysperheeseen kuuluva ilmainen ja avoimen lähdekoodin yksikkötestaustyökalu PHP-sovelluksille. PHPUnit on lisensoitu BSD-lisenssillä (Bergmann 2009a), ja se on portattu vuonna 2002 jul-

kaistusta JUnit 3.8.1:stä PHP-sovellukseksi. PHPUnitin ominaisuuksiin kuuluvat muun muassa mock-olioiden (mock object) tuki, tietokannan testaamisen tuki, keskeneräiset testit, testien ohittaminen ja yksikkötestauskoodin ketterä kommentointi. (Bergmann 2009c.)

PHPUnitia voi käyttää suoraan komentoriviltä tai sovelluskehitysympäristöstä, esimerkiksi Eclipsestä. Sen voi myös asettaa suorittamaan ennen yksikkötestien ajamista halutun PHP-tiedoston eli niin sanotun bootstrap-tiedoston (Bergmann 2009b: 21). Tätä voidaan käyttää esimerkiksi testien ajamisessa tarvittavien luokkatiedostojen lataamiseen. Seuraavaksi tarkastellaan String-luokan ja sitä testaavan StringTest-luokan toteutusta ja ajamista.

3.4.1 Esimerkki yksikkötestin ajamisesta

Tässä esimerkissä tutustutaan yksikkötestin rakenteeseen ja ajamiseen PHPUnitilla. Esimerkissä on kaksi luokkaa: sovellusluokka String, jota testataan, ja testiluokka StringTest, joka testaa String-luokkaa. String-luokka (Esimerkki 3.4) sisältää metodin merge, joka yhdistää kaksi parametrina saamaansa merkkijonoa toisiinsa ja palauttaa yhdistetyn merkkijonon.

```
1 <?php
2 class String
3 {
4     public function merge($a, $b)
5     {
6         return $a . $b;
7     }
8 }
9 ?>
```

Esimerkki 3.4. Testattava String-luokka

StringTest-luokka (Esimerkki 3.5) testaa String-luokan merge-metodin toimintaa. Rivillä 2 tuodaan StringTest-luokkaan testaamiseen tarvittava PHPUnit_Framework_TestCase-luokka. Rivillä 3 tuodaan testaamisen kohde eli String-luokka StringTest-luokkaan, jotta String-luokasta voidaan luoda olio ja kutsua sen merge-metodia.

```
 1 <?php
 2 require_once 'PHPUnit/Framework.php';
 3 require_once 'String.php';
 4
 5 class StringTest extends PHPUnit_Framework_TestCase
 6 {
 7     protected $object;
 8
 9     protected function setUp()
10     {
11         $this->object = new String;
12     }
13
14     protected function tearDown()
15     {
16         unset($this->object);
17     }
18
19     public function testMerge()
20     {
21         $this->assertEquals("PHPUnit",
22                             $this->object->merge("PHP", "Unit"));
23     }
24 }
25 ?>
```

Esimerkki 3.5. String-luokkaa testaava StringTest-luokka

Esimerkissä StringTest-luokasta rivillä 5 määritellään itse luokka, joka johdetaan luokasta PHPUnit_Framework_TestCase. Tällä periytymisellä luokan käyttöön saadaan kaikki yksikkötestauksen työkalut, kuten setUp-, tearDown- ja assert-metodit. Suojattu ilmentymämuuttuja `$object` esitellään rivillä 7. Tähän muuttujaan luodaan olio testattavana olevasta luokasta, eli tässä tapauksessa String-luokasta. Olio luodaan setUp-metodissa rivillä 11. Metodi setUp (rivit 9–12) suoritetaan aina ennen jokaisen testimetodin suorittamista. Sen tarkoituksena on sisältää esimerkiksi jokaisessa testissä tarvittavien olioiden luominen ja olioiden tietokenttien alustaminen tietyillä arvoilla, jotta niitä ei tarvitsisi kirjoittaa jokaiseen testimetodiin. Metodi tearDown (rivit 14–17) suoritetaan aina jokaisen testimetodin suorittamisen jälkeen, ja se on tarkoitettu esimerkiksi testissä luotujen olioiden tuhoamiseen, kuten tässä esimerkissä on tehty rivillä 16. (Bergmann 2009b: 15.)

Itse yksikkötesti testMerge määritellään riveillä 19–23. PHPUnit ajaa automaattisesti test-sanalla alkavat metodit (Bergmann 2009b: 122). Metodissa testMerge testataan merge-metodin toimintaa assertEquals-metodilla, joka ottaa vastaan kaksi erityypistä arvoa. Metodi assertEquals testaa kahden annetun parametrin sisällön samansisältöisyyden tai samanarvoisuuden. Parametrit voivat olla erityyppisiä, esimerkiksi merkkijonoja, taulukkoja (array) tai lukuarvoja. Jos parametrien sisältö ei ole sama, testi epäonnistuu, muussa tapauksessa testi onnistuu. (Bergmann 2009b: 94.)

PHPUnit tarjoaa myös joukon muita assert-metodeja, esimerkiksi

- assertNull testaa, onko annettu parametri null
- assertFalse testaa, onko annettu parametri false
- assertTrue testaa, onko annettu parametri true
- assertNotEquals testaa, eroaako kahden annetun parametrin sisältö.

String-luokan testaaminen onnistuu komennolla `phpunit StringTest`. PHPUnit palauttaa testistä alla olevan tuloksen (Esimerkki 3.6). Tuloksessa näkyvä yksi piste tarkoittaa yhtä suoritettua testiä (Bergmann 2009b: 11).

```
PHPUnit 3.3.17 by Sebastian Bergmann.

.

Time: 0 seconds

OK (1 test, 1 assertion)
```

Esimerkki 3.6. String-luokan testaamisen tulos

Tuloksessa voi esiintyä pisteen lisäksi myös muita merkkejä, joiden merkitykset selviävät seuraavasta taulukosta (Taulukko 3.1).

Merkki	Selite
F	Testin väittämä epäonnistui
E	Testissä tapahtui ohjelmavirhe
S	Testi ohitettiin
I	Testi on keskeneräinen eikä sitä suoritettu

Taulukko 3.1. Tuloksissa esiintyvien merkkien selitykset (Bergmann 2009b: 11)

3.4.2 Testiluokan ja sovellusluokan generointi

PHPUnit osaa generoida testiluokkien runkoja sovelluksen luokista, mikä helpottaa testien toteuttamista. PHPUnit luo vastaavat test-sanalla alkavat metodien rungot sovelluksen julkisista metodeista. Jos sovelluksen luokassa on metodi merge, tekee PHPUnit metodin testMerge, jonka sisällöksi tulee ilmoitus, että testi on keskeneräinen. Sen lisäksi PHPUnit tekee tearDown-metodin rungon ja setUp-metodin, jossa luodaan testattavasta luokasta olio. (Bergmann 2009b: 65.)

PHPUnitille voi antaa sovellusluokan metodin kommentissa PHPUnitin ymmärtämiä ohjeita siitä, miten PHPUnitin on testiluokkaa generoitaessa tehtävä assert-metodien kutsut testiluokan metodeihin. Nämä ohjeet määritetään `@assert`-tageilla (Esimerkki 3.7). Rivillä 5 kerrotaan PHPUnitille, että testMerge-metodissa on tutkittava assert-metodilla, palauttaako merge-metodi parametreilla PHP ja Unit arvon PHPUnit. PHPUnit tekee automaattisesti myös StringTest-luokan, johon se sijoittaa testMerge-metodin. StringTest-luokka voidaan generoida komennolla `phpunit --skeleton-test String`. Tuloksena saadaan alaluvussa 3.4.1 esitelty StringTest-luokka. (Bergmann 2009b: 65.)

```
 1 <?php
 2 class String
 3 {
 4     /**
 5      * @assert (‘PHP’, ‘Unit’) == ‘PHPUnit’
 6      */
 7     public function merge($a, $b)
 8     {
 9         return $a . $b;
10     }
11 }
12 ?>
```

Esimerkki 3.7. String-luokan **`@assert`**-tagit

Sovelluksen testilähtöisessä ohjelmistokehyksessä voidaan käyttää apuna PHPUnitin ominaisuutta, jossa testiluokasta voidaan tuottaa varsinaisen sovellusluokan runko. PHPUnit analysoi, mitä sovellusluokan metodeja testiluokka kutsuu ja kirjoittaa ne sovellusluokan runkoon. (Bergmann 2009b: 66.)

3.4.3 Koodin kattavuuden analysointi

Koodin kattavuuden analysointi kertoo, mitä sovelluksen koodirivejä suoritetaan ja mitkä jäävät suorittamatta testejä ajettaessa. Kattavuuden analysoinnilla voidaan havaita testattavan sovelluksen suorittamattomat ja siten myös testaamattomat koodirivit. Näin saadaan arvio siitä, kuinka hyvin testit testaavat sovellusta. PHPUnit tuottaa koodin kattavuuden analyysin käyttämällä apuna PHP:hen erikseen asennettavaa, ilmaista ja avoimen lähdekoodin Xdebug-laajennusta (extension). (Bergmann 2009b: 56.)

Tarkastellaan seuraavaksi esimerkkiä (Esimerkki 3.8), jossa Divider-luokkaa testaava testiluokka DividerTest ei testaa jakoa nollalla. Metodi testMerge testaa vain kahden osamäärän eli lukujen 2 ja 4 oikeellisuuden.

```
 1 class DividerTest extends PHPUnit_Framework_TestCase
 2 {
 3     protected $divider;
 4
 5     protected function setUp()
 6     {
 7         $this->divider = new Divider;
 8     }
 9
10     protected function tearDown()
11     {
12     }
13
14     public function testMerge()
15     {
16         $this->assertEquals(2, $this->divider->divide(14, 7));
17         $this->assertEquals(4, $this->divider->divide(8, 2));
18     }
19 }
```

Esimerkki 3.8. Divider-luokkaa testaava DividerTest-luokka

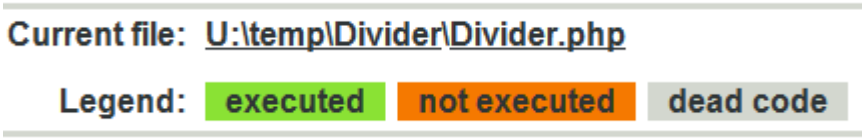
Analysoidaan seuraavaksi, kuinka hyvin DividerTest-luokan testit kattavat Divider-luokan koodin. PHPUnit tuottaa analyysin html-sivuina haluttuun hakemistoon komennolla `phpunit --coverage-html .\report DividerTest`, jossa `.\report` on hakemisto, johon analyysi sijoitetaan ja DividerTest on ajettava testi. (Bergmann 2009b: 56.) Tuloksena saadaan seuraavanlainen analyysi (Esimerkki 3.9).

```
 1                 : <?php
 2                 : class Divider
 3                 : {
 4                 :     public function divide($x, $y)
 5                 :     {
 6                 1 :         if ( $y == 0 )
 7                 1 :         {
 8                 0 :             throw new Exception();
 9                 :             echo $y;
10                 :         }
11                 :
12                 1 :         $result = $x / $y;
13                 :
14                 1 :         return $result;
15                 :         echo $result;
16                 :     }
17                 : }
18                 : ?>
```

Esimerkki 3.9. Esimerkki koodin kattavuudesta

Esimerkissä on katkelma PHPUnitin ja Xdebugin tuottamasta koodin kattavuuden analyysistä, josta ilmenee, ettei `if`-lauseen sisälle mennä testejä ajettaessa koskaan. Vasemmanpuoleiset luvut ovat rivinnumeroita, ja kaksoispisteen vasemmalla puolella

oleva luku kertoo, kuinka moni testimetodi on suorittanut kyseisen rivin. Suorittamatta jääneet rivit ovat oransseja ja suoritetut rivit vihreitä. Saavuttamaton koodi (dead code) väritetään harmaalla (Kuva 3.3).



Kuva 3.3. Kuvakaappaus ohjelmasta, josta ilmenee värien merkitykset

3.5 Testilähtöinen ohjelmistokehitys

3.5.1 Ketterät menetelmät

Testilähtöinen ohjelmistokehitys on eräs Extreme Programming -menetelmien useista käytännöistä. Extreme Programming (XP) on puolestaan yksi monista ketteristä (Agile) menetelmistä (Martin 2006). XP-menetelmä korostaa tiimityöskentelyä, viestintää ohjelmoijien ja asiakkaan välillä, yksinkertaisuutta ja laadukkuutta sovelluksen toteutuksessa sekä testilähtöistä ohjelmistonkehitystä (Wells 2009a).

Testilähtöisyys on tärkeä osa XP-menetelmää, sillä sen avulla varmistutaan koodin toimivuudesta ja torjutaan vanhojen ohjelmavirheiden uudelleen esiintyminen. XP-menetelmässä asiakkaan muuttuviin vaatimuksiin pystytään vastaamaan nopeasti (Wells 2009b).

Ketterien menetelmien yhteiset periaatteet esiteltiin vuonna 2001 julkaistussa Ketterässä manifestissa. Ketterä manifesti on seitsemäntoista ohjelmistokehittäjän yhdessä synnyttämä lausuma, jossa määritellään ketterän ohjelmistokehityksen perusideat. Manifestin tarkoitus on levittää ketterien sovelluskehitysmenetelmien käyttöä ja ”ketterää ajatustapaa”, sekä vastustaa ohjelmistotuotannon yleistä mielikuvaa, jossa kehitystyö on jäykkää sekä tiukasti sidottu sopimuksiin, suunnitelmiin ja dokumentteihin (Beck, Beedle, Bennekum ym. 2001).

3.5.2 Testilähtöisen ohjelmistokehityksen periaatteet

Testilähtöinen ohjelmistokehitys (test-driven development, TDD) on eräs ketteriin menetelmiin lukeutuva lähestymistapa sovelluskehitykseen. Testilähtöisen ohjelmistokehityksen idean toi suuren yleisön tietoon Kent Beck (Ambler 2009).

Testilähtöisen ohjelmistokehityksen ideana on yksikkötestien tekeminen ennen varsinaista ohjelmakoodia. Beck määrittelee testilähtöisen ohjelmistokehityksen kehityssykliksi seuraavat askeleet (Beck 2003: 1):

1. Lisää testi.
2. Aja kaikki testit ja varmista, että uusi testi epäonnistuu.
3. Kirjoita ohjelmakoodi uudelle testille.
4. Aja testit uudelleen ja varmistu, että uusi testi onnistuu.
5. Refaktoroi.

Nämä viisi askelta tiivistyvät kahdeksi testilähtöisen ohjelmistokehityksen perussäännöksi (Beck 2003: xix):

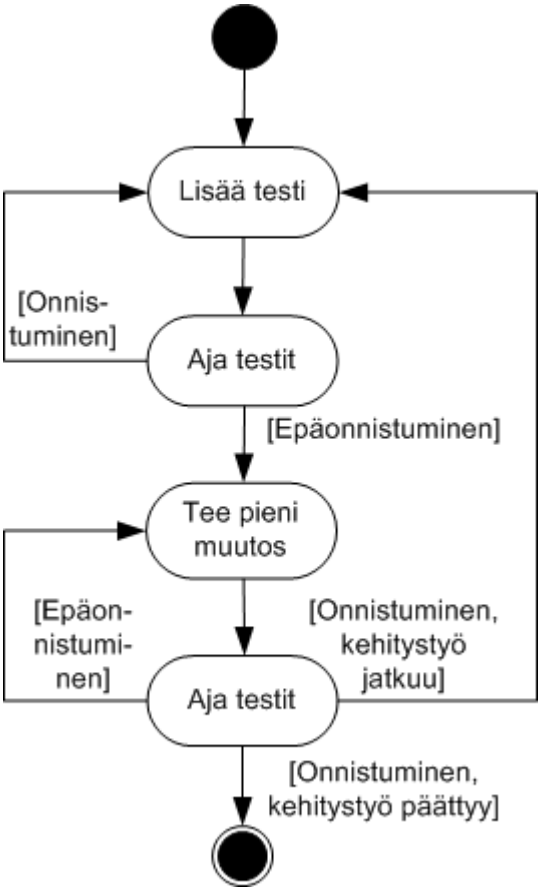
1. Sovelluskoodiin kajotaan silloin ja vain silloin, kun sitä vastaava yksikkötesti on epäonnistunut.
2. Kaikki toisto koodissa tulee poistaa.

Ensimmäinen perussääntö kuvaa testilähtöisen kehityksen luonteen: uutta ohjelmakoodia kirjoitetaan vain ja ainoastaan, kun testi tai testit epäonnistuvat. Tämän säännön ulkopuolelle jää luonnollisesti koodin refaktorointi, joka edellyttää onnistuvia yksikkötestejä. Kuten kehityssyklin askelissa kuvattiin, uuden ominaisuuden kehitys alkaa testin lisäämisellä kyseiselle ominaisuudelle, mutta tämä koskee myös koodin rakenteen parantamista: mikäli sovellus yksikkötestien lisäämisen, ominaisuuden koodaamisen ja refaktoroinnin jälkeen toimii ja myös testit onnistuvat, ei koodia ole tarpeen enää muuttaa. (Beck 2003: 11.)

Käytännössä testien epäonnistuminen voi tarkoittaa myös sitä, ettei sovellus edes käänny, jolloin yksikkötestaussovelluskehyskään ei mahdollisesti voi suorittaa yksikkötestejä loppuun. Näin voi käydä esimerkiksi tilanteessa, jossa sovellukseen ollaan lisäämässä uusi luokka, ja yksikkötestit sekä sovelluksen kääntäminen epäonnistuvat kehityssyklin kohdassa 2 koska kyseisestä luokasta yritetään luoda olio.

Testien luonne ja viesti myös muuttuu, kun ne suoritetaan testilähtöisen ohjelmistokehityksen mukaisesti ensin testi, sitten koodi – järjestyksessä. Testin epäonnistuminen ei tarkoita tässä menetelmässä sitä, että sovellus sisältäisi virheitä, vaan testaussovelluskehyksen antama virheilmoitus merkitsee joko sitä että ominaisuutta, jota testi koskee, ei ole vielä toteutettu tai koodin rakenteen muokkaaminen muutti koodin sisäistä logiikkaa. Testilähtöisessä ohjelmistokehityksessä virheilmoitus epäonnistuneesta testistä ei ole niinkään negatiivinen asia kuin ensikädeltä olettaisi, vaan paremminkin merkki edistyksestä uuden ominaisuuden kehityksen kohdalla tai ilmoitus refaktoroinnin epäonnistumisesta eli koodin logiikan muuttumisesta. (Ambler 2009.)

Testilähtöisen ohjelmistokehityksen askeleet voidaan kuvata seuraavalla vuokaaviolla (Kuva 3.4):



Kuva 3.4. Testilähtöisen ohjelmistokehitykseen askeleet (mukaillen Ambler 2009)

Edellä kuvattu kehityssykli tuo hyvin esiin testilähtöisen ohjelmistokehityksen pääperiaatteet (Beck 2003: 1):

- Jokainen yksikkötesti kattaa vain pienen osan toiminnallisuutta.
- Ohjelmakoodia muokataan usein.
- Testit ajetaan usein.

Ohjelmakoodin muokkaaminen ja parantaminen tehdään pienissä osissa, ja on tärkeää huomata, että menetelmän luonteeseen kuuluu paitsi koko kehityssyklin toistaminen projektin edistymiseksi myös yksittäisten kohtien toistaminen syklin sisällä. Ohjelmakoodi, joka kirjoitetaan uuden testin läpäisyä varten voi ja saakin olla usein nopeasti kirjoitettu eikä välttämättä toimi virheettömästi tai optimaalisesti. Muokkaamalla koodia ja ajamalla testit jokaisen muutoksen välissä uudelleen voidaan olla varmoja, että ohjelmakoodi toimii edelleen samalla kun koodin rakenne paranee. Jos taas koodin muutoksen jälkeen testi epäonnistuu, nähdään että koodin rakenteen muokkauksessa epäonnistuttiin, ja tehdyt muutokset voidaan palauttaa. Koodin rakenteen parantamista ja testien ajamista jatketaan, kunnes koodi on selkeää eikä sisällä toistoa. (Beck 2003: 4.)

Vaikka yksi yksikkötestien sovelluskehyksen tavoitteista onkin, että kukin yksikkö voidaan testata erillisenä ilman muita yksiköitä, ajetaan kehityssyklin jokaisessa vaiheessa kaikki testit aina, kun sovellukseen tehdään muutoksia. Näin huomataan välittömästi, mikäli uusi testi tai koodi aiheuttaa sovelluksen jonkin toisen testin epäonnistumisen ja siten aiheuttaisi myös testiä vastaavan koodin rikkoontumisen. (Shore & Warden 2007: 288.)

Yksikkötestien kattavuus (test coverage) on myös tärkeä seikka testilähtöisessä ohjelmistokehityksessä, sillä yksikkötestauksen tavoite on olla niin kattava, että jokainen sovelluksen yksikkö tulee testatuksi (Ambler 2009). Luonnollisesti lähtökohta, jossa testit kirjoitetaan ennen sovelluskoodia, enemmän tai vähemmän takaa tämän onnistumisen, mutta sovellettuna testilähtöistä ohjelmistokehitystä jo olemassa olevaan sovelluskoodiin, testien kattavuus auttaa hahmottamaan testattuja yksiköitä kokonaisuudessaan.

Testilähtöinen ohjelmistokehitys tuo mukanaan paljon etuja. Testien tekeminen ennen varsinaista ohjelmakoodia pakottaa ohjelmoijan suunnittelemaan työn alla olevan ominaisuuden tarkasti, jolloin ohjelmakoodi vaatii kokonaisuudessaan vähemmän muutoksia. Tällöin myös sovellus muotoutuu modulaarisemmaksi. Testien ajaminen myös antaa rehellistä palautetta sovelluksen toiminnasta – testien onnistuessa voidaan olla varmoja, että ohjelmakoodi toimii niin kuin yksikkötestit sitä testaavat. (Shore & Warden 2007: 301.)

Testilähtöisen ohjelmistokehityksen sovelluskehitysprosessissa toimiva ohjelmakoodi takaa myös toimivan kokonaisuuden. Kun ohjelmakoodin rakenteen muokkaaminen on saatu päätökseen ja testit onnistuvat, on ohjelmakoodi valmis. Nämä virstanpylväät osoittavat havainnollisesti ja totuudenmukaisesti projektin etenemisen tahdin. (Beck 2003: ix.)

Vaikka monet sovelluskehittäjät kokevat testilähtöisen ohjelmistokehityksen elegantiksi tavaksi ohjelmoida, on tekniikkaa myös kritisoitu. Yksikkötestin, joissakin tapauksissa useamman, kirjoittaminen jokaista varsinaisen ohjelmakoodin yksikköä vastaan teettää ohjelmoijalla luonnollisesti enemmän työtä kuin pelkkä ohjelmakoodi yksinään. Testilähtöinen ohjelmistokehitys myös pakottaa suunnittelun etenemään pienissä askelissa. Tämän omaksuminen voi olla vaikeaa ohjelmoijille, jotka ovat tottuneet hallitsemaan kokonaisuutta laaja-alaisesti. Usein testilähtöistä ohjelmistokehitystä ovatkin kritisoineet eniten vanhoihin työskentelytapoihin tiukasti piintyneet ohjelmoijat, joilla ei ole vielä vahvaa osaamista työskentelytavoista ketterissä menetelmissä. Testilähtöisen ohjelmistokehityksen edut ovat kuitenkin luonteeltaan epäsuorempia: lopputuotteen korkea laatu, pienempi virheiden etsimiseen kulunut aika, tunne jatkuvasta työn edistymisestä sekä ”turvaverkko”, joka kertoo erehtymättömästi, mikäli refaktorointi on muuttanut koodin logiikkaa. Vaakakupissa nämä hyödyt painavatkin useimmilla ohjelmoijilla enemmän kuin näennäinen lisätyö sekä suunnittelutyön osittaminen pieniksi palasiksi. (Ambler 2009)

3.5.3 Esimerkki testilähtöisestä ohjelmistokehityksestä PHPUnitilla

Tässä esimerkissä käydään läpi luokan ja metodin luominen testilähtöisen ohjelmistokehityksen periaatteiden mukaisesti PHPUnit-työkalulla. Esimerkin aiheena on luoda luokka String, joka sisältää metodin merge. Metodin tarkoituksena on yhdistää kaksi parametrina annettavaa merkkijonoa yhdeksi merkkijonoksi ja palauttaa yhdistetty merkkijono. Työ alkaa testin kirjoittamisella uudelle ominaisuudelle (Esimerkki 3.10):

```
 1 class StringTest extends PHPUnit_Framework_TestCase
 2 {
 3     protected $object;
 4
 5     public function setUp()
 6     {
 7         $this->object = new String();
 8     }
 9
10     public function testMerge()
11     {
12         $this->assertEquals("PHPUnit",
13                 $this->object->merge("PHP", "Unit"));
14     }
15 }
```

Esimerkki 3.10. Yksikkötesti uutta ominaisuutta varten

Testin kirjoittamisen jälkeen se ajetaan, joka ennakoidusti aiheuttaa virheilmoituksen testin kääntämisessä. Tämä johtuu luonnollisesti siitä, että setUp-metodissa String-Test-luokan `$object`-tietojäsenen alustaminen rivillä 7 epäonnistuu, koska kyseistä luokkaa ei ole vielä toteutettu. Nyt ollaan siis siinä testilähtöisen ohjelmistokehityksen kehityssyklin kohdassa, jossa luotu testi halutaan saada onnistumaan mahdollisimman vähällä työllä ja mahdollisimman nopeasti.

String-luokan esimerkissä rivillä 1 (Esimerkki 3.11) määritellään itse luokka. Luokka sisältää metodin merge, joka määritellään rivillä 3. Metodille määritellään kaksi parametria, `$a` ja `$b`. Rivillä 5 on metodin palautuskäsky, joka palauttaa metodia kutsuttaessa merkkijonon ”PHPUnit”.

```
1 class String
2 {
3     public function merge($a, $b)
4     {
5         return "PHPUnit";
6     }
7 }
```

Esimerkki 3.11. String-luokan merge-metodi

String-luokan toteuttaminen ylläkuvatulla tavalla saa testMerge-testin onnistumaan, jolloin testattavaa yksikköä voidaan alkaa parantaa. Nykyisellään merge-metodi palauttaa aina kutsuttaessa merkkijonon "PHPUnit", mikä tietenkään ei ole metodin tavoitteena. Lähdettäessä poistamaan tämänkaltaisia kovakoodattuja eli koodiin kirjoitettuja tilanteesta riippumattomia kohtia voidaan testilähtöisessä ohjelmistokehityksessä käyttää triangulation-tekniikkaa, jossa nimensä mukaisesti jäljitellään ajatustasolla kolmiomittausta niin, että kolmion ensimmäinen sivu on ensimmäinen testi, toinen si-

vu testattava yksikkö ja kolmanneksi sivuksi lisätään uusi testi erilaisilla arvoilla, jolloin koko yksikkötestiin tulee vaihtelua. (Beck 2003: 16.)

Esimerkissä 3.12 testMerge-metodiin riville 14 lisätyssä assertEquals-tutkimuksessa vertaillaan samalla tutulla tavalla odotettua ja saatavaa lopputulosta; tässä tapauksessa odotettu merkkijono on “Framework” ja todellinen saatava lopputulos on String-luokasta luodun `$object`-jäsenmuuttujan merge-metodin palautusarvo parametreina merkkijonot ”Frame” ja ”work”.

```
10     public function testMerge()
11     {
12         $this->assertEquals("PHPUnit",
13                $this->object->merge("PHP", "Unit"));
14         $this->assertEquals("Framework",
15                $this->object->merge("Frame", "work"));
16     }
```

Esimerkki 3.12. Yksikkötestin laajentaminen

Uuden assert-väittämän lisääminen testMerge-yksikkötestiin aiheuttaa testin epäonnistumisen rivillä 14, jossa odotettu merkkijono "Framework" ei vastaa merge-metodin palauttamaa "PHPUnit"-merkkijonoa. Tämä pakottaa muuttamaan merge-metodia (Esimerkki 3.13) niin, että se käyttää parametreina annettuja argumentteja. Tällöin testin onnistuminen on jälleen välitön tavoite, johon päästään testattavaa yksikköä nopeasti muuttamalla.

```
3     public function merge($a, $b)
4     {
5         $returnValue = $a . $b;
6         return $returnValue;
7     }
```

Esimerkki 3.13. Yksikön korjaaminen

Metodia merge muutettiin niin, että metodin parametrien `$a` ja `$b` yhdistämisen lopputulos sijoitettiin tilapäiseen `$returnValue`-muuttujaan rivillä 5, jonka metodi palauttaa rivillä 6.

Muutoksen jälkeen testit onnistuvat. Ominaisuus on toiminnallisuudeltaan valmis, mutta testilähtöisen ohjelmistokehityksen viimeisessä vaiheessa, koodin rakenteen parantamisessa, koodattua ominaisuutta täytyy tarkastella vielä kriittisesti ja parannusmahdollisuuksia etsien. Esimerkkitoteutuksessa (Esimerkki 3.13) silmään osuu välit-

tömästi koodissa esiintyvä toisto: muuttuja `$returnValue` ilmenee metodissa riveillä 5 ja 6. Toistoa voidaan lähteä turvallisesti poistamaan (Esimerkki 3.14), sillä testit ilmoittavat virheestä ja toimivat turvaverkkona tilanteessa, jossa refaktorointi on muuttanut koodin logiikkaa.

```
3     public function merge($a, $b)
4     {
5         return $a . $b;
6     }
```

Esimerkki 3.14. Valmis yksikkö

Metodia refaktoroitiin niin, että parametrit `$a` ja `$b` yhdistetään ja palautetaan samalla rivillä 5. Koodin refaktoroinnin jälkeen ajetaan testit, mikä paljastaa, että koodin logiikka ei ole muuttunut refaktoroinnissa. Näin testilähtöisen ohjelmistokehityksen kehityssykli on päättynyt yhden ominaisuuden osalta.

4 Yksikkötestauksen käyttöönotto

Koska kohdesovellusta ei alun perin kehitetty testilähtöisesti eikä sen yksiköitä ajateltu niiden kehityksen aikana testattaviksi, kohdesovellus ei sellaisenaan soveltunut yksikkötestaamiseen. Tämän vuoksi kohdesovellukseen tehtiin paljon muutoksia. Tässä luvussa kuvataan muun muassa, mistä tilanteesta yksikkötestien suunnittelu ja toteutus aloitettiin, mitä muutoksia kohdesovellukseen tehtiin ennen kuin yksikkötestejä voitiin alkaa kirjoittaa ja miten yksikkötestit organisoitiin hakemistohierarkiaan. Luvun lopussa esitetään käyttöönoton lopputulos eli mitä yksikköjä testattiin, mitä refaktorointeja tehtiin sekä mitä virheitä ja ongelmia korjattiin.

4.1 Lähtötilanne

Opinnäytetyön työosuuden aloittamishetkellä syyskuussa 2009 kohdesovelluksessa ei ollut lainkaan yksikkötestejä, ja sen koodi oli eri ohjelmointitavoilla toteutettua. Vanhimmissa PHP-tiedostoissa ei ollut lainkaan funktioita tai korkeintaan muutama erittäin pitkä funktio. Myöskään sovelluslogiikkaa, käyttöliittymän muodostamista ja SQL-lauseita ei ollut erotettu toisistaan. Tällaisia PHP-tiedostoja oli kuitenkin suhteellisen vähän. Paljon suurempi osa sovelluksen koodista oli proseduraalisesti toteutettua, kuten liikunta- ja käyttäjätilastot, sivujen näyttämisestä vastaava `index.php` sekä rekisteröityminen ja kirjautuminen. Suurimmaksi osaksi sovellus oli kuitenkin olioperustaisesti toteutettu niin, että käyttöliittymä, sovelluslogiikka ja SQL-lauseet oli erotettu toisistaan omiksi luokikseen. Sovelluksen koodin rakenteen taso oli vaihtelevaa. Sovelluksesta löytyi paljon pitkiä hankalasti testattavia ja refaktoroimista vaativia funktioita, metodeja ja luokkia.

4.1.1 Yleinen toiminta

Sovelluksen toiminnan avaimena on juuritiedosto `index.php`, joka noutaa osoiterivillä (GET-taulukossa) välitetyn sivun tunnistenumeron perusteella halutun sivun tietokannasta. Tiedosto `index.php` vastaa sivun näyttämisen lisäksi myös siitä, että käyttäjä ei pääse tarkastelemaan sellaista sisältöä, johon kyseisellä käyttäjällä ei ole oikeutta. Kirjautumis- ja hakusivua lukuun ottamatta sivujen lähdekoodi haetaan tietokannasta. Pelkästään staattista eli muuttumatonta sisältöä sisältävien sivujen lähdekoodi on suoraan tietokannassa. Sen sijaan dynaamisissa eli toimintalogiikkaa sisältävissä sivuissa sisällytetään PHP:n require_once-funktiolla erillinen `main.php`-tiedosto suoritettavan

sivun lähdekoodiin. Tiedosto `index.php` lopulta suorittaa PHP-kielisen sivun koodin PHP:n eval-funktiolla.

Moduulien käynnistystiedostoina toimivat `main.php`-tiedostot mahdollistavat käyttäjän pääsyn tiettyyn ominaisuuteen. Niiden tehtävänä on luoda ominaisuuteen liittyvästä ohjain-luokasta (Controller) olio ja kutsua sen selectAction-metodia, jossa valitaan tietty toiminto tai suoritetaan oletustoiminto, ellei mitään toimintoa ole määritelty. Usein oletustoiminto on pelkästään pääsivun näyttäminen. Esimerkiksi liikuntasuoritukset-ominaisuudesta vastaavan Exercise-moduulin `main.php` näyttää oletusarvoisesti liikuntasuoritukset-ominaisuuden pääsivun, josta liikuntasuorituksia voi tarkastella ja lisätä. Tämänkaltaisia tiedostoja voi olla myös useita samassa moduulissa, esimerkiksi Exercise-moduulissa on lisäksi `goalMain.php`-tiedosto, jonka kautta viikkotavoitteen voi asettaa ja muuttaa.

Myös PHP:llä muodostetut kuvat, PDF-tiedostot ja muut erityyppiset resurssit välitetään selaimeen erillisten php-tiedostojen kautta. Esimerkiksi Gallup-moduulin `gallupPie.php` palauttaa ajettaessa mielipideympyräkaavion PNG-kuvan, ja Pass-moduulin `download.php` tuottaa PDF-muotoisen passitodistuksen, jonka käyttäjä voi ladata selaimellaan.

Sovellus tukee myös monikielisyyttä, ja opinnäytetyön kirjoitushetkellä se tuki suomea ja ruotsia. Monikielisyys oli toteutettu siten, että kaikki sovelluksen staattiset tietokannasta noudettavat sivut löytyivät tietokannasta sekä suomeksi että ruotsiksi. Käyttöliittymän muodostamisen sisältävissä moduuleissa ladattiin tarpeelliset tekstit sekä suomen- että ruotsinkielisistä vakiotiedostoista. Vakiotiedostot oli ryhmitelty kielihakemistoihin, ja tämän lisäksi oli määritelty kieliriippumattomia vakioita, joilla kohdesovelluksen ominaisuuksia oli mahdollista konfiguroida.

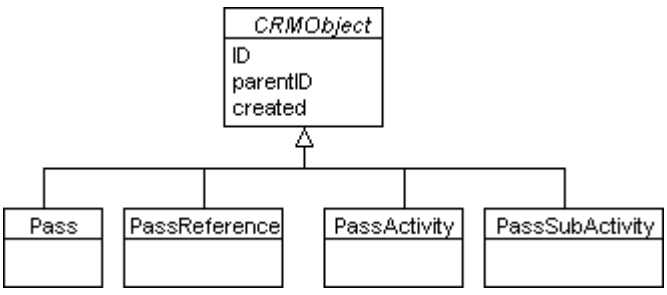
4.1.2 Olioperustaisuus

Sovelluksen toiminnallisuudet on jaettu kautta linjan hieman eri tavalla toteutettuihin moduuleihin. Moduuli kohdesovelluksessa tarkoittaa yksittäiseen ominaisuuteen liittyvien luokkien ja muiden tiedostojen kokonaisuutta, joka on yhdessä hakemistossa. Tyypillisesti moduulissa esiintyy edellisessä alaluvussa esiteltyjen moduulin käynnistystiedostojen ja resursseja palauttavien tiedostojen lisäksi sovelluslogiikasta vastaavat Controller-luokat, käyttöliittymästä vastaavat View-luokat, tiedon tallentamiseen käy-

tetyt malliluokat ja DBObject-luokat, jotka suorittavat tietokantaan kyselyitä ja muuttavat kyselyiden tulokset malliluokan olioiksi. Controller-luokat eli ohjaintason luokat on toteutettu niin, että ne kutsuvat yhden tai useamman View-luokan metodeja, jotka palauttavat tiettyjä osia käyttöliittymästä, esimerkiksi kokonaisen www-sivulla olevan lomakkeen. View-luokat eivät kutsu Controller-luokan metodeja. Mallitason luokkien yliluokkana toimii abstrakti CRMObject-luokka, joka sisältää olion tunnisteen (ID), olioon liittyvän toisen olion tunnisteen (parentID) ja olion luontihetken (created).

Tietokantayhteyksien yliluokka DBObject sisältää metodit kyselyiden sekä lisäys- ja päivityslauseiden suorittamista varten. Tietokantalauseet suoritetaan DB-luokasta luodun olion avulla, ja itse tietokantayhteys avataan DBConnection-luokan avulla, jossa yhteys on toteutettu Singleton-tekniikalla niin, että ainoastaan yksi yhteys on kulloinkin auki. Abstraktiin DBObject-luokkaan liittyy aina yksi mallitason luokka, esimerkiksi liikuntasuoritukset-ominaisuuteen liittyvässä Exercise-moduulissa luokkaan ExerciseDBObject liittyy luokka Exercise. DBObject-luokalla on yksi abstrakti metodi toObjectArray, jonka konkreettisten aliluokkien on toteutettava. ExerciseDBObject-luokka toteuttaa toObjectArray-metodin niin, että se palauttaa taulukossa (array) tietokantakyselyn tuloksena saatavia Exercise-olioita.

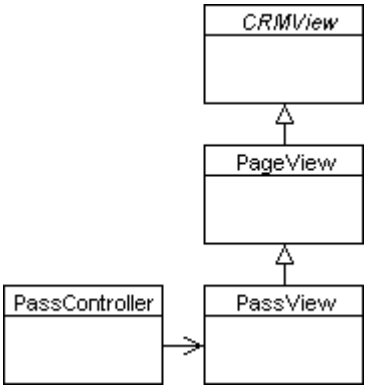
Tarkastellaan seuraavaksi Pass-moduulia (Kuva 4.1), joka vastaa työkykypassitodistusten myöntämiseen liittyvästä kohdesovelluksen ominaisuudesta. Pass-moduulin malliluokat Pass, PassReference, PassActivity ja PassSubActivity periytyvät abstraktista CRMObject-luokasta.



Kuva 4.1. Pass-moduulin malliluokkien periytyminen CRMObject-luokasta

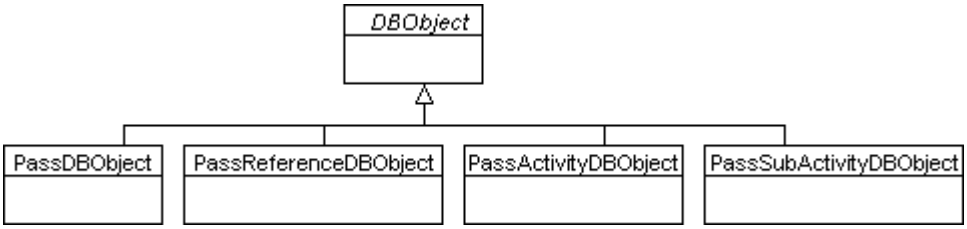
Pass-moduulin käyttöliittymä sijaitsee PassView-luokassa, joka periytyy PageView-luokasta, joka puolestaan sisältää kaikkien näkymäluokkien yleisiä käyttöliittymän muodostamiseen liittyviä metodeja. PageView-luokka periytyy taas abstraktista

CRMView-luokasta. Seuraavasta kuvasta (Kuva 4.2) ilmenee PassView-luokan periytymishierarkia ja PassController- ja PassView-luokan välinen yhdensuuntainen suhde: PassController kutsuu PassView-luokan metodeja, jotka palauttavat osia käyttöliittymästä. PassView ei koskaan kutsu PassControllerin metodeja.



Kuva 4.2. View-luokat ja Controllerin yhdensuuntainen linkki PassView-luokkaan

Jokaiseen Pass-moduulin mallityypin luokkaan liittyy abstraktista DBObject-luokasta periytyvä luokka. Esimerkiksi (Kuva 4.3) Pass-luokkaan liittyy luokka PassDBObject, jonka vastuulla on Pass-olioiden vieminen tietokantaan sekä tietokannassa olevan tiedon noutaminen ja muuttaminen Pass-olioiksi. Seuraava kuva (Kuva 4.3) esittää DBObject-luokkien periytymishierarkian.



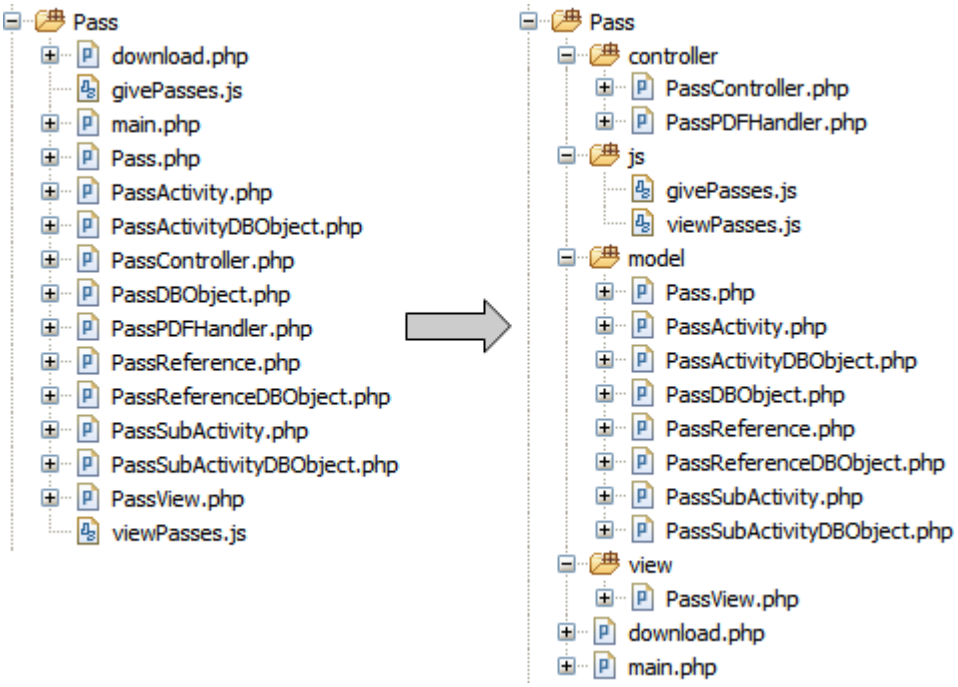
Kuva 4.3. Pass-moduulin DBObject-luokkien periytyminen DBObject-luokasta

4.2 Testausympäristön luominen

4.2.1 Muutokset sovellukseen

Moduulihakemiston juuressa oli enimmillään jopa noin 20 kappaletta erityyppisiä tiedostoja, kuten PHP-tiedostoja, PHP-luokkatiedostoja ja JavaScript-tiedostoja, mikä hankaloitti yksittäisen tiedoston paikantamista. Hyvän hakemistohierarkian mukaisesti luotiin kohdesovelluksen moduuleihin dataa, logiikkaa ja näkymää vastaavat model, controller ja view hakemistot, joihin siirrettiin niitä vastaavat luokat. Tämän lisäksi

luotiin oma hakemisto JavaScript-tiedostoille. Tällaisella menettelyllä moduulin juureen jäivät ainoastaan tiedostot, joiden tehtävänä on käynnistää moduulin varsinaisia toimintoja tai hakea ladattavia resursseja, kuten PDF-tiedostoja ja PNG-kuvia. Esimerkiksi Pass-moduulissa hakemistorakenne muutettiin seuraavasti (Kuva 4.4):



Kuva 4.4. Pass-moduulin hakemistorakenteen uudistaminen

Jotta yksikkötesteissä voidaan luoda testattavista luokista olioita, on kyseisen luokan lähdekoodin oltava ladattuna muistiin. Kohdesovellusta ajettaessa luokkien lähdekoodit ladataan automaattisesti muistiin aina, kun luokasta luodaan olio. Tämä on toteutettu PHP:n __autoload-funktiolla. Luokkien lisäksi kohdesovelluksessa käytetään paikka paikoin `functions.php`-tiedoston funktioita, jotka tietenkin myös on oltava ladattuna muistiin ennen kuin niitä voidaan kutsua.

Tiedosto `functions.php` ladattiin lähes kaikissa kohdesovelluksen PHP-tiedostoissa erikseen. Tätä tiedostoa ei ladattu luokissa, koska luokkia käytetään proseduraalisten tiedostojen kautta, jolloin tiedoston funktiot näkyivät myös luokille. Tämän tiedoston lisäksi kohdesovelluksessa ladattiin aina `common.php`, kun käytettiin sovelluksen muita luokkia tai funktioita. Tiedosto `common.php` asettaa yleisiä koko sovelluksen laajuisia asetuksia ja jota ilman sovellus ei käynnisty.

Koska `functions.php` ladattiin käytännössä aina, kun `common.php` ladattiin, siirrettiin `functions.php`-tiedoston lataaminen `common.php`-tiedostoon, mikä vähensi sovelluksen koodin toistoa. Esimerkiksi Pass-moduulin `main.php`-tiedostossa ladattiin molemmat mainitut tiedostot, mutta muutoksen jälkeen vain `common.php`.

Kohdesovelluksella oli kaksi erilaista toimintaympäristöä: tuotantoympäristö ja kehitysympäristö. Tuotantoympäristöä käyttävät loppukäyttäjät ja kehitysympäristö on tarkoitettu kehittäjien testaus- ja kehitysympäristöksi. Ympäristöillä on eri PHP-, WWW-, MySQL-palvelimet, mikä vaatii sovellukselta ympäristökohtaisia asetuksia, kuten tietokannan tunnukset ja kohdesovelluksen ajohakemisto. Kohdesovellus yksikkötestattiin komentoriviltä, joten komentorivi lisättiin kohdesovelluksen uudeksi toimintaympäristöksi.

Koska yksikkötestit eivät saa olla yhteydessä tietokantaan tai käyttää tiedostoja estettiin nämä asiat, kun kohdesovellus ajetaan komentoriviltä. Kohdesovellus lähettää tarvittaessa myös sähköposteja, esimerkiksi rekisteröitymisen hyväksymisestä tai hylkäämisestä. Yksikkötestit eivät saa myöskään olla yhteydessä verkon yli, joten `functions.php`-tiedostossa ollut sähköpostin lähetys siirrettiin uuteen Email-luokkaan. (Shore & Warden 2007: 298.) Email-luokasta voitiin muodostaa PHPUnitilla mock-olio, jota käytettiin Email-luokan sijaan yksikkötestauksessa, jotta sähköpostien lähetys saatiin estettyä.

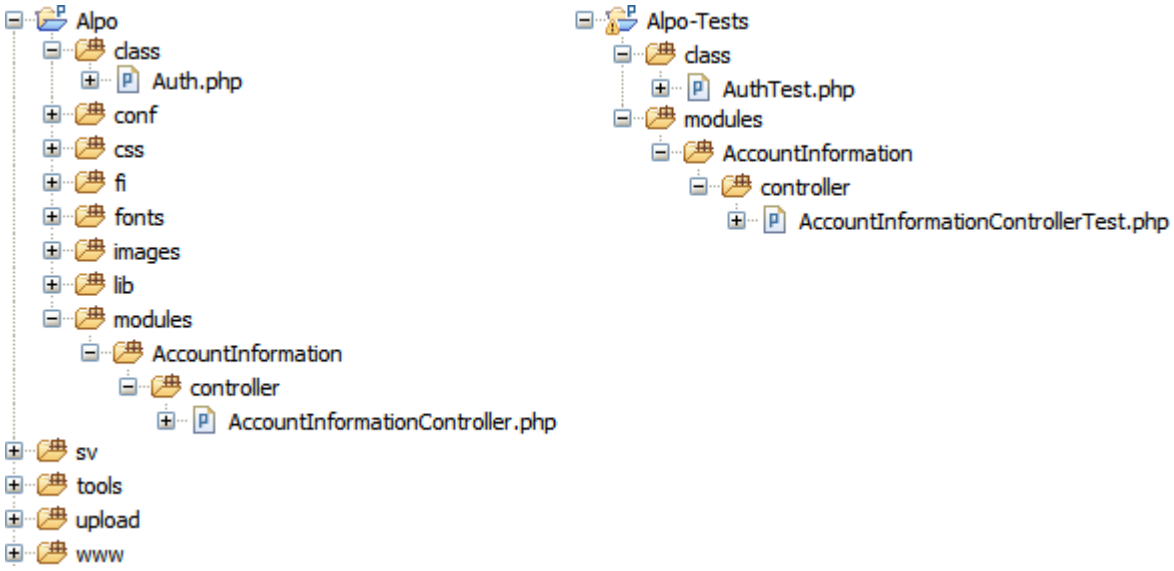
4.2.2 Testien organisointi

Yksikkötestit voidaan sijoittaa testattavan koodin yhteyteen (inline tests), mikä tarkoittaa, että sovelluksen luokkaan toteutetaan myös luokkaa testaava testiluokka. Tällä tavalla testit ovat helpommin ja nopeammin saatavilla, mutta tässä tavassa on kuitenkin huonoja puolia. Jos yksikkötestauskoodi on jostain syystä poistettava, esimerkiksi asiakkaalle toimitettaessa, yksikkötestauskoodin joutuu poistamaan käsin. Tämä tapa voi myös heikentää sovelluksen suorituskykyä. (Schlossnagle 2004: 114.)

Toinen, suositeltavampi tapa on sijoittaa testiluokat omiin tiedostoihinsa (outline tests). (Schlossnagle 2004: 114-115.) Yksikkötestien organisoinnissa päätetään myös yksikkötestien sijoittaminen hakemistohierarkiaan. Sebastian Bergmannin mukaan testit on helpointa sijoittaa omaan hakemistoon, jonka alihakemistot peilaavat testatta-

van sovelluksen hakemistorakennetta. Yksikkötestit sijoitettiin kohdesovellukseen Bergmanin esimerkin mukaisesti. (Bergmann 2009b: 19.)

Seuraavassa on esimerkki Auth-luokkaa sekä AccountInformation-moduulin AccountInformationController-ohjainluokkaa testaavien tiedostojen sijoittamisesta hakemistohierarkiaan. Kuvan 4.5 vasemmanpuoleisessa hakemistossa on sovellushakemisto ja oikeanpuoleisessa hakemistossa testit sisältävä hakemisto.



Kuva 4.5. Esimerkki testiluokkien sijoittamisesta hakemistohierarkiaan

Testihakemistot peilattiin ja luotiin vain niistä sovelluksen hakemistoista, jotka sisälsivät testattavia PHP-luokkatiedostoja. Esimerkiksi css ja fonts hakemistoja ei luotu testihakemistoon.

4.2.3 Testausympäristön rajaaminen

Testauksen piiriin tulisi ottaa mukaan kaikki koodi, joka saattaa rikkoontua, poikkeuksena koodi, joka ei sisällä lainkaan logiikkaa, kuten luokan yksityisen jäsenmuuttujan tilaa tutkivat tai muuttavat metodit, sekä kolmannen osapuolen koodit, ellei ole erityistä syytä epäillä niiden toimivan virheellisesti (Shore & Warden 2007: 301). Tämän nojalla jätettiin kohdesovelluksessa mallitason luokat testauksen ulkopuolelle, sillä nämä luokat sisälsivät ainoastaan tietojäsenensä sekä metodit tietojäsenten arvojen asettamiseksi ja palauttamiseksi. Kohdesovelluksen kolmannen osapuolen koodia ei näin ollen myöskään testattu. Logiikkaa sisältämättömiin koodeihin lukeutuivat myös moduuleiden `main.php`-tiedostot, jotka sisälsivät vain main-funktion moduulin käyn-

nistämiseksi. Kohdesovelluksen yksikkötestauksen ulkopuolelle jätettiin seuraavat sovelluksen osat:

- sovelluksen vanhimmat koodit, jotka eivät sisällä yksiköitä
- kolmannen osapuolen koodit
- proseduraalisesti toteutetut tiedostot
- moduuleiden tietokantakyselyistä vastaavat DBObject-luokat, jotka eivät sisällä juurikaan logiikkaa
- kuvia tai muita resursseja tuottavat tiedostot
- `main.php`-tiedostot
- Challenge-moduuli; tämä moduuli ei ollut tuotantoympäristössä aktiivisena käytössä, eikä näin ollen vaatinut testaamista
- funktiot tai metodit, jossa kutsutaan PHP:n omia funktioita, esimerkiksi Email-luokan sendMail-metodia (käyttää mail-funktiota) ja Finals-luokan requireOnce-metodia (käyttää require_once-funktiota)

4.3 Yksikkötestien suunnittelu ja toteutus

4.3.1 Yksikkötestien suunnittelu

Yksikkötestien toteuttaminen olemassa olevalle koodille edellyttää, että koodin rakenne on hyvien ohjelmointitapojen mukaisesti toteutettua. Jos koodin rakenne on huono, yksikkötestien kirjoittaminen on vaikeaa. Koodia on siten tarvittaessa refaktoroitava. (Shore & Warden 2007: 298, 300.)

Yksikkötestien suunnittelu ja toteutus aloitettiin yleisesti käytetyistä class-hakemiston luokista. Kun class-hakemiston luokat oli testattu, testattiin tärkeimpiä olioperustaisesti toteutettuja moduuleita.

Yksikkötestiluokkien suunnittelussa käytettiin apuna PHPUnitin ominaisuutta, joka luo valmiista luokasta kyseistä luokkaa vastaavan testiluokan rungon. Generoitu testiluokka sisälsi testit pelkästään julkisia metodeja varten, mikä helpotti testattavien julkisten metodien selvittämistä. Jos testiluokassa oli vain muutama metodi, vaikka testattava sovellusluokka sisälsi useita kymmeniä metodeja, viittasi tämä siihen, että testattavassa sovellusluokassa oli paljon refaktorointia vaativia yksityisiä metodeja.

Ennen yksikkötestiluokan varsinaista toteuttamista tarkistettiin, mitä refaktorointeja sovellus tarvitsi, jotta yksikkötestejä voitiin kirjoittaa. Tarvittaessa purettiin suuria sovellusluokkia useiksi luokiksi ja pitkiä metodeja joko muihin luokkiin tai samaan luokkaan useiksi metodeiksi. Mahdollisen refaktoroinnin jälkeen kirjoitettiin sovelluksen yksiköille testit. Yksikkötestin kirjoittamisen jälkeen yksikköä vielä tarvittaessa refaktoroitiin. Nyt refaktoroinnissa voitiin käyttää apuna luotua testiä, joka varmisti refaktoroinnin onnistumisen.

Yksikön toiminnassa oliot, jotka olivat yhteydessä testausasetelman ulkopuolisiin resursseihin, korvattiin PHPUnit-sovelluskehyksen avulla luoduilla mock-olioilla. PHPUnit korvaa sovelluksen olion metodit stub-mallia käyttäen tyhjällä toteutuksella. Mock-olioiden metodit konfiguroitiin tarvittaessa palauttamaan testidataa.

Kohdesovellus sisältää useita luokkia, joiden metodit ovat staattisia, ja nämä luokat ovat vastuussa toiminnoista, joita käytetään kaikkialla sovelluksessa, kuten kirjautuneena olevan käyttäjän tietojen tai valittuna olevan kielen noutaminen sessiosta tai osoiteriviltä. Koska staattisia metodeja kutsutaan luomatta oliota itse luokasta, ei tällaisissa tilanteissa voida käyttää test doubles -mallin ratkaisuja. Yksikön luonteesta riippuen voidaan käsin asettaa testidataa esimerkiksi juuri GET-taulukkoon, jossa osoiterivin argumentit sijaitsevat, ja näin varmistua siitä, että yksikkö käyttää staattisia kutsuja oikein. Mikäli staattisten metodien palautusarvo on yhdentekevä yksikön lopputuloksen kannalta, voidaan samaa staattista metodia kutsua myös yksikön testissä. Esimerkiksi kohdesovelluksen TeacherToolsView-luokan getJsDefinitions-metodissa kutsutaan Finals-luokan staattista getJavaScriptFinals-metodia. Metodia getJsDefinitions testaavaan yksikkötestiin kirjoitettiin myös getJavaScriptFinals-metodin kutsu, jotta ei oteta kantaa siihen, mitä Finals-luokan metodi palauttaa. Näin Finals-luokkaa ei myöskään testata kyseisessä yksikkötestissä.

Testien nimeämisessä käytettiin yleistä tapaa, jossa testin nimi muodostetaan testattavan yksikön nimestä test-alkuliitteellä. Esimerkiksi Pass-luokan deletePass-metodia testaavalle yksikkötestille annettiin nimi testDeletePass. Jos yksikkötesti jaettiin kahdeksi tai useammaksi testiksi näiden nimien loppuun lisättiin mahdollisimman informatiivinen kuvaus tilanteesta, jota kyseinen testi testaa. Esimerkiksi Pass-luokan

getPassTableData-metodi toimii hieman eri tavalla riippuen siitä, onko sen käsittelemä työkykypassi myönnetty käyttäjälle, joka on kohdesovelluksen käyttäjä vai ei-rekisteröitynyt. Tämän vuoksi yksikkötestit nimettiin seuraavasti:

- testGetPassTableDataWhenStudentIsRegistered
- testGetPassTableDataWhenStudentIsUnregistered.

PHPUnitilla voi tulostaa yksikkötestien ajon tulokset myös testdox-muodossa (Esimerkki 4.1), mikä antaa erittäin informatiivista tietoa. PHPUnit muodostaa yksikkötestin nimestä lauseen, jonka eteen se merkitsee ruksin, jos kyseinen yksikkötesti onnistuu.

```
PassController
 [x] Get pass table data when student is registered
 [x] Get pass table data when student is unregistered
```

Esimerkki 4.1. PassController-luokan testaustulos testdox-formaatissa

PHPUnit-sovelluskehyksen tarjoama setUp-metodi toteutettiin kulloisenkin testattavan luokan tarpeiden mukaan. Metodissa saatettiin luoda olio testattavasta luokasta sekä mock-olioita korvaamaan esimerkiksi tietokantayhteyksiä käyttäviä DBObject-olioita. Mock-oliot asetettiin testattavan luokan jäsenmuuttujiksi set-metodeilla, jotka toteutettiin, mikäli niitä ei testattavassa luokassa entuudestaan ollut.

Sebastian Bergmannin mukaan tearDown-metodia ei ole välttämätön toteuttaa, jos setUp-metodissa ei allokoida ulkoisia resursseja (Bergmann 2009b: 16), mikä ei ole yksikkötestauksessa sallittuakaan. Näin ollen tearDown-metodin toteutukset testiluokissa voitiin jättää tyhjäksi.

4.3.2 Esimerkki yksikkötestin toteutuksesta

Tarkastellaan seuraavaksi, miten kohdesovelluksen SystemUserController-luokan käyttäjän hylkäämisestä vastaava decline-metodi yksikkötestattiin. Metodi hylkää tunnisteen (userID) perusteella tietyn käyttäjän. Hylkäyksessä asetetaan käyttäjä tietokannassa hylätyksi, ja hylkääjän tunniste, joka saadaan Auth-luokan metodilta getSessionUserID, asetetaan myös käyttäjän tietoihin. Kohdesovelluksen staattisen Auth-luokan metodi getSessionUserID lukee PHP:n `$_SESSION`-taulusta kirjautuneen käyttäjän tunnisteen.

Tietokannassa olevien tietojen muuttamiseen käytetään riveillä 3-4 SystemUserDBObject-luokasta luotua oliota userDBO (Esimerkki 4.2), jolla on metodit setActive ja setActivatedBy. Metodi setActive muuttaa tietyn käyttäjän aktiivisuustason ja metodi setActivatedBy muuttaa tiedon siitä, kuka muutti kyseisen käyttäjän aktiivisuustasoa.

```
1 public function decline($userID)
2 {
3     $this->userDBO->setActive($userID, USER_ACCOUNT_DECLINED);
4     $this->userDBO->setActivatedBy($userID, Auth::getSessionUserID());
5 }
```

Esimerkki 4.2. SystemUserController-luokan decline-metodi hylkää tietyn käyttäjän

Koska decline-metodissa käytetään tietokannan kanssa keskustelevaa luokkaa (SystemUserDBObject), ei tätä luokkaa voida suoraan käyttää metodia yksikkötestattaessa. Sen sijaan käytetään mock-olioita. SystemUserControllerTest-luokan setUp-metodi (Esimerkki 4.3) luo testattavasta luokasta olion, alustaa testiluokan ja testattavan luokan tietojäseniä sekä luo yhteensä kaksi mock-oliota SystemUserDBObject- ja ViewingRightsController-luokista. Mock-olion avulla voidaan testata, kutsutaanko mock-olion tiettyjä metodeja oikeilla parametreilla, sekä voidaan myös asettaa mock-olion metodit palauttamaan tiettyjä arvoja (Bergmann 2009b: 41–42.) Alla olevan metodin rivillä 6 ja 7 asetetaan testattava luokka käyttämään mock-olioita aitojen sovelluslusluokkien sijaan.

```
1 protected function setUp()
2 {
3     $this->object         = new SystemUserController;
4     $this->mockUserDBO    = $this->getMock("SystemUserDBObject");
5     $this->mockRightsCtrl = $this->getMock("ViewingRightsController");
6     $this->object->setUserDBO($this->mockUserDBO);
7     $this->object->setRightsCtrl($this->mockRightsCtrl);
8 }
```

Esimerkki 4.3. SystemUserControllerTest-luokan setUp-metodi

Rivillä 3 (Esimerkki 4.4) asetetaan PHP:n `$_SESSION`-tauluun käyttäjän tunnisteeksi luku 2, jotta rivillä 10 voidaan testata, että mockUserDBO-olion metodia setActivatedBy kutsutaan arvolla 2. Riveillä 5–7 konfiguroidaan mockUserDBO-olio odottamaan setActive-metodin kutsua täsmälleen yhden kerran parametrilla 1 ja vakion USER_ACCOUT_DECLINED arvolla. Samaan tapaan rivillä 8–10 konfiguroidaan mockUserDBO-olio odottamaan setActivatedBy-metodin kutsua täsmälleen yhden kerran luvulla 1 ja 2. (Bergmann 2009b: 41–42.) Metodissa testDecline esiintyvä luku

1 tarkoittaa hylättävän käyttäjän tunnistetta. Rivillä 11 kutsutaan testattavan luokan decline-metodia. Metodi testDecline testaa, käyttäytyykö decline-metodi oikein, jos hylättävän käyttäjän tunniste on 1 ja hylkäävän käyttäjän tunniste on 2.

```
 1 public function testDecline()
 2 {
 3     $_SESSION['USER_ID_IN_SESSION'] = 2;
 4
 5     $this->mockUserDBO->expects($this->once())
 6                       ->method("setActive")
 7                       ->with($this->equalTo(1, USER_ACCOUNT_DECLINED));
 8     $this->mockUserDBO->expects($this->once())
 9                       ->method("setActivatedBy")
10                       ->with($this->equalTo(1, 2));
11     $this->object->decline(1);
12 }
```

Esimerkki 4.4. SystemUserControllerTest-luokan testDecline-metodi

4.3.3 Yksikkötestien ajaminen

Kohdesovellus vaatii toimiakseen tiettyjen luokkien, funktioiden ja asetusten lataamista muistiin. Tähän kohdesovellus käyttää `common.php`-tiedostoa, joka ladataan heti sovelluksen käynnistymisen jälkeen, esimerkiksi `main.php`-tiedostossa `common.php` ladataan ennen varsinaisen muun koodin suorittamista.

Yksikkötestien ajamiseen käytettiin PHPUnitin versiota 3.3.17, PHP:n versiota 5.3 ja Xdebugin versiota 2.0.5. Yksikkötestit ajettiin komentoriviltä Windows XP –ympäristössä, johon oli asennettu WAMP-sovellus. Ajettaessa kohdesovellusta yksikkötestien kautta komentoriviltä pitää `common.php` ladata muistiin. Tähän käytettiin PHPUnitin ominaisuutta, jossa voi määrittää bootstrap-tiedoston parametrina. Jotta parametria ei tarvinnut kirjoittaa joka kerta yksikkötestejä ajettaessa, tehtiin komentojonotiedosto `_phpunit.bat`, joka kutsui itse phpunit-sovellusta bootstrap-parametrilla ja muilla käyttäjän mahdollisesti antamilla parametreilla. Komentojonotiedostoon lisättiin myös nykyisen päivämäärän ja ajan tulostaminen, koska PHPUnit ei tulosta niitä.

Koska testihakemisto organisoitiin omaan hakemistoonsa, joka peilaa testattavia sovellushakemistoja, voitiin testejä ajaa halutuilta sovelluksen osilta. Esimerkiksi jos nykyinen hakemisto sisältää hakemiston modules, kaikki sen moduulit voidaan testata komennolla `_phpunit modules`. Yksittäinen moduuli, esimerkiksi Exercise-moduuli, voidaan ajaa komennolla `_phpunit modules\Exercise`. Koko kohdesovelluksen yk-

sikkötestien ajaminen onnistuu testihakemiston sisältävässä hakemistossa komennolla `_phpunit tests`, joka antaa seuraavan tuloksen (Kuva 4.6).

```
---------------------------------
ke 28.10.2009 10:29:32,70
---------------------------------
PHPUnit 3.3.17 by Sebastian Bergmann.

.............................................................   60 / 1222
.............................................................  120 / 1222
.............................................................  180 / 1222
.............................................................  240 / 1222
.............................................................  300 / 1222
.............................................................  360 / 1222
.............................................................  420 / 1222
.............................................................  480 / 1222
.............................................................  540 / 1222
.............................................................  600 / 1222
.............................................................  660 / 1222
.............................................................  720 / 1222
.............................................................  780 / 1222
.............................................................  840 / 1222
.............................................................  900 / 1222
.............................................................  960 / 1222
............................................................. 1020 / 1222
............................................................. 1080 / 1222
............................................................. 1140 / 1222
............................................................. 1200 / 1222
......................

Time: 14 seconds

OK (1222 tests, 4795 assertions)
```

Kuva 4.6. Kohdesovelluksen kaikkien yksikkötestien ajon tulos

4.4 Lopputulos

4.4.1 Testatut yksiköt

Kohdesovelluksen moduulien luokkakirjastona toimiva class-hakemisto testattiin miltei kokonaisuudessaan. Testiluokkia kirjoitettiin yhteensä 21 kappaletta. Testauksen ulkopuolelle jätettiin mallitason luokat, jotka eivät sisällä logiikkaa, kuten diagrammeihin liittyvä Legend-luokka ja abstraktit CRM-yliluokat, sekä ulkoisia resursseja käyttävät luokat, kuten tietokantayhteyttä käyttävä ModelNameDBObject sekä kuvaresursseja manipuloiva luokka ImageController. Myös kaikki tietokantayhteyksien perusluokat sekä poikkeusluokat jätettiin testaamatta. Kohdesovelluksen moduuleista testattiin seuraavat:

- AccountInformation
- Activity
- Blog

- EducationInformation
- Exercise
- GroupMail
- Pass
- TeacherTools
- User
- ViewingRights.

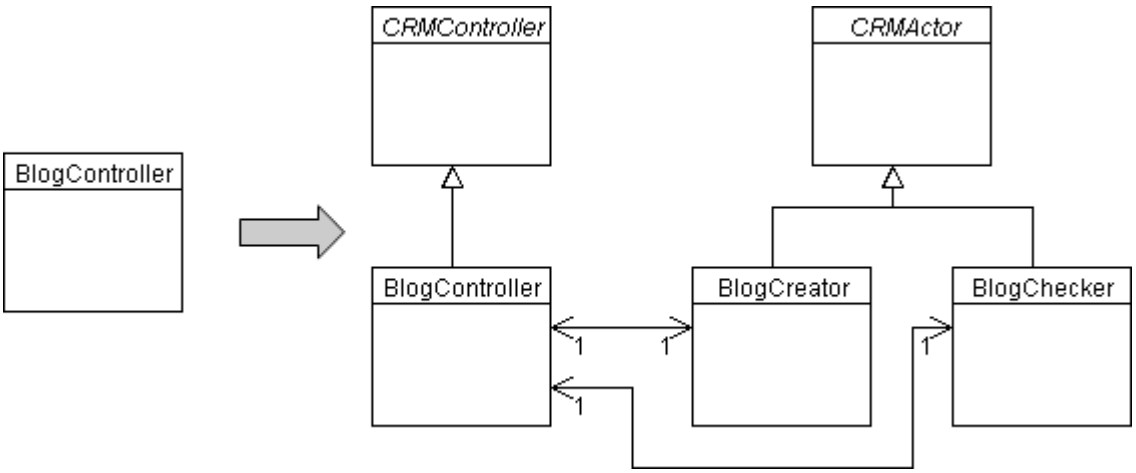
Nämä moduulit vastaavat kohdesovelluksen käyttötarkoitusta silmällä pitäen tärkeimmistä ja oleellisimmista ominaisuuksista, kuten käyttäjien rekisteröitymisestä palveluun, käyttäjien tiedottamisesta sekä ammattiosaajan työkykypassin suorittamisesta ja myöntämisestä.

4.4.2 Tehdyt refaktoroinnit

Yksikkötestien toteuttamisessa sovelluskoodia jouduttiin refaktoroimaan joissakin tilanteissa paljonkin. Tyypillisimpiä refaktorointeja olivat pitkän metodin pilkkominen useammaksi metodiksi sekä suuren luokan vastuiden jakaminen muille luokille. Esimerkki pitkästä metodista on PassView-luokan getViewingList-metodi, joka alussa oli 120 rivin mittainen ja joka refaktoroitiin viideksi 20–30 rivin mittaiseksi metodiksi. Tämänkaltainen muutos helpotti merkittävästi yksikkötestien kirjoittamista.

Testattujen moduulien ohjainluokkien yhteiseksi yliluokaksi luotiin CRMController-luokka. Tähän luokkaan voitiin asettaa ilmentymämuuttujia, jotka toistuivat useimmissa ohjainluokissa. Tällaisia muuttujia olivat esimerkiksi muuttujat näkymäluokan oliota varten sekä suoritettavasta toiminnasta kertova tunniste. Ohjainluokkien vastuuta myös jaettiin luomalla uusia ohjaintason luokkia, ja näiden luokkien yliluokaksi luotiin CRMActor-luokka. Tästä luokasta johdettujen luokkien vastuina ovat mallitason olioiden tietojen tarkistaminen ja olioiden luominen POST- ja GET-datasta. CRMActor-luokkaan toteutettiin yksityinen ilmentymämuuttuja, joka viittaa moduulin varsinaiseen CRMController-luokasta periytyvään olioon ja jonka palveluita, esimerkiksi tietokantakyselyitä, tästä luokasta periytyvä olio voi käyttää. Esimerkki tästä refaktoroinnista on Blog-moduuli, jonka BlogController-ohjaimesta johdettiin luokat BlogChecker ja BlogCreator (Kuva 4.7). Myös alaluvun ”3.2.2 Koodin heikkouksia”

esimerkki Pie-luokan laskutoimitusten siirtämisestä PieEngineer-luokkaan on myös käypä esimerkki tehdyistä refaktoroinneista.



Kuva 4.7. BlogController-luokan refaktorointi

Suuren luokan vastuiden jakaminen useammiksi luokiksi myös mahdollisti yksityisten metodien muuttamisen julkisiksi ja näin ollen testattaviksi. Yksikkötestien toteutuksessa useita muitakin yksityisiä metodeja muutettiin julkisiksi ja niiden logiikkaa muutettiin niin, ettei muutos yksityisestä julkiseksi rikkonut kapseloinnin periaatteita. Tästä esimerkki on TeacherToolsController-luokan createResultArray-metodi, joka alussa oli määritelty yksityiseksi, koska se muutti suoraan luokan yksityisen ilmentymämuuttujan tilaa. Metodi muutettiin julkiseksi niin, että se palauttaa luotavan taulukon metodin kutsujalle, joka asettaa sen ilmentymämuuttujaan itse.

4.4.3 Löydetyt virheet

Yksikkötestien käyttöönotto myös paljasti kohdesovelluksen toiminnassa olevia bugeja ja puutteita. Näitä ja muita kohdattuja ongelmia ryhdyttiin korjaamaan refaktoroinnin ja yksikkötestien kirjoittamisen yhteydessä.

Käyttäjätunnusten hallinnasta eli AccountInformation-moduulista löytyi bugi, jossa käyttäjätason muuttamisessa käyttäjälle näkyvän ”Hae käyttäjätunnus listalta” -linkin osoite muuttui osoittamaan tunnuksen poistamiseen sen jälkeen, kun Hae-painiketta painettiin ilman yhdenkään hakutiedon syöttämistä. Tämä olisi ehkä voinut saada ylläpitäjän vahingossa poistamaan käyttäjän, vaikka tarkoituksena olisi ollut vain vaihtaa käyttäjän käyttäjätaso.

Edellä mainittu oli löydetyistä virheistä vakavimmasta päästä. Muut virheet olivat vähemmän vakavia. Muun muassa Blog-moduuliista korjattin bugi, jossa kommenttipainiketta ei näytetty käyttäjälle – käytännössä käyttäjä ei voinut poistaa kommenttia lainkaan.

XHTML-taulukoiden piirtämisestä vastaavasta Table-luokasta korjattiin väritystoimintoa niin, että `$everyNthLineColor` muuttujan arvo tarkoittaa myös käytännössä sitä, mitä muuttujan nimi tarkoittaa. Jos muuttujan arvo on kolme, taulukosta väritetään joka kolmas rivi eli rivit 1, 4, 7 ja niin edelleen.

JSON-luokan jsonEncode-metodi muutti boolean-tyyppiset arvot virheellisesti joko ykköseksi tai nollaksi riippuen siitä, oliko arvo true tai false. Lukuarvojen ympäriltä puuttuivat hakasulkeet, esimerkiksi true-arvon tapauksessa metodin olisi pitänyt palauttaa [1] eikä 1. Bugi korjattiin niin, että boolean-tyyppisten arvojen kohdalla käytetään PHP:n json_encode-funktiota.

Tidy-luokassa metodi, jota käytetään rivinvaihtomerkkien muuttamiseen XHTML-kielen vastaaviksi rivinvaihtoelementeiksi, sisälsi virheen. Metodissa tutkittiin virheellisellä säännöllisellä lausekkeella sisälsikö parametrina annettu merkkijono XHTML-tageja. Virhe korjattiin niin, että säännöllisen lausekkeen sijaan merkkijono kopioitiin tilapäiseen muuttujaan, jota käsiteltiin PHP:n strip_tags-funktiolla. Tämän jälkeen alkuperäistä merkkijonoa verrattiin käsiteltyyn merkkijonoon, ja mikäli merkkijonot erosivat, merkkijono sisälsi XHTML-tageja.

Bugien lisäksi yksikkötestaus paljasti kohdesovelluksesta myös turhaa koodia. Metodin kutsussa saattoi olla enemmän parametreja kuin itse metodin määrittelyssä, joista PHP-tulkki ei koskaan ilmoita. Joidenkin metodien kaikkia määriteltyjä parametreja ei käytetty metodissa lainkaan, ja metodien toteutuksessa saattoi myös olla tarpeettomia `for`-silmukoita ja muuta turhaa koodia. Metodeista löytyi myös turhia paikallisia muuttujia.

PHP-kieli sallii myös uusien julkisten attribuuttien luomisen oliolle, vaikka niitä ei olisi määritelty olion luokkaan lainkaan. Koodissa käytettiin `$this->attribute` viittausta, vaikka luokassa ei ollut määritelty kyseistä attribuuttia lainkaan. Tämä korjattiin määrittelemällä luokkaan näin käytettyjä attribuutteja.

4.4.4 Kohdatut ongelmat

Yksikkötestien kirjoittamisprosessin alussa ongelmia tuotti ensimmäiseksi PHPUnit. Testien ajo hakemistokokonaisuuksina epäonnistui, koska PHPUnitin lähdekoodiin oli kovakoodattu hakemistoerotin kauttamerkiksi (/), ja testausympäristön Windows XP -käyttöjärjestelmä käyttää kenomerkkiä (\). Myös mock-olioiden konfiguroinnissa käytettävä at-vaihtoehto, jonka tarkoituksena oli asettaa olio odottamaan metodikutsua vaaditussa kohdassa, ei osoittautunut toimivaksi, ja sen käyttöä testeissä tuettiin returnValue-konfiguraatiolla, jonka avulla oliot asetettiin palauttamaan lyhyt merkkijono. Näin testin lopussa metodien oikea kutsujärjestys voitiin varmistaa vertaamalla mock-olioiden palauttamien yhteen liitettyjen merkkijonojen lopputulosta odotettuun lopputulokseen.

Vakioiden käyttäytyminen testausympäristössä nousi esille vasta, kun kaikki yksikkötestit oli kirjoitettu. Vaikka kukin moduuli läpäisi testinsä ajettaessa ne yksinään, koko kohdesovelluksen testien ajaminen aiheutti sen, että Pass-moduulin testeistä moni epäonnistui. Tämä johtui siitä, että joissakin testiluokissa oli ladattu kielivakioita ja käytetty niitä testeissä, kun taas Pass-moduulin testit oli toteutettu niin, ettei vakioita ladattu. Koska PHPUnit suorittaa kaikki testit yhdessä ja samassa prosessissa, vaikuttivat nämä vakiotiedostojen lataamiset muistiin jokaisen testiluokan testeihin, ja Pass-moduulin testit epäonnistuivat, koska varsinainen lopputulos sisälsi tekstiä, jota odotetussa lopputuloksessa ei ollut. Ongelma korjattiin muuttamalla vakioita käyttävät testit toimimaan niin, ettei kielivakioita tarvita.

Osa testeistä epäonnistui siksi, että ne oli kovakoodattu siihen hetkeen, jolloin testi oli kirjoitettu. Esimerkiksi ExerciseTableView-luokan päivämäärädataa käsittelevän processDateString-metodin testi epäonnistui, kun testin kirjoittamisesta oli kulunut viikko, sillä testissä käytettiin testidatana testin ajamisen hetkeä, vaikka odotettu lopputulos oli sidottu tiettyyn hetkeen. Tämänkaltaiset ongelmat korjattiin poistamalla päivämääräriippuvuudet.

Toinen ongelma liittyi globaaleihin muuttujiin, ja se kohdattiin vasta kun yksikkötestien kirjoittaminen oli lopetettu. Tarkastellessa HTTP-luokan testiluokkaa huomattiin, etteivät sen globaaleja vakioita käyttävät testit toimineet täydellisessä eristyksessä, koska globaaleihin muuttujiin tehdyt muutokset vaikuttivat muihinkin testeihin. Tätä

lähdettiin korjaamaan tyhjentämällä globaalit muuttujat tearDown-metodissa. HTTP-luokan testit onnistuivat edelleen, mutta tämä sai taas PageHandler-luokan getPageNavigation-metodin testin epäonnistumaan. Tämä testi käytti web-palvelimen luomaa dataa $_SERVER-taulukosta testeissään. Taulukko $_SERVER on globaali muuttuja ja sisältää tietoa palvelimesta ja ympäristöstä, jossa PHP-tiedosto suoritetaan. Tällaisenaan testi oli riippuvainen ympäristöstä, jossa se suoritettiin, ja se korjattiin asettamalla käsin muuttujaan testissä tarvittava data.

Myös PHPUnitin tuottamat koodin kattavuuden analysointiraportit tuottivat virheellistä tietoa. Generoitu raportti saattoi laskea kokonaiskattavuuden todellista pienemmäksi, koska joissakin tilanteissa suorittamatta jääväksi koodiksi laskettiin koodirivit, jotka todellisuudessa suoritettiinkin. Seuraavassa kuvassa (Kuva 4.8) on esimerkki esiintyneestä koodin kattavuuden ongelmasta. Rivit 834 ja 835 rekisteröidään virheellisesti suorittamattomiksi, mikä vääristää kattavuuden lopputulosta. Ilmeisesti Xdebug ei ymmärrä usealle riville jaettuja lausekkeita.

```
825                 :     public function calcImageHeight()
826                 :     {
827                 3 :         $id = "calcImageHeight";
828                 3 :         $this->debug($id, "Input", "");
829                 :
830                 3 :         $groupTitleHeight = $this->getGroupTitleHeight();
831                 :
832                 :         $height = DIAGRAM_TOP_MARGINAL
833                 3 :                 + $this->diagram->getDiagramHeight()
834                 0 :                 + $groupTitleHeight
835                 0 :                 + LEGEND_TOP_MARGINAL
836                 3 :                 + $this->diagram->getLegendLeftColumnHeight();
837                 :
838                 3 :         $this->debug($id, "Return", $height);
839                 :
840                 3 :         return $height;
841                 :     }
```

Kuva 4.8. Koodin kattavuuden analysoinnin virheellinen toiminta

5 Testilähtöinen ohjelmistokehitys kohdesovelluksessa

Kohdesovelluksen kehittämisessä ja ylläpidossa ei ole ollut sen elinkaaren aikana määritelty minkäänlaisia ohjeita tai suositeltavia menetelmiä ohjelmistoprojektin työtavoille. Luvussa 4 kuvattiin yksikkötestien käyttöönotto osalle kohdesovellusta, ja tässä kappaleessa kerrotaan, miten kohdesovelluksen yksikkötestattuja luokkia kehitetään ja ylläpidetään testilähtöisen ohjelmistokehityksen periaatteita noudattaen. Kohdesovelluksen se koodi, jota ei luvussa 4 yksikkötestattu, vaatii kokonaan toisenlaisen suhtautumisen sovelluksen ylläpitämisessä, eikä tässä luvussa näin ollen esitellä toimintatapoja tämänkaltaisen koodin kanssa toimimiseen.

5.1 Vakioiden käsittely

Kohdesovelluksen PHP:n define-funktiolla määriteltyjä vakioita tulee yksikkötesteissä käyttää aina pelkästään vakioiden nimillä. Yksikkötesteissä ei siis tule käyttää vakioiden arvoja, sillä jos vakion arvoa muutetaan vakiotiedostossa, on muutos tehtävä myös yksikkötestiin. Tämä vakion nimen käyttö vakion arvon sijaan koskee sekä sovelluksen konfigurointiin tarkoitettuja vakioita että kielikohtaisia vakioita.

Yksikkötesteissä ladataan vain tarvittavat sovelluksen konfigurointiin tarkoitetut vakiotiedostot. Lataaminen tarkoittaa PHP-tiedoston sisällyttämistä yksikkötestikoodiin, esimerkiksi PHP:n require_once-funktiolla. Esimerkiksi Diagram- ja Pie-luokat tarvitsevat sovelluksen konfigurointiin liittyvistä vakioista laskutoimituksiin käytettäviä vakioita, joten tässä tapauksessa kyseisten luokkien konfigurointiin tarvittavat vakiot on perusteltua ladata.

Kielikohtaisia vakiotiedostoja ei tule koskaan ladata, sillä niiden lataaminen on täysin turhaa. Jos PHP ei löydä yksikkötestissä vakion nimen perusteella määriteltyä vakiota, PHP käyttää vakion nimeä sellaisenaan merkkijonona. Yksikkötesteissä ei tarvitse ottaa kantaa vakion arvoihin.

5.2 Yhteensopivuus tulevaisuuteen

PHPUnitiin on kehitteillä ominaisuus, jossa jokainen yksikkötesti ajetaan erillisessä PHP-prosessissa. Tämä vähentää muistinkulutusta ja eristää testit täysin toisistaan. (Bergmann 2007.) Yksikkötestin ajon aikana määritellyt vakiot ja globaalit muuttujat

eivät siten näy seuraavalle ajettavalle yksikkötestille. Erilliset PHP-prosessit olisivat estäneet alaluvussa 4.4.4 esitetyn ongelman, jossa vakioiden arvot näkyivät myös toisille yksikkötesteille. Kielikohtaisten vakioiden lataaminen on kuitenkin täysin turhaa, vaikka yksikkötestit ajettaisiin erillisissä PHP-prosesseissa.

Globaalien muuttujien, kuten PHP:n `$_SERVER`-, `$_GET`-, `$_POST`-taulukoiden arvot näkyvät myös muille yksikkötesteille. Tämän voi kuitenkin estää tyhjentämällä globaalien muuttujien ja taulukoiden arvot tearDown-metodissa.

Yksikkötestejä on myös syytä tarvittaessa refaktoroida, esimerkiksi jos huomaa, että jokin asian voi tehdä paljon helpommin tai yksinkertaisemmin. Koska osa luvussa 4 testatuista yksiköistä oli pitkiä ja hankalasti testattavia, tuli myös niiden yksikkötesteistä pitkiä ja hankalia. Jos sovelluskoodia refaktoroidaan, on hyvä miettiä myös, miten kyseistä sovelluskoodia testaavaa yksikkötestiä kannattaisi myös refaktoroida. Yksikkötestien eli yksittäisten testimetodien tulisi myös olla hyvien ohjelmointitapojen mukaisesti toteutettu.

Muuttamalla PHP-prosessia ajavan käyttöjärjestelmän päivämäärää, esimerkiksi kymmenen vuotta eteenpäin, voidaan testata itse yksikkötestejä. Yksikkötestien ei pidä epäonnistua siksi, että päivämäärä on esimerkiksi 10 vuotta tulevaisuudessa tai että kuukausi vaihtuu. Yksikkötestien on oltava myös toisistaan erillisiä, eivätkä ne saa jakaa mitään tietoja keskenään. Jokaisen yksikkötestin ehdoton tavoite on onnistua aina olosuhteista huolimatta.

6 Johtopäätökset

Opinnäytetyön myötä opimme paljon uusia asioita ja kertasimme ennestään osaamiamme teorioita. Muun muassa olioperustaisesta ohjelmistokehityksestä teki hyvää kerrata asioita. Käytännön kokemus ja osaaminen karttui refaktoroinnista ja yksikkötestauksesta PHPUnitilla. Vaikka emme päässeet kehittämään kohdesovellusta testilähtöisesti, olemme kuitenkin oppineet niitä tärkeitä perustietoja ja -taitoja, joita tarvitaan testilähtöisessä ohjelmistokehityksessä. Tähän tietopohjaan on hyvä opiskella ja hankkia käytännön kokemusta testilähtöisestä ohjelmistokehityksestä.

Lähtötilanne yksikkötestien toteuttamiseen ei ollut järin hyvä, vaikka sovellus olikin suurelta osin toteutettu oliomaisesti. Kuitenkin oma mielipiteemme kohdesovelluksen luokkien toteutustavasta, etenkin itse toteuttamistamme, oli alussa hyvä. Yksikkötestien kirjoittamisen ja ohjelmakoodin refaktoroimisen myötä tulimme kuitenkin näkemään, että ylläpidettävyydeltään ja ymmärrettävyydeltään sovelluksen koodi oli hyvinkin heikkoa ja vaatikin näin suuria refaktorointeja monissa kohdin. Toisaalta taas luokat, joiden toteuttamisessa olimme kiinnittäneet erityistä huomiota esimerkiksi juuri metodien pituuksiin ja logiikkaan, osoittautuivat nopeiksi ja helpoiksi testata, ja täten vahvistivat käsitystämme siitä, että ohjelmistokehityksessä suunnittelemiseen ja hiomiseen käytetty aika on kannattava sijoitus.

Yksikkötestien kirjoittaminen oli alussa hitaampaa, koska rutiinia ei testien kirjoittamiseen eikä esimerkiksi mock-olioiden käyttöön ollut. Taitojen ja tietojen karttumisen myötä työskentelynopeutemmekin kasvoi, kun taas yritysten ja erehdysten määrä väheni.

Yksikkötestien kattavista koodiriveistä kertovien raporttien mukaan suurin osa testatuista luokista testattiin kokonaisuudessaan. Joissakin pitemmissä metodeissa oli jäänyt jokin ehtolause suorittamatta, mikä osoitti sen, ettei täydellisen kattavuuden saavuttaminen ole itsestäänselvyys missään metodissa, ja että kyseessä olevat metodit olisivat vaatineet vielä enemmän refaktorointia.

Opinnäytetyön vaativuus ja kohdesovelluksen laajuus estivät meitä testaamasta kaikkia sovelluksen luokkia ja moduuleita, mikä ei luonnollisestikaan ollut edes tarkoituksemme, koska melko monta kohdesovelluksen kokonaisuutta olisi vaatinut täydellistä

uudistamista, ja tämänkaltainen työ ei kuulunut opinnäytetyömme piiriin. Testattavien moduulien listalta jäivät kuitenkin esimerkiksi moduulit Frontpage ja Gallup.

Toimivan ja siistin koodin saavuttaminen on testilähtöisen ohjelmistokehityksen vaatimaton tavoite (Beck 2003), ja työskennellessämme projektissa olemme oppineet arvostamaan näitä ominaisuuksia erittäin paljon. Omien kokemustemme perusteella pidämme testilähtöistä ohjelmistokehitystä mitä parhaimpana menetelmänä ohjelmistokehitykseen, ja tulemmekin varmuudella käyttämään sitä innolla tulevissa projekteissamme.

Lähteet

Ambler, Scott W. 2009. Introduction to Test Driven Desing (TDD) [online][viitattu 23.9.2009]. www.agiledata.org/essays/tdd.html

Beck, Kent 2003. Test-driven development : by example. Boston : Addison-Wesley.

Beck, Kent, Beedle, Mike, Bennekum, Arie van, Cockburn, Alistair, Cunningham, Ward, Fowler, Martin, Grenning, James, Highsmith, Jim, Hunt, Andrew, Jeffries, Ron, Kern, Jon, Marick, Brian, Martin, Robert C., Mellor, Steve, Schwaber, Ken, Sutherland, Jeff & Thomas, Dave 2001. Manifesto for Agile Software Development [online][viitattu 3.10.2009]. www.agilemanifesto.org

Beck, Kent: Simple SmallTalk Testing with Patterns. [online][viitattu 18.9.2009]. xprogramming.com/testfram.htm

Bergmann, Sebastian 2007. Isolated and Parallel Test Execution in PHPUnit 4. [online][viitattu 2.11.2009]. sebastian-bergmann.de/archives/730-Isolated-and-Parallel-Test-Execution-in-PHPUnit-4.html

Bergmann, Sebastian 2009a. Copyright. [online][viitattu 15.9.2009]. www.phpunit.de/wiki/Copyright

Bergmann, Sebastian 2009b. PHPUnit Manual. [online][viitattu 16.9.2009]. www.phpunit.de/manual/3.3/en/phpunit-book.pdf

Bergmann, Sebastian 2009c. PHPUnit. [online][viitattu 15.9.2009]. www.phpunit.de

Bjork, Russell C. 2004: An Example of Object-Oriented Design: An ATM Simulation. [online][viitattu 7.9.2009]. www.cs.gordon.edu/courses/cs211/ATMExample/index.html

Burback, Ronald LeRoi 1998. Unit Testing. [online][viitattu 12.9.2009]. infolab.stanford.edu/~burback/watersluice/node22.html

Clark, Mike 2006. JUnit FAQ. [online][viitattu 10.11.2009]. junit.sourceforge.net/doc/faq/faq.htm

Dahl, Ole-Johan & Nygaard, Kristen 2001. How Object-Oriented Programming Started. [online][viitattu 31.8.2009]. heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html

Designing Multi-Tier IIS Applications 2009. [online][viitattu 14.10.2009]. msdn.microsoft.com/en-us/library/ms524900.aspx

EDU - Ammattiosaajan työkykypassi 2008. Ammattiosaajan työkykypassi. [online][viitattu 31.8.2009]. www.edu.fi/page.asp?path=498,529,26515,27466,86190

Fowler, Martin, Beck, Kent, Brant, John, Opdyke, William & Roberts, Don 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional. 14th printing.

Fowler, Martin 2007. Mocks Aren't Stubs. [online][viitattu 18.9.2009]. martinfowler.com/articles/mocksArentStubs.html

Koskimies, Kai 2000. Oliokirja. 2. muuttumaton painos. Satku. Helsinki.

Langr, Jeff 2002. Enlightened Java Style. [online][viitattu 14.10.2009]. www.ddj.com/architect/184414826

Lauder, Anthony: Test Driven Development. [online][viitattu 18.9.2009]. c2.com/cgi/wiki?TestDrivenDevelopment

Lecky-Thompson, Ed, Nowicki, Steven D. & Myer, Thomas 2009. Professional PHP6. Programmer to programmer. Indianapolis, IN: Wiley.

Link, Johannes & Fröhlich, Peter 2002. Unit testing in Java: how tests drive the code. CA: Morgan Kaufmann Publishers.

Martin, Robert C. 2006. Agile vs. XP: The Differences and Similarities. [online][viitattu 14.10.2009]. www.objectmentor.com/omSolutions/agile_xp_differences.html

Meszaros, Gerard 2008. Test Double. [online][viitattu 10.11.2009]. xunitpatterns.com/Test%20Double.html

Osherove, Roy 2009. The Art of Unit Testing: With Examples in .Net. Manning Pubs Co Series. Manning Publications.

PHP: New Object Model – Manual 2009. New Object Model. [online][viitattu 1.9.2009]. www.php.net/manual/pt_BR/migration5.oop.php

Schlossnagle, George 2004. Advanced PHP programming. Sams Publishing.

Shore, James & Warden, Shane 2007. The Art of Agile Development. O'Reilly Media. CA.

Wells, Don 2009a. Extreme Programming: A gentle introduction. [online][viitattu 14.10.2009]. www.extremeprogramming.org/

Wells, Don 2009b. Surprise! Software Rots! [online][viitattu 3.10.2009]. www.agile-process.org/change.html