jamk.fi

Testivetoinen kehitys osana Magento 2 -moduulikehitysprosessia

Roope Salminen

Opinnäytetyö
Toukokuu 2019
Liiketalouden ala
Tietojenkäsittelyn tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Salminen, Roope	Opinnäytetyö, AMK	Toukokuu 2019
		Julkaisun kieli Suomi
	Sivumäärä 46 + 22	Verkkojulkaisulupa myönnetty: x

Työn nimi
Testivetoinen kehitys osana Magento 2 -moduulikehitysprosessia

Tutkinto-ohjelma
Tietojenkäsittelyn tutkinto-ohjelma

Työn ohjaaja(t)
Niko Kiviaho

Toimeksiantaja(t)
Solteq Oyj

Tiivistelmä

Nykyaikainen ohjelmistotuotanto käsittää useita eri kehitysmenetelmiä ja prosesseja. Näiden prosessien hyödyntäminen vaihtelee runsaasti alusta- ja teknologiakohtaisesti. Eräs tunnettu kehittämismenetelmä on testivetoinen kehitys (test driven development), joka toimii tutkimuksen ytimenä.

Tutkimuksen tavoitteena oli selvittää testivetoisen kehittämisprosessin hyödyntämismahdollisuudet Magento 2 -verkkokauppa-alustan moduulikehityksessä, havainnoida kehitysmenetelmän kannalta oleellisimmat seikat ja räätälöidä tuloksista toimeksiantaja Solteq Oyj:n Magento -kehitystiimin tarpeisiin sopivat ratkaisut.

Tutkimus suoritettiin kehittämistutkimuksena, sillä tuotoksena pyrittiin kehittämään toimeksiantajan moduulikehitysprosessia. Tutkimuksen aikana luotiin moduuli Magenton kehitysmenetelmiä noudattaen ja testivetoisen kehityksen tärkeimpiä elementtejä hyödyntäen. Kehityksen aikana dokumentoitiin prosessissa esiintyneitä ilmiöitä, joiden perusteella laadittiin johtopäätökset menetelmän käyttöönottoa koskien.

Opinnäytetyön tuotoksena luotiin ohjeistus kehitysmenetelmän ja sen työkalujen käyttöönotosta. Aiheesta jaettiin tietoa toimeksiantajan Magento kehitystiimille. Johtopäätöksenä kehitysmenetelmän todettiin olevan mahdollinen osa moduulikehitysprosessia, mikäli sen käyttöönottoon panostetaan tarpeeksi. Menetelmän tarjoamien laadunhallinnallisten ratkaisujen todettiin vaativan tarkempaa käytännön testausta ennen kuin luotettavia johtopäätöksiä voidaan muodostaa.

Avainsanat (asiasanat)

Testivetoinen kehitys, TDD, PHPUnit, PhpStorm, Magento 2, Ohjelmistotestaus

Muut tiedot

jamk.fi

Description

Author(s)	Type of publication	Date
Salminen, Roope	Bachelor's thesis	Toukokuu 2019
		Language of publication: Finnish
	Number of pages 46 + 22	Permission for web publication: x

Title of publication

Test driven Magento 2 module development process

Degree programme

Business information systems

Supervisor(s)

Kiviaho, Niko

Assigned by

Solteq Oyj

Abstract

Modern software development encompasses a multitude of different development methods and processes. The utilization of these processes varies greatly between different platforms and technologies. Test driven development is a well-known development method that acts as the central point for this research.

The goal of this research was to investigate the possibilities of utilizing test driven development in the module development process of Magento 2 e-commerce platform, observe the most relevant issues related to the usage of this development method and apply the findings to create methods suitable for the requirements of the Magento development team of the research assigner Solteq Oyj.

The research was implemented as a development research, as the research aims to create an improvement to the process of module development. During the research, a module was created using a process that utilized Magento's development methods and the most important elements of test driven development. The results concerning the deployment of the development method were compiled from the remarks observed and documented during the process.

As the result of this research, instructions for the deployment and utilization of the development method and tools were created and distributed to the Magento development team. As the conclusion to the research, test driven development was established to be a prospective part of the module development process, provided that the deployment of the method is carried out in an accurate manner. The benefits of this method considering the quality of code were deemed to require a more hands-on approach before any feasible conclusions could be made on the subject.

Keywords/tags (subjects)

Test driven development, TDD, PHPUnit, PhpStorm, Magento 2, Software testing

Miscellaneous

Sisältö

Kuviot

Määritelmät

Integraatio	Kahden tai useamman toisistaan teknisesti eroavan kokonaisuuden välisen kommunikaation mahdollistava rakennelma.
Integrointitesti	Testityyppi, joka testaa kahden eri järjestelmän tai yhden järjestelmän sisältämien komponenttien välistä kommunikaatiota.
HTML	Hyper text markup language on merkintäkieli, jota käytetään esimerkiksi verkkosivujen rakentamiseen
Luokka	Luokka on olio-ohjelmoinnissa olion ominaisuuksien määritelmä, joka käsittää sen tiedot ja toiminnan. Yhdestä luokasta voidaan luoda useita ilmentymiä.
Magento 2	PHP ohjelmointikielellä rakennettu vapaan lähdekoodin verkkokauppa-alusta.
Malli	Malli, eli model sisältää toiminnallisuudet tiedon tallentamiseen, käsittelyyn ja ylläpitämiseen MVC-arkkitehtuurimalliin pohjautuvassa järjestelmäkokonaisuudessa.
Metodi	Luokan aliohjelma, joka suorittaa sille määritetyn toiminnon. Sen toimintaa voidaan kutsua luokan sisältä tai muualta ohjelmistosta.
Moduuli	Ohjelmiston itsenäinen osanen, joka suorittaa sille tarkoitetun tehtävän. Modulaarisessa ohjelmoinnissa tietokoneohjelma rakennetaan moduuleista.
Olio	Olio-ohjelmoinnissa käytettävä luokan ilmentymä. Se sisältää tiedot luokan attribuuteista ja metodeista ja mahdollistaa luokan käytön ohjelmiston kokonaisuudessa.

PHP	Hypertext Preprocessor on ohjelmointikieli, jota käytetään erityisesti web-palvelinympäristöissä ja dynaamisten sivujen luonnissa.
Refaktorointi	Prosessi, jossa ohjelmiston koodia muokataan niin, että sen toiminnallisuus säilyy, mutta rakenne muuttuu, usein luettavampaan ja selkeämpään muotoon.
Service Contract	Sopimuspohjaiseen ohjelmointimalliin pohjautuva käsite, jossa ohjelmistokomponenteille määritellään muodolliset, täsmälliset ja tarkastettavat rajapintamääritykset, joita kutsutaan sopimuksiksi.
TDD	Testivetoinen kehitys (test driven development) on kehitysprosessi, jossa ohjelman automatisoidut testi luodaan pienissä sykleissä ennen ohjelmistokoodia. Menetelmässä ohjelmistokoodia kirjoitetaan vain läpäisemään luodut testit.
Testitapaus	Kirjoitettavan toiminnallisuuden testattava tapaus, joka sisältää testisyötteet ja testien odotusarvot.
Yksikkötesti	Ohjelman pienen kokonaisuuden testi, joka pyrkii tarkistamaan ohjelman osien itsenäisen toimivuuden.
XML	Extensible Markup Language on merkintäkieli, jota käytetään formaattina tiedon kuljettamiseen järjestelmien välillä, sekä dokumenttien tallentamiseen.

1 Johdanto

Yksi nykyaikaisen ohjelmistotuotannon tärkeimmistä piirteistä on loppukäyttäjälle tarjottavan tuotteen laatu. Laatu voi tarkoittaa montaa asiaa, ja sen mittaamiseen on useita keinoja, mutta ohjelmistotuotannossa sitä käytetään usein kuvastamaan koodin selkeyttä ja ”puhtautta”, sekä ohjelmistokokonaisuuden toimivuutta.

Valinnanvaran määrän ollessa tänä päivänä massiivinen, asiakkaat etsivät usein parhaan laadun tarjoavaa palvelua. Tämän takia yhä useammat ohjelmistoyritykset etsivät ratkaisuja oman tuotteensa laadun varmistamiseksi. Laadunvarmistuksen tärkeimpiä elementtejä on ohjelmistotestaaminen, joka on kulkenut käsi kädessä ohjelmistokehityksen kanssa vuosikymmenien ajan. Testaaminen on koettu välttämättömäksi, tarkkuutta vaativaksi prosessiksi ohjelmistoon väistämättä kulkeutuvien virheiden takia.

Laadukkaan koodin tuottamiseen on kehitelty useita menetelmiä ohjelmistotuotannon historian aikana, joista eräs on testivetoinen kehittäminen (test driven development). Tätä mallia noudattamalla luodaan ohjelmiston koodia testaavat yksikkö- ja integrointitestit ennen varsinaisen tuotantokoodin kirjoittamista ja tämä johtaa laadukkaampaan lopputulokseen. Näin voidaan huomioida ohjelmiston laadunvarmistus jo ennen ohjelmistotestauksen aloittamista.

Testivetoisen kehittämisen aiemmat tutkimukset eivät kuitenkaan ulotu jokaiseen alustaan ja teknologiaan asti ja tässä opinnäytetyössä tutkitaan menetelmän hyödyntämismahdollisuuksia Magento 2 -verkkokauppa-alustassa.

2 Tutkimusasetelma

Tässä luvussa kuvataan tutkimuksen lähtöasettelun keskeiset asiat kuten tutkimuksen taustat, tavoitteet, asetetut rajaukset ja työkalut. Näillä pyritään saavuttamaan halutut tulokset. Luvun tarkoituksena on myös antaa lukijalle selkeä

kuva kehitysprosessista, jonka tutkimisen ja kehittämisen ympärille opinnäytetyö on rakennettu.

Solteq Oyj osti Magento-verkkokauppoja toimittavan yrityksen Pardco Group Oy:n 20.12.2016. Solteqin uusi Magento osa-alue haluaa kehittyä vastaamaan vielä paremmin laajentuneen ja yrityskooltaan suuremman asiakaskunnan tarpeisiin. Siksi myös kehitysprosessin parantaminen ja laadunvarmistaminen on ajankohtaista.

Opinnäytetyön tavoitteena on tunnistaa testivetoisen kehitysprosessin mahdollisuudet ja haasteet, sekä muodostaa niistä Magento tiimin tarpeisiin soveltuvat ratkaisut. Opinnäytetyöstä on rajattu pois kehitettävän prosessin testaus käytännössä. Opinnäytetyön suunnittelun aikana koettiin, ettei prosessin vaikutusten mittaaminen tuottaisi luotettavia tuloksia työlle varatussa ajassa, sillä uusi prosessi olisi ollut käytössä liian lyhyen aikaa. Myös hyväksymis- testivetoinen kehitys (ATDD) on rajattu opinnäytetyön ulkopuolelle ja työ käsittelee vain testivetoisen kehitysprosessin rakentamista Magento kehittäjille.

2.1 Tutkimuskysymykset ja -menetelmä

Opinnäytetyössä haetaan vastauksia seuraaviin kysymyksiin:

- Mitä on testivetoinen kehitys ja mitkä ovat sen työvaiheet?
- Mitä vaatimuksia testivetoinen kehitys tuottaa moduulikehitysprosessin eri vaiheille?
- Millaisilla työkaluilla ja miten testivetoista kehitystä kannattaa hyödyntää Magento 2 -alustassa?

Opinnäytetyön tutkimusmenetelmä on kehittämistutkimus, sillä tavoitteena on kehittää Solteqin Magento -tiimin moduulikehitysprosessia. Kehitystutkimuksen aikana koostetaan omat tilanteeseen sopivat menetelmät, jotka muodostuvat tutkimuksen vaatimuksista ja tarpeista. Kananen (2015, 33) kuvastaa tätä menetelmänä, jossa yhdistetään sekä kvalitatiivisia, että kvantitatiivisia tutkimusmenetelmiä.

Solteqilla ei olla suoritettu aiempia tutkimuksia testivetoisen kehityksen hyödyntämisessä Magento 2 -alustassa. Tutkimuksen aikana käydään läpi sille olennaisiin aiheisiin liittyvää kirjallisuutta, artikkeleita ja muita ulkopuolisia lähteitä. Lähteistä kerätään aiheeseen liittyviä tutkimustuloksia, joita analysoimalla saadaan selvyys tutkimuskohteen nykytilanteesta ja kehitysmahdollisuuksista. Aiheesta on vielä suhteellisen rajallinen määrä tutkimusta, mutta erilaisen lähteiden tuloksia yhdistämällä saadaan lähtökohta siitä kuinka menetelmän kehitys tulisi aloittaa.

Kehittämistutkimuksen tarkoituksena on luoda parannuksia kehityskohteeseen, kuten tuotteeseen, järjestelmään tai käytänteisiin. Se ei ole pelkästään teoreettista tutkintaa, vaan tutkimus paikannetaan fyysisesti valittuun kohteeseen. Hautamäen (2015) mukaan tutkimuksen tulee erota aiemmin tehdyistä tutkimuksista ja aiempia tutkimuksia onkin tarpeellista hyödyntää lopputuloksen tarkastelussa ja analysointivaiheessa. Kehittämistutkimus sopii opinnäytetyön menetelmäksi, sillä tuloksena halutaan saada aikaiseksi ohjeistus sellaisen kehitysprosessin käyttöönottoon, joka vastaa kehitystiimin tarpeita.

2.2 Opinnäytetyön tietoperusta ja rakenne

Tutkimuksessa pyritään ymmärtämään ja täyttämään kehitystiimin tarpeet hakemalla aiheesta tietoa mm. luotettavasta kirjallisuudesta ja internetistä löytyvistä alan erityisosaajien kirjoittamista artikkeleista. Tutkimustuloksia saadaan lopulta testaamalla analysoidun aineiston pohjalta luotua työprosessia käytännössä tutkimuksessa määritellyillä työkaluilla.

Opinnäytetyön teoriaosiossa avataan tutkimuksen kannalta tärkeimpiä tekijöitä, eli Magento 2 -alustaa, testivetoista kehitystä ja erilaisia testausmenetelmiä. Yhteenvetona tiivistetään tutkimuksessa löytyneitä tuloksia ja havaintoja, sekä luodaan johtopäätökset siitä, kuinka tutkimustuloksina löydetyt menetelmät soveltuvat käytännössä kehitystiimin tarpeisiin. Pohdintaosiossa tarkastellaan tutkimustyötä reflektiivisesti, sekä esitetään ideoita aiheen jatkokehittämisestä.

2.3 Tutkimuksen työkalut

Tutkimuksessa hyödynnetään seuraavia työkaluja kehitettävän koodiosion luonnissa. Kehitettävä osio sisältää testivetoista kehitysmenetelmää käyttäen luodun Magento 2 -moduulin, jonka kehittämisprosessi kuvataan Luvussa 7. Tutkimuksen työkalut valittiin sen perusteella mitä työkaluja Magento tiimi käyttää päivittäisessä työssään, koska tutkimuksen haluttiin tuottavan mahdollisimman paljon hyötyä kehitystiimin nykyiseen työskentelymenetelmään. Tarkoituksena on liittää tuotettu menetelmä osaksi olemassa olevaa kehitysprosessia ja saada siitä käytännön hyötyä projektien kehitykseen.

Ubuntu

Tutkimuksessa käytettävä käyttöjärjestelmä on Debian Linux-jakeluun perustuva avoimen lähdekoodin Ubuntu. Ubuntun komentorivi ja käytön esteettömyys tekevät siitä toimivan valinnan tutkimustyön alustaksi.

PhpStorm

Tutkimuksen ohjelmointiympäristöksi valittiin JetBrainsin kehittämä PhpStorm sen käyttäjäystävällisyyden, sisäänrakennettujen yksikkötestityökalujen ja muiden PHP kehitystä hyödyttävien ominaisuuksien, kuten sisäänrakennetun versionhallintatuen takia. PhpStorm on myös Magento -tiimin pitkäaikaisesti suosima ohjelmointiympäristö, joka on tärkeä seikka kehitysprosessin luonnissa.

Uuteen ohjelmointiympäristöön vaihtaminen saattaisi tuoda tiimin kehittäjille testivetoisen kehitysprosessin sisäistämisen lisäksi turhan paljon haasteita tehokkaan työprosessin säilyttämiseksi.

PHPUnit

Testauksen työkaluksi valittiin yksikkötestauksen ohjelmistokehys PHPUnit. PhpStorm sisältää sisäänrakennetun tuen PHPUnit testaamista varten, ja sen hyödyntämisestä Magenton 2:n testaamisessa on runsaasti aineistoa Magenton virallisessa dokumentaatiossa. Testausprosessin helpottaminen on tämän tutkimuksen avainasia, jolloin työkalujen valinnan kriteerinä on ehdottomasti oltava niiden käyttäjäystävällisyys ja dokumentaation kattavuus.

Kehityksessä hyödynnetään myös Magento 2:n tarjoamia testityökaluja, sekä yritetään noudattaa Magenton ohjeistamia yksikkötestauksen käytänteitä. Huomiota kiinnitetään myös Magenton muiden työkalujen toimintaan osana testivetoista kehitysprosessia ja uusia kehitysmahdollisuuksia kartoitetaan jatkuvasti tutkimuksen aikana.

3 Magento

Magento 2 on yksi tämän hetken johtavista avoimen lähdekoodin verkkokauppa-alustoista, jonka suosion lähteenä ovat alustan avoimuus, lisäosat ja muokattavuus, sekä lukuisat ominaisuudet, jotka on tehty parantamaan verkkokauppa-alustan kokonaisvaltaista käytettävyyttä. (Nyári 2017.)

Magenton on kehittänyt web-ohjelmointiyritys Varien, joka julkaisi ensimmäisen käytettävän Magento version maaliskuussa 2008. Helmikuussa 2011 Varienin entinen toimitusjohtaja Roy Rubin myi huomattavan määrän Magenton osakkeita eBaylle, joka myöhemmin osti Magenton kokonaan itselleen ja perusti uuden yhtiön sen ympärille. (Nyári 2017.)

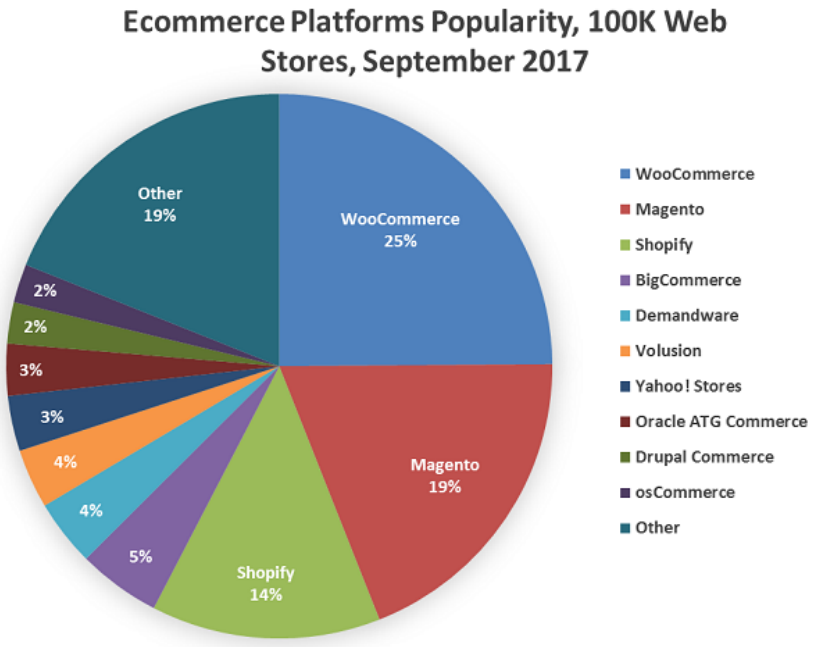
2015 Magento irtautui eBaysta omaksi yhtiökseen, yksityisen (private equity) pääomayhtiö Permiran Alaisuuteen. Samana vuonna tapahtui myös suuri harppaus Magenton kehityksen kannalta, eli Magento 2 alustan julkaisu. Magento 2

mahdollisti kehittäjille modernien palveluiden integroimisen järjestelmään Yksinkertaisesti API:en avulla, sekä paransi käyttäjäystävällisyyttä tekemällä kriittisiä muutoksia tilauksenhallintaan ja kassajärjestelmään. Koodista tehtiin luettavampaa, ja uuteen järjestelmään tuotiin joukko uusia teknologioita helpottamaan kehitystyötä, sekä tekemään kaupan navigoinnista tehokkaampaa. (Nyári 2017.)

Magento 2 teknologiat (Nyári 2017):

- PHP 7
- Varnish
- Redis
- Modern JS Stack (JQuery, RequireJS, Knockout.js)
- Solr
- PHPUnit
- Composer.

Magento 2 sopii käytettävyytensä ja skaalautuvuutensa vuoksi sekä keskisuuriin, että pieniin verkkokauppatoteutuksiin, ja se lasketaankin tänä päivänä suurimpien verkkokauppa-alustojen joukkoon käyttäjien perusteella. Syyskuussa 2017 Aheadworks-sivuston tekemän verkkokauppojen markkinatutkimuksen mukaan Magento on onnistunut pysymään viime vuosien ajan verkkokauppamarkkinoiden kärkijoukossa, vaikkakin WooCommerce on ohittanut sen suosiossa. Wordpress sivustoille luotu verkkokaupparatkaisu WooCommerce on Magenton tämän hetken suurin kilpailija. (Who Leads the Market of Ecommerce Platforms in 2017, 2017.)



Kuvio 1. Kymmenen suurimman nerkkokauppa-alustan markkinaosuuksia sadan tuhannen kaupan otteella kuvastava kaavio (Who Leads the Market of Ecommerce Platforms in 2017, 2017).

Eräs tämän päivän verkkokauppojen tärkeimmistä ominaisuuksista on käyttäjäkokemuksen tarjoaminen. Yksi tapa tähän on sisältörikkaiden, laadukkaiden sivujen luominen sisällönhallintajärjestelmää (Content Management System) hyödyntäen. Magento 2:n sisällönhallintaa on kritisoitu sen muita ominaisuuksia heikommaksi rajallisten sisältötyökalujen ja -mallien, sekä vanhentuneen editorin vuoksi (Yablonskaya 2018).

Useat moduulivalmistajat ovat luoneet ratkaisuja sisällönhallintajärjestelmän aiheuttamiin puutteisiin. Näitä ovat mm. moduulit, jotka laajentavat hallintatyökaluja, sekä integraatiot, joissa Magento yhdistetään ulkoisiin sisällönhallintajärjestelmiin, kuten Wordpressiin. (Yablonskaya 2018.)

Adobe julkisti aikomuksensa ostaa Magenton toukokuussa 2018, ja tästä seurasi spekulaatiota Magento 2:n jatkokehitystä koskien. Adobe tunnetaan sen tarjoamasta,

massiivisesta luovan tuotannon ekosysteemistä, joten tulevaisuudennäkymät Magenton käyttäjäkokemuksen kehitykselle vaikuttavat kirkkaalle. (Porter 2018.)

3.1 Magento 2 arkkitehtuuri

Magenton 2:n arkkitehtuuri koostu sen ydinkoodista, sekä moduuleista, jotka laajentavat tai korvaavat sen toiminnallisuuksia. Moduulit toimivat osana Magenton ydinkoodia, joka on järjestetty kerroksiin. Kerroksittainen sovellusarkkitehtuuri on yksi Magento 2:n rakenteen tärkeimmistä elementeistä. (Magento 2018.)

Magento 2:n dokumentaatiossa (2018) kuvatut arkkitehtuuriin kuuluvat kerrokset ovat:

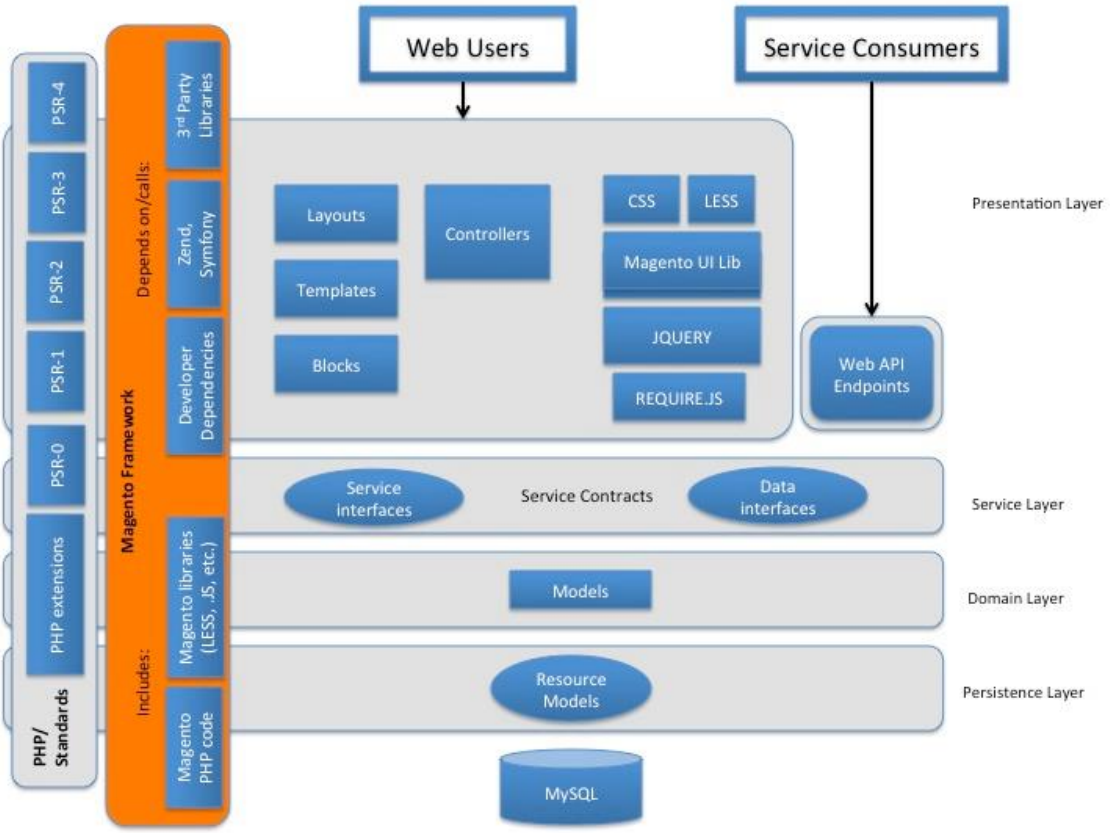
Presentation layer on Magenton neljästä kerroksesta päällimmäisin. Se sisältää näkymä elementtejä (layout, block, template) ja kontrollereita (controller), jotka prosessoivat käyttöliittymään kohdistuvia komentoja.

Service layer toimii siltana Presentation layerin ja Domain layerin sisältämien mallien (model) välillä käyttäen sopimuksia (service contract). Sopimukset ovat joukko moduulille määriteltyjä PHP rajapintoja, jotka säilyttävät datan eheyden ja piilottavat ydinkoodin tarkemmat yksityiskohdat. Service layer tarjoaa rajapinnat, joihin voi lähettää pyyntöjä toisista moduuleista, tai SOAP ja REST API:n kautta.

Domain layer sisältää Magenton ydinkoodin (business logic), joka määrittää minkälaisia toimintoja voidaan suorittaa kullekin datatyypille. Domain layerilla sijaitsevat mallit ovat yhteydessä resurssimalliin (resource model), jota ne käyttävät hakemaan dataa tietokannasta. Domain layerilla sijaitsevaan logiikkaan päästään moduulien välillä käsiksi palvelupyyntöjen avulla.

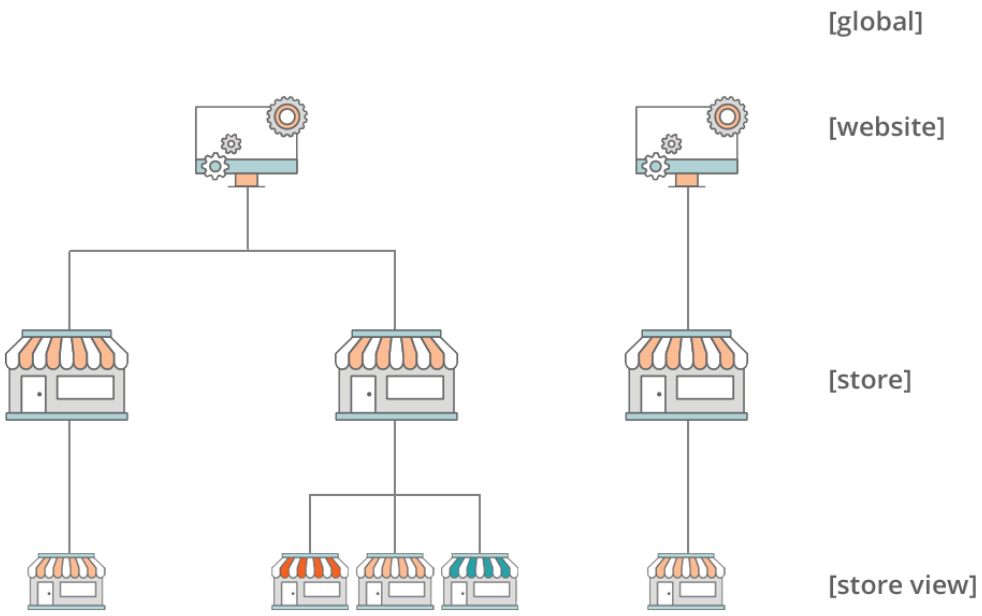
Persistence layer sisältää resurssimallit, jotka liittävät oliot (object) tietokannan taulujen riveihin. Resurssimallin velvollisuutena on myös suorittaa kaikki tietokantaan

kohdistuvat CRUD (create, read, update, delete) pyynnöt. Yksinkertainen resurssimalli määrittelee yhden tietokannan taulun, jonka kanssa se on vuorovaikutuksessa.



Kuvio 2. Magento 2 arkkitehtuuri (Magento 2018).

Magento 2 –verkkokaupan ylläpito tapahtuu sen hallintapaneelista, joka sisältää lukuisia hyödyllisiä toimintoja yhdistettynä käyttäjäystävälliseen käyttöliittymään. Hallintapaneelista voidaan konfiguroida verkkokaupan asetuksia globaalisti, jolloin vaikukset näkyvät jokaisessa asennuksen verkkokaupassa ja kauppanäkymässä, tai rajata pienempään scopeen, joita ovat website, store ja store view (kuvio 3). Näitä rajauksia voidaan hyödyntää esimerkiksi tapauksissa, jossa samasta verkkokaupasta luodaan useita kieliversioita. (Magento 2018.)



Kuvio 3. Kuvaus Magenton 2 verkkokaupan scope hierarkiasta (Magento 2018).

Hallintapaneeli tarjoaa kattavat mahdollisuudet verkkokaupan sisällön ja ulkoasun säätöihin. Hallittavia osa-alueita ovat asiakas, katalogi, markkinointi, sisältö ja myyntiosio, sekä verkkokaupan yleisasetukset. Hallinta on tehty käyttäjäystävälliseksi, mutta kuitenkin tarvittavan monipuoliseksi ja joustavaksi tarjoamaan kauppiaille kaikki tarvittavat työkalut toimivan verkkokauppa-alustan pyörittämiseen. (Magento 2018.)

3.2 Moduulikehitys

Moduuli on Magenton ydinkoodia laajentavista tai uusia toiminnallisuuksia sisältävistä tiedostoista koostuva kansiorakenne. Magenton modulaarisuutta noudattaen moduulin pyrkimyksenä on täyttää yksi määritelty tehtävä, ja sisältää minimaalinen määrä riippuvuutta muiden moduulien toimintaan. Moduulin sisältö koostuu Magenton arkkitehtuurin eri tasoilla sijaitsevista PHP ja XML tiedostoista, jotka liittyvät johonkin Magenton toimintalogiikan osa-alueeseen. (Magento 2018.)

Magenton omat moduulit sijaitsevat verkkokaupan kansiorakenteen **vendor** kansiossa, kun taas kolmannen osapuolen moduulit useimmiten sijoitetaan **app/code** kansioon. Tässä ovat poikkeuksena Composer ohjelman kautta asennetut kolmannen osapuolen moduulit, jotka asentuvat myös **vendor** kansioon. (Magento 2018.)

Magento käyttää koodissaan riippuvuusinjektio (dependency injection) suunnittelumallia käsittelemään luokkien välisiä riippuvuuksia. Nämä riippuvuudet voidaan määritellä luokan konstruktoriin tai niitä hyödyntävään metodiin. Moduulin di.xml tiedostossa määritellään riippuvuuksien tyyli ja toteutus. Tämä ominaisuus yhdistettynä rajapintojen käyttöön matalan ja korkean tason koodin välillä varmistaa moduulien koodin integriteetin säilymisen, vaikka ydinkoodia päivitettäisiin. (Magento 2018.)

Moduulikehityksessä tulee usein vastaan tilanteita, jossa Magenton ydinkoodia täytyy laajentaa tai muokata toteuttamaan haluttua toiminnallisuutta. Magento 2:n dokumentaatiossa (2018) ohjeistetaan suorittamaan muokkauksia seuraavilla keinoilla:

Plugin (interceptor) on luokka, joka muokkaa jonkin metodin toiminnallisuutta keskeyttämällä sen suorittamisen määriteltynä hetkenä ja sen jälkeen laajentamalla tai korvaamalla sen toiminnallisuutta. Pluginit määritellään moduulin di.xml tiedostossa, ja niitä ei voida määritellä luokan yksityisiin metodeihin. (Magento 2018.)

```
<config>
    <type name="{ObservedType}">
        <plugin name="{PluginName}" type="{PluginType}"
        sortOrder= "1" />
    </type>
</config>
```

Kolme Plugin tyyppiä ovat:

- Before, joka ajetaan ennen alkuperäisen metodin suorittamista. Before metodin avulla voidaan muokata alkuperäisen metodin parametrejä halutunlaiseksi ennen sen suoritusta.

- After, joka ajetaan sen jälkeen, kun alkuperäisen metodin suoritus on valmis. After metodilla muokataan alkuperäisen metodin palauttamaa arvoa. After pluginia ei voida käyttää metodiin, jolla ei ole palautusarvoa.

- Around, joka ajetaan ennen ja jälkeen alkuperäisen metodin suorittamisen. Around metodi ylikirjoittaa metodin toiminnallisuuden muokkaamalla sen logiikkaa ja palautusarvoa. Around voi myös tarvittaessa kutsua alkuperäistä metodia tai seuraavaa plugin jonossa olevaa metodia. Magento kehottaa välttämään around metodin käyttöä, sillä se vaikuttaa negatiivisesti ohjelman suorituskykyyn ja kasvattaa stack tracen määrää.

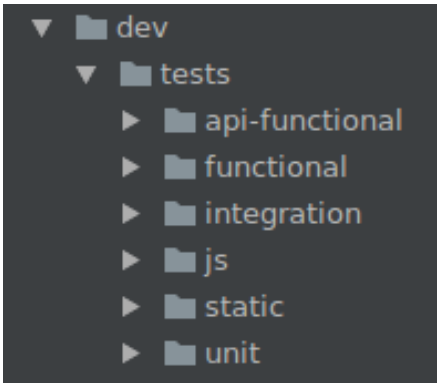
Class Preference on tapa konfiguroida mistä luokasta luodaan ilmentymä, kun haetaan jotain tiettyä luokkaa esim. riippuvuusinjektion tai kolmannen osapuolen haun kautta. Stormin (2015) mukaan tätä ominaisuutta käytetään pääasiassa määrittelemään minkä luokan ilmentymää käytetään rajapintaa kutsuessa. Sen avulla voidaan kuitenkin joissain tapauksissa ylikirjoittaa ydinkoodin toiminnallisuuksia tai osia niistä esimerkiksi määrittelemällä preference jollekin Magenton ydinluokalle ja luomalla tälle uusi ilmentymä muokatulla toiminnallisuudella. (Magento 2018.)

```
<config>
    <preference for="TestVendor\TestModule\Api\Data\TestInterface"
    type="TestVendor\TestModule\Model\Test"
    />
</config>
```

Event Observer:in käyttö on myös tapa laajentaa Magenton ydinkoodin toiminnallisuutta. Eventit ovat suoritettavia tapahtumia, jotka käynnistetään moduulien koodissa. Kun Event käynnistyy, sitä kuuntelevien observerien logiikka suoritetaan. Eventille on mahdollista syöttää parametreina dataa, jota observer voi hyödyntää toiminnallisuudessaan. Observerien käytössä tulee olla tarkkaavainen, ja välttää kutsumasta esimerkiksi sellaisia metodeja, jotka voivat laukaista event loopin. (Magento 2018.)

```
public function testEventDispatch()
{
   $eventData = null;
   // Code...
   $this->eventManager->dispatch('my_module_event_before');
   // More code that sets $eventData...
   $this->eventManager->dispatch('my_module_event_after',
   ['myEventData' => $eventData]);
}
```

Magento tarjoaa myös kattavat työkalut moduulien testaamiseen. Järjestelmän monipuolisuuden ja laadunvarmistuksen vuoksi testaustukea tarjotaan useasta eri näkökulmasta. Viitekehykset on rakennettu yksikkö-, järjestelmä-, Web Api, integrointi-, staattisille, sekä JavaScript testeille, ja ne on toteutettu modulaarista järjestelmäarkkitehtuuria noudattaen, kuten muukin Magenton järjestelmästä. Yksikkötesteihin on panostettu suuresti mm. rakentamalla tuki PHPUnit ohjelmistokehyksen sisäänrakennettuihin työkaluihin. (Magento 2018.)



Kuvio 4. Testauskehysten sijainti Magento 2 kansiorakenteessa.

4 Testivetoinen kehittäminen

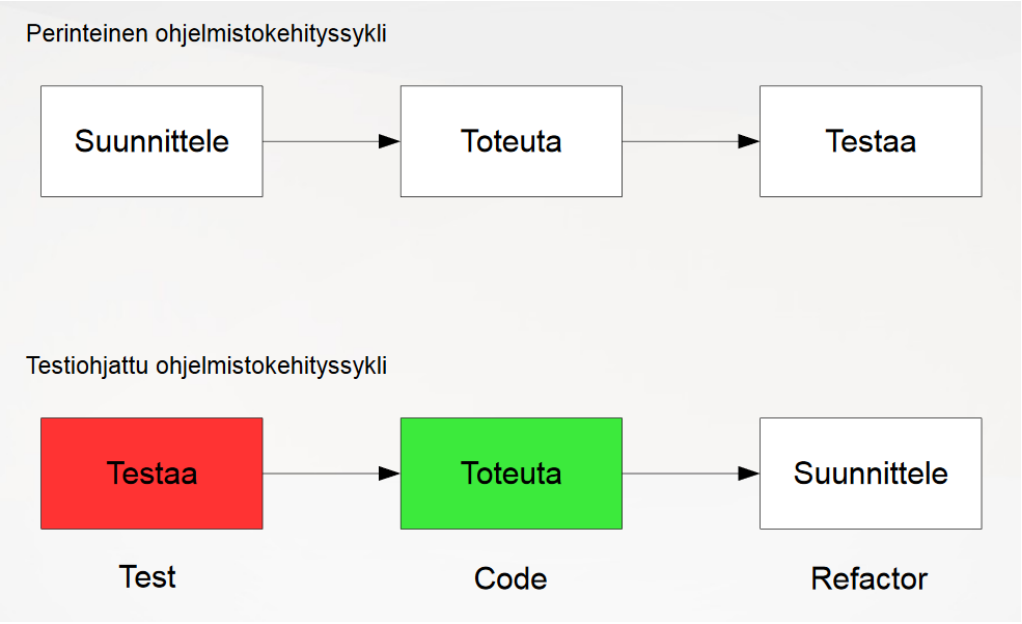
Testaaminen on ollut osa ohjelmistokehitystä sen alkuajoista lähtien. Testaaminen on kehittynyt ajan myötä niin käsitteenä, kuin tekniikoiltaankin. Testaustekniikoiden kehitystä ja tutkimusta ovat ohjanneet testauksen kohteiden ja itse testauksen määritelmän kehittyminen. Gelpern ja Hetzel (1988) jaoittelivat testaustekniikoiden kehittymisen artikkelissaan “The growth of software testing” viiteen eri vaiheeseen:

Ensimmäinen vaihe oli aika ennen vuotta 1956, jolloin testausta ei oltu eritelty vaan se oli osa debugausprosessia. Vaihetta voidaan pitää ohjelmistotestauksen esiasteena. Toinen vaihe oli aika vuosien 1957 ja 1978 välillä, jolloin testauksen päämääränä oli varmistaa, että ohjelma vastaa sille asetettuja vaatimuksia. Kolmas vaihe oli aika vuosien 1979 ja 1982 välillä, jolloin testauksen pääpaino siirtyi virheiden löytämiseen ohjelmasta. Neljäs vaihe oli aika vuosien 1983 ja 1987 välillä, jolloin testauksella pyrittiin löytämään virheitä myös vaatimuksista ja suunnittelusta itse ohjelman lisäksi. Viides vaihe on aika vuoden 1988 jälkeen, jolloin testauksella pyritään ennaltaehkäisemään virheitä vaatimuksissa, suunnittelussa ja ohjelman toteutuksessa. (Gelpern, Hetzel 1988; Luo 2013.)

4.1 Testivetoisen kehityksen määrittely

Testivetoisen kehityksen modernista “uudelleen löytymisestä” voidaan antaa kunnia Kent Beckille, jonka tuotoksena on syntynyt mm. Extreme Programming -kehitysmenetelmä. Moderni TDD näki päivänvalon Extreme Programmingin ja ketterien kehitysmenetelmien suosion kasvun tuloksena.

Vaikka Beck löysi ja sovelsi TDD:n menetelmiä 2000-luvun alussa, juontaa se juurensa 1960-luvun tietojenkäsittelyyn. Aikana, jolloin ohjelmointi suoritettiin reikäkorttien avulla, ohjelmoijilla oli hyvin rajalliset mahdollisuudet tutkia syötettyä dataa virheiden varalta. Eräs kirjattu menetelmä oli merkitä ylös odotettu tulos toiminnosta, joka tietokoneeseen syötettiin. Kun keskusyksikkö tuotti siihen syötetyn ohjelman tulokset, voitiin niitä vertailla ylös kirjattuihin tuloksiin, jolloin mahdolliset eroavaisuudet huomattiin välittömästi. (Barber 2012.)



Kuvio 5. Eroavaisuudet testivetoisen ohjelmointisyklin ja perinteisen ohjelmointisyklin perusteissa (Collin 2010).

Beckin (2002) määrittelemä Testivetoinen kehityssykli etenee yksinkertaisesti esitettynä seuraavalla tavalla:

1. Lisää testi.

   TDD:n syklin mukaisesti jokaisen uuden toiminnallisuuden kehittäminen alkaa testin kirjoittamisella. Kirjoittaaksen testin, ohjelmoijalla täytyy olla selvä kuva halutun toiminnallisuuden määrityksistä ja lopputuloksesta.

2. Aja kaikki testit, joka johtaa uuden testin epäonnistumiseen.

   Ohjelmoija odottaa tämän testin epäonnistuvan, koska testin läpäisemiseen tarvittavaa koodia ei ole vielä olemassa. Ohjelmoija tietää, mitä tehdä saadakseen koodin läpäisemään testin.

3. Tee korjaukset koodiin.

   Tässä vaiheessa ohjelmoija tekee tarvittavat muutokset koodin, jotta aiemmin epäonnistunut testi saadaan läpäistyksi. Tavoitteena on olla kirjoittamatta koodia, mikä ei vaikuta testin läpäisyyn.

4. Aja testit uudelleen ja katso kuinka kaikki testit onnistuvat.

   Mikäli kaikki testit menevät nyt läpi, ohjelmoija tietää koodin toimivan halutulla tavalla. Jos testit epäonnistuvat, on koodia muokattava kunnes uusin testi menee läpi.

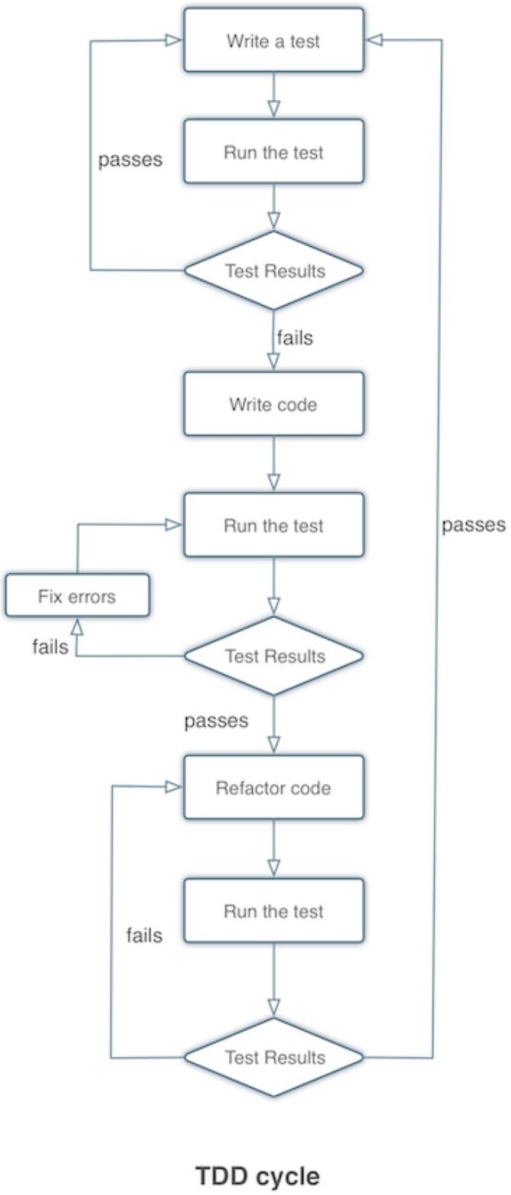
5. Refaktoroi koodi.

   TDD:n aikana syntyvän koodin rakennetta on paranneltava silloin tällöin muuttamatta kuitenkaan sen toiminnallisuutta. Refaktorointiin kuuluu mm. Koodin luettavuus ja ohjelmakomponenttien toiminnan selkeytys. Ohjelmointivirheisiin ei tässä vaiheessa kuitenkaan keskitytä.

TDD:n erikoisuus on kehittämissyklin näennäisesti järjenvastainen toimintamalli: Yksikkötestit kirjoitetaan ennen tuotantokoodin kirjoittamista. Tämä määrittää Test-first komponentin merkityksen testivetoisessa kehityksessä. Järjenvastaiselta

kyseinen toimintamalli voi tuntua siksi, että normaaliksi koettu ohjelmistokehityksen malli seuraa vahvasti Test-last kehitysmallia, jossa yksikkötestaus suoritetaan perinteisesti koodin toteutuksen jälkeen.

TDD kehityssykli koostuu Test-first lähestymismallin lisäksi refaktoroinnista, jonka avulla koodin rakennetta parannetaan sisäisesti ilman muutoksien näkymistä luoduissa testeissä tai ulkoisessa toiminnallisuudessa. Refaktorointi voi sisältää duplikaattien poistoa, koodin luettavuuden parantamista ja väliaikaisten koodintynkien loppuunkirjoittamista. Refaktoroinnin päätytyttyä jokaisen olemassa olevan testin tulisi mennä läpi osoittaakseen tuotantokoodin toiminnallisuuden olevan ennallaan sen rakenteen parantamisen jälkeenkin. (Fucci, Erdogmus, Turhak, Oivo, Juristo 2017.)



Kuvio 6. TDD Kehityssyklin vaiheiden tarkempi analysointi (Ambler 2002–2019).

Barberin (2012) mukaan TDD:n perustana ovat automatisoidut testit, joita ilman kyseistä menetelmää ei voitaisi toteuttaa. Jotta TDD:tä voitaisiin täysin ymmärtää, pitää automatisoidun testaukset perusteet ymmärtää hyvin. Kolme automatisoidun testauksen päätyyppiä, joita kehittäjät useimmiten käyttävät, ovat:

Yksikkötestaus, jonka ideana on yksilöidä pienin mahdollinen testattava pala koodia, ja testata sen toimivuus.

Integrointitestaus, jossa testataan useamman toimivan yksikön tai järjestelmän kyky tuottaa haluttua tulosta yhdessä.

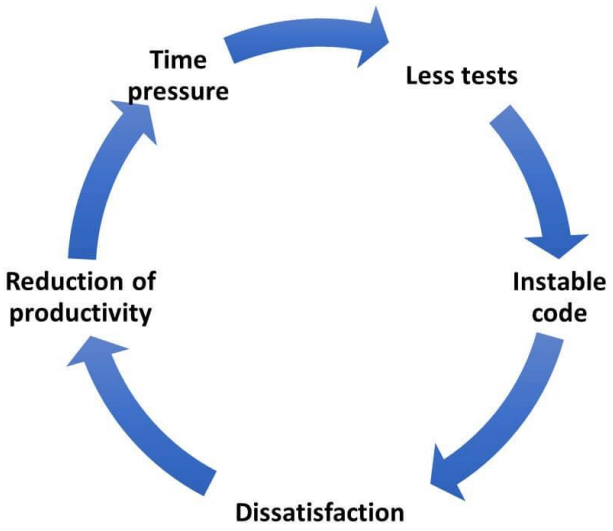
Järjestelmätestaus, jonka tarkoituksena on simuloida oikeaa käyttötilannetta vastaavat olosuhteet, ja testata ohjelman toimivuus kuormituksen alaisena.

Testivetoisessa kehityssyklissä ohjelmoijan ensisijainen tavoite on yksikkötestien ja integrointitestien kirjoittaminen, sillä nämä testit rakentuvat järjestelmävaatimusten ympärille ja varmistavat, että ohjelman eristetyt toiminnallisuudet toimivat yksinään, sekä yhdessä muiden kanssa. Järjestelmätestaaminen jätetään usein ulos TDD:n ensisijaisesta kehityssyklistä, mutta tilanteessa, jossa esimerkiksi kehitettävän kokonaisuuden suorituskyky tulee varmistaa lähtöpisteestä saakka, voidaan järjestelmätestit liittää osaksi TDD:tä. (Barber 2012.)

4.2 TDD:n hyödyt ja haasteet

TDD:llä on tutkitusti useita hyötyjä oikeaan ympäristöön sovellettuna, ja monien asiaan perehtyneiden tutkimuksien tuloksien perusteella se on yksi merkittävimmistä, laadun parantamiseen liittyvistä käytänteistä ketterässä ohjelmistokehityksessä. (Causevic, Punnekkat, Sundmark 2013.)

Eräs positiivinen vaikutus TDD:n käyttöönotossa Beckin (2002, 127–128) mukaan on stressinhallinta. Mitä enemmän stressiä kehittäjä tuntee, sen todennäköisempää on, ettei hän tule testaamaan koodia tarpeeksi. Kun kehittäjä taas tietää laiminlyövänsä testausta, hänen stressitasonsa nousevat entisestään. Tämä johtaa ikävään kierteeseen, josta on vaikea päästä irti. Jos kehittäjä toimii test first menetelmän mukaisesti ja omaksuu testin kirjoittamisen aina ennen koodin tuottamista, hänen stressitasonsa vähenevät. Tämä johtaa jatkossa yhä useampiin testeihin ja miellyttävämpiin työolosuhteisiin.



Kuvio 7. Kuvaus testauksen puutteen ja stressitekijöiden aiheuttamasta kierteestä (Mumah 2018).

Artikkelissa ”Does test-driven development really improve software design quality?”, kirjoittajat totesivat TDD:n edesauttavan paremman tuotantokoodin syntymistä. Artikkeli osoittaa TDD:n menetemillä tuotetun koodin olevan jaoteltuna pienempiin, hyvin suunniteltuihin osioihin, joita on helpompi testata. Tällaisen koodin koettiin olevan helpompaa ylläpitää ja muuntaa tulevaisuudessa. (Janzen, Saiedan 2008; Barber 2012.)

Haasteena TDD:n käyttöönotossa voi olla saada jokainen kehitystiimin jäsen hyväksymään testivetoinen kehityskulttuuri. Esimiehet voivat kokea haaskaavansa arvokkaita resursseja, kun jatkuva testaaminen ja refaktorointi ei itsessään laajenna koodin toiminnallisuutta. Kehittäjien näkökulmasta voi haastavaa aloittaa testivetoisen kehittämisen täsmällisten säädösten noudattaminen. Argumenttina tällaisissa tapauksissa voidaan kuitenkin käyttää testivetoisen kehityksen tuottamien testien parempaa laatua, jolla on suora vaikutus myös siihen liittyvän tuotantokoodin laatuun. (Madeyski 2010.)

5 Yksikkötestaus

Yksikkötestauksen tavoitteena on yksilöidä ohjelman osat, kuten luokat ja metodit omiin testattaviin yksiköihinsä, joka mahdollistaa virheiden tarkistuksen ja ennaltaehkäisemisen pienemmässä ja tarkemmassa mittakaavassa. Nämä yksittäiset testit antavat usein tarkkoja tuloksia halutuista kohteista, mikä tuottaa useita etuja ohjelmoijalle.

Yksikkötestaus toimii parhaiten silloin, kun testit voidaan ohjelmoida niin, että ne on mahdollista ajaa yksin ilman muita ohjelman osia. Testin tekijän täytyy tuntea testattava osanen kokonaisuudessaan ja tämän takia testin tekee yleensä ohjelman osan suunnittelija. Testit ovatkin yleensä vain tuloksia ohjelman osiosta. Testien tyylit vaihtelevat ohjelman osion mukaan, koska yksikkötestaus on juurikin yhden palasen testausta. Tämä mahdollistaa nopean virheraportoinnin ohjelman tuottamasta virheestä. (Zandstra 2008.)

Yksikkötestaus alkaa valmiin testisuunnitelman sekä vaatimusten muuntamisesta testitapauksiksi. Testitapaukset ovat ryhmä testisyötteitä, joiden mallia ohjaa suurelta osin tapaukseen kytkeytyvän ohjelmiston määritykset ja käyttötarkoitus. Yksikkötestauksessa testitapaukset määritellään varmistamaan, että jokainen yksikkö noudattaa kaikkia mallin määrityksiä. Perinpohjaisen testitapauksen tulisi sisältää positiivista testausta, joka varmistaa yksikön tuottavan haluttua tulosta, sekä negatiivista testausta, joka tarkistaa ettei yksikkö tee mitään ylimääräistä. Vaikka testitapaukset vaihtelevat riippuen testattavasta kokonaisuudesta, tulisi jokaisen testitapauksen Shirastavan ja Jainin (2011) mukaan sisältää ainakin neljä keskeistä osaa:

- Ohjeistus (statement) yksikölle, joka on testitapauksen aloituspiste.
- Syöte (input) yksikölle, mukaanlukien ulkoiselta datalta luettu syöte.
- Mitä varsinaisesti testataan, yksikön toiminnallisuuden näkökulmasta.

- Testitapauksen oletettu lopputulos.

5.1 Yksikkötestauksen hyödyt

Silloin tällöin esiin nousee kysymyksiä yksikkötestamisen hyödyistä ja sen tehokkuudesta osana ohjelmointiprosessia. Glavas (2018) luettelee yksikkötestauksen tärkeimpiä piirteitä, jotka on kirjoitettu pitäen mielessä juuri Magento 2 kehitystä.

- Ongelmakohdat paikannetaan nopeasti

Yksikkötestauksen avulla ongelmat koodissa paikannetaan aikaisin kehityssyklin aikana. Lisäksi testien kirjoittaminen pakottaa kehittäjän ajattelemaan syötteiden avulla, sekä varautumaan mahdollisesti esiintyviä virheitä varten. Tämä johtaa tarkemmin kirjoitettuun yksikön toiminnallisuuteen.

- Toiminnallisuuden muokkaaminen helpottuu

Yksikkötestit mahdollistavat koodin refaktoroinnin ja järjestelmäkirjastojen päivittämisen jälkeenpäin varmistaen aina, että koodi toimii halutulla tavalla. Mitä tarkemmin yksikkötestit on kirjoitettu, sen helpommaksi koodin muokkaaminen jälkeenpäin tulee.

- Ohjelman rakenne paranee

Kun yksikkötestejä kirjoitetaan testivetoisesti, on vaikutus ohjelman designiin merkittävä. Tämä lähestymistapa saa kehittäjän ajattelemaan yksikköä sen suorittaman lopputuloksen näkökulmasta ennen toiminnallisuuden luontia, ja täten luomaan parempaa ohjelmallista designia. (Glavas 2018.)

5.2 PHPUnit kehys

PHPUnit ohjelmistokehys tarjoaa erinomaiset mahdollisuudet hyödyntää yksikkötestejä Magenton moduulikehityksessä. Siihen kuuluvaa PHPUnit\Framework\TestCase -luokkaa hyödyntäen kehittäjä määrittelee testitapaukselle syötettävät tiedot ja tulosten odotusarvot. Odotusarvoksi voidaan asettaa halutunlainen tietotyyppi, mm. alla näkyvän esimerkin mukaisesti Exception (virheenkäsittelyä varten luotu olio). Luokka useimmiten sisältää setUp() -metodin, joka määrittelee testiolion esiasetukset ja tearDown() -metodin, joka ns. ”siivoaa” testidatan testattavasta luokasta kun testit on suoritettu. (Trucchia, Romei 2010, 66–70.)

```
<?php
use PHPUnit\Framework\TestCase;
class ExceptionTest extends TestCase
{
    public function testException()
    {
        $this->expectException(InvalidArgumentException::class);
    }
}
```

PHPUnit tarjoaa testaajalle myös muita hyödyllisiä yksikkötestaustyökaluja, kuten Sijaisoliot (Mock -ja Stub -objekti), jotka voidaan luoda simuloimaan haluttua luokkaa testauksen aikana. Tämä on hyödyllistä tilanteissa, jossa testataan muista luokista riippuvaisia toiminnallisuuksia. Suoritettavia testejä voidaan myös rajata tilanteissa, jossa vain osa luoduista testeistä on tarpeellista testata. PHPUnit tarjoaa myös mahdollisuuden testituloksien tallentamiseen lokeihin. (Bergmann, 2005–2011.)

Magenton virallisessa testausdokumentaatiossa ohjeistetaan tarkkailemaan luokkien kokoa, käyttämään riippuvuuksia harkiten, sekä hyödyntämään rajapintoja, kun kirjoitetaan yksikkötestejä. Myös luokkien sisältämien funktioiden sisältöön pitäisi

kiinnittää erityistä huomiota. Lyhyitä funktioita on helpompi lukea, ja niiden tehtävä on usein selkeämpi kuin suurien, monimutkaisten funktioiden. Testattavaa koodia kirjoittaessa täytyisi funktiot pyrkiä pitämään suoraviivaisina, sekä jaoitella erilliset pienet tehtävät, kuten olioiden arvojen hakeminen useampiin funktoihin, joilla on selkeä tehtävä testattavassa kokonaisuudessa. (Magento 2018.)

6 Integrointitestaus

Integrointitestaus on moduulien tai järjestelmän palasten välisien rajapintojen testaamista. Se keskittyy tiedonkulun toimivuuteen järjestelmässä, joka rakentuu pienemmistä osakokonaisuuksista, joista jokaisella on oma tehtävänsä. Testaussuunnitelu voidaan normaalisti aloittaa, kun arkkitehtuurisuunnitelun tuotoksena syntyneet ohjelmistomoduulien kuvaukset ja niiden välisten yhteyksien määrittelyt ovat valmistuneet. Ensimmäisenä suoritetaan yksittäisten moduulien testaaminen, jonka tuloksien perusteella voidaan aloittaa integrointitestaus. (Rauhala 2010.)

Integrointitestien suunnitelijalla on oltava laaja ymmärrys testattavan kokonaisuuden eri palasten yhteyksistä, sekä rajapintojen odotetusta toiminnasta. Kun testaaja havaitsee virheen integrointitestissä, se raportoidaan ohjelmistosuunnittelijalle, joka tekee ohjelmaan korjaavat toimenpiteet. Tämän jälkeen testausprosessi aloitetaan usein alusta varmistamaan, ettei yksittäisten moduulien testeissä ilmene uusia ongelmia. (Dooley 2011, 189-193.)

Magento 2:n Integrointitesteissä hyödynnetään yksikkötestien lailla PHPUnit ohjelmistokehystä, mutta yksittäisen moduulin toiminnan sijasta testataan eri moduulien välistä tiedonkulkua, moduulien integroitumista Magenton järjestelmään, sekä luokkariippuvuuksien ja -laajennuksien konfigurointia. Testaamista varten on luotava ja konfiguroitava oma tietokanta, sillä integrointitestit tyhjentävät testattavan kannan datasta testauksen päätteksi. (Magento 2018.)

Kopp (2016) ohjeistaa hyödyntämään Magenton integrointitestejä testivetoisessa kehityssyklissä esimerkiksi kun luodaan uudesta moduulista runko, ja halutaan testata sen konfigurointien toimivuus järjestelmässä ennen varsinaisten yksikkötestien luontia. Integrointitestin avulla voidaan mm. varmistaa, että uusi moduuli löytyy Magenton komponenttirekisteristä.

```
<?php
use Magento\Framework\Component\ComponentRegistrar;
use PHPUnit\Framework\TestCase;

class ModuleRegisterTest extends TestCase
{
    /** @var string */
    private $moduleName = 'TestVendor_TestModule';
    public function testModuleRegistration()
    {
        $componentRegistrar = new ComponentRegistrar();
        $this->assertArrayHasKey($this->moduleName,
        $componentRegistrar->getPaths(ComponentRegistrar::MODULE));
    }
}
```

Magento 2:n integrointitestikehys tarjoaa keinon suorittaa integrointitestejä kehittäjän määrittelemän testidatan avulla käyttäen fikstuureja (fixture). Nämä ovat PHP skriptejä, jotka asettavat määriteltyä dataa testikantaan. Fikstuureja voidaan luoda joko erillisiin tiedostoihin, tai metodien sisään, ja niitä kutsutaan testitiedostossa @magentoDataFixture -annotaation avulla. (Magento 2018.)

```
/**
 * @magentoDataFixture <script_filename>|<method_name>
 */
```

7 Tutkimuksen toteutus

Tämän tutkimuksen tavoitteena on löytää toimiva kehitysprosessi Magento alustalla testivetoista kehitysmenetelmää käyttäen. Tutkimuksen toteuttamisen ensimmäinen vaihe on kehitysympäristön ja tutkimuksessa tarvittavien työkalujen asettaminen valmiuteen kehittämisprosessia varten.

Tutkimuksessa käytetyn Magento 2 -ympäristön tulee olla tyhjä eli ”puhdas”, jotta vältettäisiin ydinjärjestelmään kuulumattoman koodin sotkeutuminen kehitettävään kokonaisuuteen. Tämä tarkoittaa käytännössä uutta Magento asennusta, johon ei olla asennettu ulkopuolisista lähteistä peräisin olevia moduuleja.

Tutkimuksen dokumentaatiosta ollaan rajattu pois Magento 2:n ja muiden työvälineiden asentaminen sekä konfigurointi käyttövalmiiksi ja oletetaan, että pohjatyö moduulikehitystä varten on ennalta tehty. Tutkimuksen aikana luodun moduulin luonti- ja kehitysvaiheet dokumentoidaan, sillä tarkoituksena on luoda kehitysprosessia kuvaava ohjeistus, jonka havaintoja voidaan hyödyntää käytännössä.

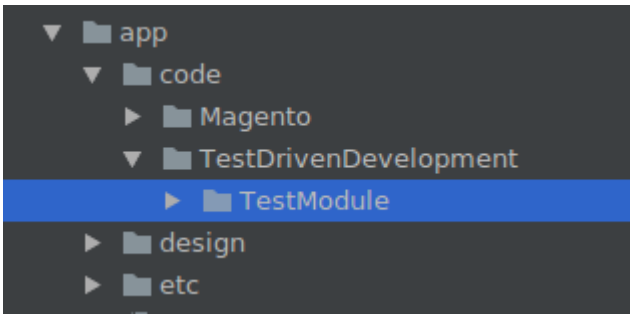
Tutkimuksessa kehitettävän moduulin lopputulokselle ei ollut tarkkoja määrityksiä, vaan tärkeintä oli saada hyödynnettyä testivetoista kehitystä Magenton yleisten moduulikehitystekniikoiden kanssa. Tarkoituksena on siis tuottaa ns. geneerinen moduuli testivetoisesti ja säilyttää sekä koodi, että testit tarkastelua ja jatko-opiskelua varten.

Moduulin geneerisenä toiminnallisuutena käytetään kuvitteellisen ”Created Orders” attribuuttin tallentamista asiakastauluun, kun rekisteröitynyt asiakas luo tilauksen.

Attribuutti pitää kirjaa rekisteröityneen asiakkaan luomien tilausten määrästä, ja määrän arvoa kasvatetaan aina, kun kyseinen asiakas luo uuden tilauksen.

7.1 Moduulin luonti ja integrointitestit

Moduulikehityksen ensimmäinen vaihe on luoda moduulikansiot ja tiedostot, jotka rekisteröivät moduulin Magenton järjestelmään. Moduulikansio luotiin app/code kansion alle. Magento 2 moduulin luominen onnistuu tekemällä ensin pohjakansio vendorille, joka on moduulin tarjoaja (esim. Magento). Moduulin vendoriksi nimettiin TestDrivenDevelopment, ja moduulin nimeksi annettiin TestModule.



Kuvio 8. Moduulin sijainti app/code kansiopolussa.

Moduulikansion luomisen jälkeen kirjoitettiin integrointitestit, jotka tarkistavat onko moduuli aktivoitunut järjestelmään. Integrointiestien luomiselle on omat edellytyksensä; Testejä varten luotiin niille omistettu tietokanta, sillä Magenton integrointitestit hävittävät kannasta kaiken datan ennen testien suoritusta. Magenton dokumentaatiossa suositellaan luomaan testikannalle määritelty käyttäjä, jolla ei ole käyttöoikeuksia muihin tietokantoihin.

```
CREATE DATABASE magento_integration_tests;

GRANT ALL ON magento_integration_tests.* TO 'm2_test_user'@

'localhost' IDENTIFIED BY 'passw0rd1234';
```

Kantakonfiguraatiot luotiin install-configmysql.php tiedostoon, joka sijaitsee Magento 2:n integrointitestikehyksen kansiorakenteessa. PHPUnit konfiguraatiot säädettiin siten, että ne ajavat vain app/code kansion alle luodut integrointitestit. Tämä tehtiin ajan säästämiseksi.

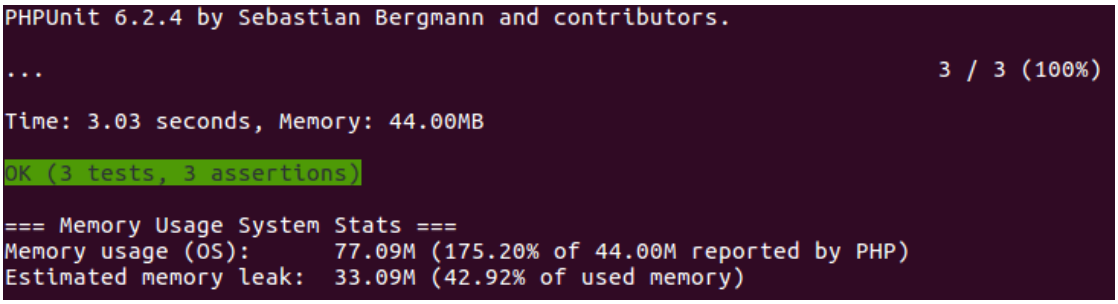
```
<testsuites>
        <testsuite name="Module Integration Tests">
            <directory>../../../app/code/*/*/Test/Integration
             </directory>
        </testsuite>
</testsuites>
```

Integrointitestit luodaan moduulin Test/Integration kansion alle. Ensimmäisten testien tarkoituksena on epäonnistua, jotta voidaan luoda koodia läpäisemään testit. Jokainen moduuli vaatii kaksi tiedostoa toimiakseen: moduulin rekisteritiedoston **registration.php** ja **module.xml** -tiedoston, joka syöttää järjestelmälle moduulin nimen ja versionumeron. Näiden tiedostojen luonnin jälkeen integrointitestit läpäistiin onnistuneesti.

```
public function testModuleIsEnabled() {
        $deploymentConfig = $this->objectManager->create(
        DeploymentConfig::class);

        /** @var ModuleList $moduleList */
        $moduleList = $this->objectManager->create(
        ModuleList::class, ['config' => $deploymentConfig]);
        $this->assertTrue($moduleList->has($this->module));
}
```

Integrointitestien ajaminen suoritettiin joko PHPStrom -käyttöliittymässä tai komentorivin (kuvio 9) kautta. Testiajon tuloksina esitetään suoritettujen testien ja testeissä tehtyjen vertailujen määrä.



Kuvio 9. Integrointitestien suoritus komentorivin kautta.

Moduulin toimivuuden testaamisen jälkeen luotiin testit moduulin toiminnallisuutta varten. Ennen toiminnallisuuden luontia oli suunniteltava paras tapa luoda laajennos tilauksen luontiin ja tallentamiseen. Lyhyen selvittämisen jälkeen päädyttiin luomaan toiminnallisuus käyttäen Event observer mekaniikkaa, sillä Magenton ydinkoodissa laukaistaan ’sales_order_save_afer’ event tilauksen tallentamisen yhteydessä. Observeria luodessa tuli pitää mielessä koodin yksinkertaisuus ja riippuvuudet, jotta toiminnallisuus ei aiheuttaisi ydinkoodin suorituksessa ongelmia.

Observerin konfiguraation tarkistamista varten luotiin integrointitesti, joka varmistaa, että määritetty observeri (testmodule_order_save_after_observer) kuuntelee ’sales_order_save_after’ eventtiä.

```
private $observerName = 'testmodule_order_save_after_observer';


public function testOrderSaveAfterObserverConfigurations() {
        /** @var ConfigInterface $observerConfig */
        $observerConfig = ObjectManager::getInstance()->create(
       ConfigInterface::class);
```

```
        $observers = $observerConfig->getObservers(
        'sales_order_save_after');
        $this->assertArrayHasKey($this->observerName, $observers);
}
```

Testin epäonnistumisen jälkeen luotiin events.xml tiedosto, johon määriteltiin observerin konfiguraatiot. Tämän seurauksena observerin integrointitestin suoritus meni onnistuneesti läpi.

Moduulille luotiin integrointitesti, joka testaa observerin suorittamista testidatan kanssa, sekä ”Created Orders” -attribuutin tallentumista. Testidata haettiin fikstuurista, joka luotiin moduulin Test/Integration/_files kansiopolun alle. Fikstuuriin syötettiin asiakasdataa, jota käytettiin luomaan testiasiakas attribuutin tallentumisen testaamista varten.

7.2 Yksikkötestit ja koodin toteuttaminen

Moduulin yksikkötestit sijaitsevat Test/Unit kansion alla. Testitapaukset luotiin observerille, sekä luokalle, joka suorittaa varsinaisen attribuutin arvon laskennan ja tallentamisen. Observerille luotujen yksikkötestien tarkoituksena oli varmistaa, että oikeilla parametreillä haetut OrderInterface -oliot laukaisevat metodin, joka laskee ja asettaa ”Created Orders” -attribuutin arvon.

Yksikkötestien ensivaiheena luotiin setUp() metodiin testioliot luokista, joita tarvitaan toiminnallisuuksien suorittamiseen. OrderInterface -rajapinnasta luotiin mock -olio, jolle annettiin kuvitteellisen ”orderNumIsSet” -attribuutin hakumetodi. Tämä tehtiin, jotta observerissa voitaisiin tarkastaa onko kyseisen tilauksen käsittelyn aikana asetettu ”Created Orders” attribuuttia asiakkaalle. Sen jälkeen luotiin eri syötteillä olevat testimetodit observerin suoritusta varten.

Alla esimerkki testistä, jossa oletetaan tilauksen asiakkaan olevan rekisteröitynyt, mutta ”orderNumIsSet” hakumetodin palauttavan true arvon, jolloin uutta arvoa

”Created Orders” -attribuutille ei enää lasketa. Tämä on mahdollista testata expects() metodilla, joka asettaa testioliolle jonkin odotteen, kuten kutsuttavan metodin.

```
public function testCreatedOrdersNumAttributeIfAlreadySet(){
        $this->observerMock->method('getEvent')
            ->willReturn($this->eventMock);
        $this->eventMock->method('getData')
            ->with('order')
            ->willReturn($this->order);
        $this->order->method('getCustomerIsGuest')
            ->willReturn(false);
        $this->order->method('getOrderNumIsSet')
            ->willReturn(true);
        $this->calculator->expects($this->never())
            ->method('getCreatedOrdersValue');
        $this->orderSaveAfterObserver->execute($this->observerMock);
}
```

Moduulin toiminnallisuudet päätettiin eriyttää selkeyden vuoksi, josta johtuen luotiin erillinen luokka, joka sisältää logiikan ”Created Orders” -attribuutin arvon laskemista varten. Testitiedostoon luotiin testit mm. attribuutin tallentamista ja metodin virheenkäsittelyä varten. setUp() metodissa luotiin CustomerInterface mock -olio, jolle määriteltiin metodit ’getCreatedOrders’ ja ’setCreatedOrders’, jotta attribuutin tallentamista voitaisiin simuloida testin aikana.

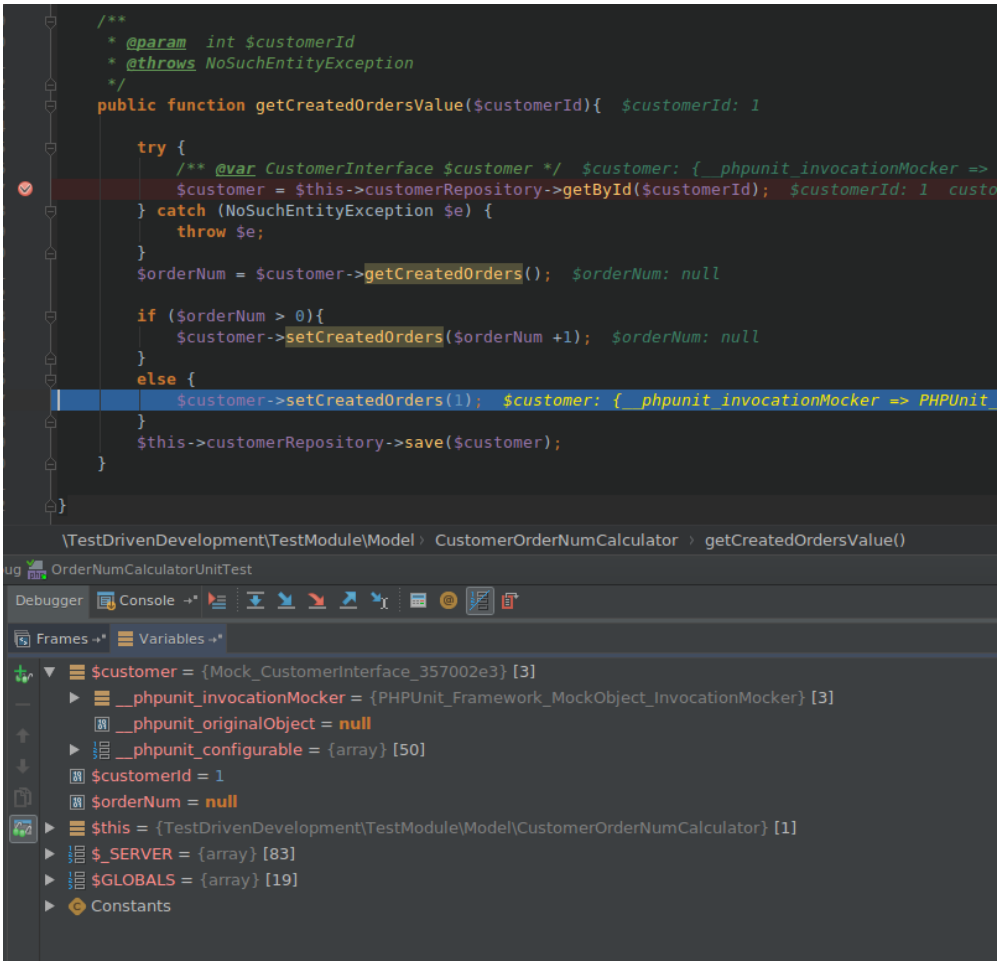
```
$methodsToStub = array_merge(
        get_class_methods( CustomerInterface::class),
        ['getCreatedOrders', 'setCreatedOrders']
);
```

```
$this->customerInterfaceMock = $this->getMockBuilder(
CustomerInterface::class)
        ->setMethods($methodsToStub)
        ->disableOriginalConstructor()->getMock();
```

Yksikkötestien ja testit läpäisevän koodin kirjoitus eteni testivetoisen kehitysperiaatteen mukaisesti pienissä sykleissä. Tarkoituksena oli pitää kokonaisuuden toimivuus hallinnassa koko prosessin ajan. Varsinaisia luokkia kirjoittaessa testeissä huomattiin välillä puutteita, joka johti uusien testien kirjoittamiseen ja testitiedostojen refaktorointiin. Lopullista toteutusta tehtäessä huomattiin kuinka tärkeä elementti suunnittelu on testivetoisessa kehitysprosessissa. Tilanteissa, joissa jokin Magenton osa-alue ei toiminut kuin oli ajateltu, täytyi testejä refaktoroida uusien tarpeiden mukaan. Tietämys Magenton komponenttien toiminnasta ja PHPUnit syntaksista oli testien suunnittelussa hyvin tärkeässä roolissa.

7.3 Ongelmat ja löydökset

Suurimpaan kompastuskiveen törmättiin kirjoittaesssa integrointitestiä, joka laukaisee luodun observerin ja tarkistaa ”Created Orders” -attribuutin tallentumisen asiakkaalle. Integrointitesteillä testataan moduulien välistä tiedonsiirtoa, eikä niinkään yhden yksikön toimintaa, jolloin kuviteltujen attribuuttien hakeminen ja tallentaminen ei onnistunutkaan samalla tavoin kuin yksikkötesteissä. Tällaisia integrointitestejä onkin mahdollista suorittaa vasta kun määritellyt EAV attribuutit on luotu tietokantaan ja määritelty oikean entiteetin (esim. asiakas) alaisuuteen. Lopullinen attribuuttia testaava integrointitesti jätettiin tämän seikan vuoksi TDD moduulissa suorittamatta.



Kuvio 10. Xdebug työkalun käyttö PhpStorm käyttöliittymässä.

Yksikkö- ja integrointitesteissä suureksi avuksi osoittautui Xdebug työkalu, joka mahdollistaa koodin debugaamisen rivi riviltä testien suorittamisen aikana (kuvio 10). Työkalua käytettäessä voitiin tarkastella muuttujien ja olioiden käyttäytymistä ja paikantaa tilanteet, joissa virheitä saattaa syntyä.

Xdebug täytyy konfiguroida PhpStormiin erikseen, ja tämä onnistui PHPStorm käyttöliittymästä. Debug työkalun koetaan soveltuvan erinomaisesti TDD:n syklin vaiheeseen, jossa luodaan koodia testien läpäisemiseksi, tai testejä ja koodia refaktoroidaan.

PHPUnitin eräs hyödyllinen ominaisuus oli sen koodikattavuus (Code Coverage) työkalu, joka luo XML tai HTML raportin kuvastamaan testien kattamia koodirivejä

viimeksi suoritetussa yksikkötestien ajossa. Työkalu pohjautuu PHPUnitin php-code-coverage komponenttiin, ja se hyödyntää myös Xdebugin kattavuusanalyysiä. Kattavuuden testaaminen koetaan hyödylliseksi varsinkin TDD:n kehityssyklin aloittaville kehittäjille.

HTML muotoisten kattavuustulosten tallentamiselle määritellään kansio phpunit.php tiedostossa, ja tulokset saatiin auki tarkastelua varten selaimessa. Alla on esimerkki CustomerOrderNumCalculator -luokan kattavuustulosten yleiskuvasta:

	Code Coverage								
	Classes and Traits			Functions and Methods			Lines		
Total		0.00%	0 / 1		50.00%	1 / 2	CRAP	90.91%	10 / 11
CustomerOrderNumCalculator		0.00%	0 / 1		50.00%	1 / 2	4.01	90.91%	10 / 11
__construct					100.00%	1 / 1	1	100.00%	2 / 2
getCreatedOrdersValue					0.00%	0 / 1	3.01	88.89%	8 / 9

Kuvio 11. HTML raportissa esitetty testien kattavuustulos moduulin luokan metodeista.

Raportista huomattiin yhden rivin testauksen puuttuminen getCreatedOrderValue -metodista. Rivi esitetään myös punaisella värillä metodin tarkemmassa analyysissä (kuvio 12).

```
33      public function getCreatedOrdersValue($customerId){
34
35          try {
36              /** @var CustomerInterface $customer */
37              $customer = $this->customer->getById($customerId);
38          } catch (NoSuchEntityException $e) {
39              throw $e;
40          }
41          $orderNum = $customer->getCreatedOrders();
42
43          if ($orderNum > 0){
44              $customer->setCreatedOrders($orderNum +1);
45          }
46          else {
47              $customer->setCreatedOrders(1);
48          }
49          $this->customer->save($customer);
50      }
```

Kuvio 12. Kattavuustulos getCreatedOrdersValue -metodista osoittaa testaamattoman rivin koodissa.

Rivi saatiin vihreäksi luomalla testi, jossa oletetaan getCreatedOrders() -metodin palauttavan arvon 1. Testit ajettiin uusiksi ja tämän jälkeen raportista voitiin tarkastella rivin värin vaihtuneen (kuvio 13).

```
41          $orderNum = $customer->getCreatedOrders();
42
43          if ($orderNum > 0){
44              $customer->setCreatedOrders($orderNum +1);
45          }
46          else {
47              $customer->setCreatedOrders(1);
48          }
```

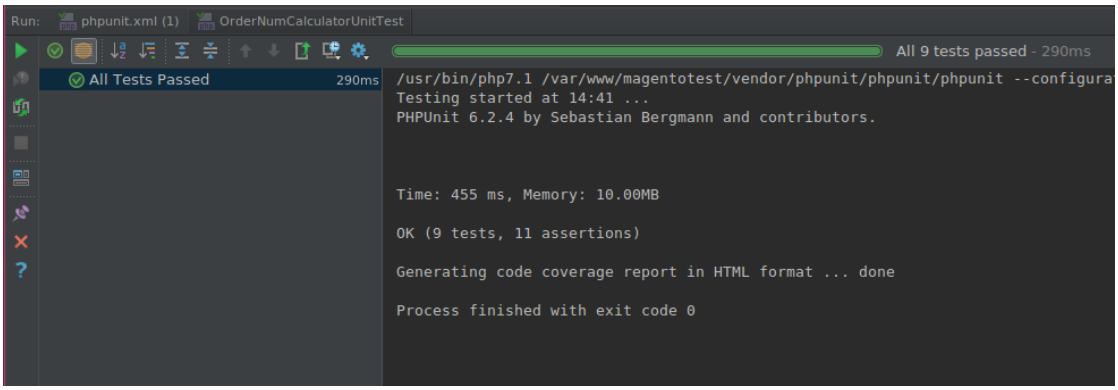
Kuvio 13. Uusi kattavuustulos osoittaa testikattavuuden parantuneen.

Kattavuusraportit koetaan hyödylliseksi varsinkin silloin, kun luodaan monimutkaisia toiminnallisuuksia. Testivetoisessa kehityssyklissä pyritään toimintamalliin, jossa koodirivejä ei syntyisi ilman niitä vaativaa testiä, mutta useita muuttujia ja tarkistuksia sisältävissä toiminnallisuuksissa tätä voi usein olla vaikeaa välttää.

Kun testejä ajettiin PHPStromin käyttäliittymän kautta, oli helposti valittavissa ajetaanko normaali testiajo, debuggausajo vai kattavuustulokset mittaava testiajo (kuvio 14). Testattava tiedosto oli mahdollista valita samasta pikavalikosta.

Kuvio 14. Testiajojen pikavalikko PhpStorm käyttöliittymässä.

Kehitysprosessin lopussa testattiin kaikki yksikkö- ja integrointitestit ajamalla phpunit.php tiedostot dev/tests/unit ja dev/tests/integration kansiopolkujen alta. Kaikkien suunniteltujen testien mennessä läpi onnistuneesti (kuvio 15), koetaan kehitysvaiheen olevan refaktorointivaiheessa tai päätöksessä.



Kuvio 15. Moduulin yksikkötestit ajettu ilman virheitä.

8 Johtopäätökset

Tässä luvussa tiivistetään tutkimuksen havaintoja ja vastataan alussa esitettyihin tutkimuskysymyksiin. Tutkimuksen aikana oli tarkoitus kartoittaa testivetoisen kehityksen mahdollisuudet Magento 2 -alustassa ja löytää näistä sopivat ratkaisut Solteqin Magento -kehitystiimin moduulikehitysprosessiin.

Mitä on testivetoinen kehitys ja mitkä ovat sen työvaiheet?

Testivetoinen kehitys kuvastaa terminä kehitysprosessia, joka aloitetaan luomalla testit kehitettävää kokonaisuutta varten. Varsinainen koodi luodaan läpäisemään testit, jonka jälkeen suoritetaan tyypillistesti koodin refaktorointi ja aloitetaan sykli alusta. Testivetoinen kehitys etenee pienissä sykleissä, joka helpottaa refaktorointia ja kokonaisuuden hallintaa.

Mitä vaatimuksia testivetoinen kehitys tuottaa moduulikehitysprosessin eri vaiheille?

Prosessin suunnitteluun on käytettävä runsaasti aikaa. On suunniteltava millaisilla ratkaisuilla tehdään mahdolliset ydinkoodin laajennokset ja mitä Magento 2 -komponentteja on tarkoitus käyttää lopullisessa moduulissa. Mikäli moduulissa on tarkoituksena hyödyntää kehittäjän luomia attribuutteja, olisi ne hyvä konfiguroida järjestelmään valmiiksi ennen kehitystä, jotta niiden käyttötapa voidaan ottaa suunniteluun mukaan.

Millaisilla työkaluilla ja miten testivetoista kehitystä kannattaa hyödyntää Magento 2 -alustassa?

Testivetoisen kehitysprosessin aloittamiseen tulisi valmistautua seuraavilla tavoilla:

- PHPUnit testikehys tulee asentaa ja konfiguroida käyttöön PHPStrom projektissa. Tähän tarjotaan opinnäytetyön liitteissä ohjeistus (Liite 1).

- PHPUnit testien syntaksiin tulisi tutustua ennen testivetoisen kehittämisen aloittamista. PHPUnit toimii ehkäpä tärkeimmässä roolissa testivetoisen moduulikehitysprosessin käyttöönotossa, joten ohjelmointikehyksen tarjoamat ominaisuudet on tärkeää ymmärtää.

- Xdebug työkalun ja koodin kattavuusanalyysien käyttö testien suorittamisen yhteydessä koettiin hyödylliseksi, joskaan ei pakolliseksi tekijäksi. Kehittäjät voivat halutessaan ottaa työkalut käyttöön kehitysprosessin aikana.

- Integrointitestaus on hyvä lähtökohta testivetoisen moduulikehityksen aloittamiselle, sillä sen avulla voidaan lähteä luomaan toiminnallisuutta moduulin komponenttien rakentamisen ja moduulien välisen tiedonsiirron näkökulmasta. Jos kehittäjällä on lähtöpisteessä on laaja näkemys siitä kuinka moduulin osien tulee keskustella Magenton komponenttien kanssa, säästytään yksikkötestien luontivaiheessa suurimmilta ongelmilta.

Kehitystutkimuksen havaintojen pohjalta voidaan todeta testivetoisen kehittämisen olevan mahdollinen osa Magento 2 -moduulikehitysprosessia, mutta sen käyttöönotto tulee vaatimaan kehittäjiltä työkaluihin perehtymistä ja kehityssyklin sisäistäminen vaatii aikaa. Kehittäjien on myös varattava lisäaikaa testien suunnittelulle ja toteutukselle, mutta testivetoisen syklin sisäistämisen myötä nämä tuntuvat luontaiselta osalta kehitysprosessia.

Laadunhallinnallisiin kysymyksin on haastavaa luoda johtopäätöksiä tutkimuksen aikana suoritetun työn perusteella. Tällaiset tulokset vaatisivat käytännön testausta menetelmää hyödyntävältä kehitystiimiltä. Käytännön testaaminen tulee vaatimaan resurssien järjestämistä menetelmän käyttöönottoa varten, ja näiden resurssien määrää on vaikeaa arvioida. Ensiaskel käyttöönottoa varten voisi olla esimerkiksi se, että osa tiimin kehittäjistä ottaa kehitysmenetelmän käyttöönsä ja havainnoi sen vaikutuksia omaan sekä tiiminsä tuottavuuteen. Testivetoisen kehitysmenetelmän käyttöönotosta ohjeistus liitteissä (liite 2).

9 Pohdinta

Tutkimuksen aikana toteutettu moduuli jäi tulosten osalta melko pieneksi eikä kaikkia erilaisia PHPUnitin tarjoamia ominaisuuksia saatu sisällytettyä työhön.

Tärkeintä tutkimuksen kannalta oli soveltaa testivetoisen kehityksen tekniikoita Magento 2 -moduulikehitykseen, jossa onnistuttiin hyvin. Opinnäytetyön aikana saatiin kartoitettua myös uusia kehitysprosessissa hyödynnettäviä työkaluja, mikä oli positiivinen lisä tutkimustuloksiin.

Tutkimuksen aikana luotiin hyödyllisiä päätelmiä tutkimuskohteen elementeistä kuten testityökaluista ja testien suunnitelusta. Testivetoisen kehitysmenetelmän käytön vaikutus moduulien kehitysprosessin kestoon ei selvinnut tutkimuksen aikana, sillä vertailukohdetta ei ollut. Käytön vaikutukset kehitykseen kestoon koetaan tärkeäksi seikaksi jatkotutkimuksia ajatellen. Myös laadullista vaikutusta olisi järkevää mitata vertailemalla erilaisten kehitysprosessien tuotoksena syntyneitä moduuleja.

Lisätutkimusta voitaisiin tehdä myös testivetoisen kehityksen ja Magento 2:n järjestelmätestauskehyksen yhteensopivuudesta projektikehityksessä, sekä siitä kuinka testivetoista kehitystä voitaisiin hyödyntää Magenton JavaScript testauskehyksessä. Käyttäjäpinnan (Frontend) kehitys jäi tämän tutkimuksen ulkopuolelle, mutta on kuitenkin tärkeä osa Magento 2 -moduulikehitystä.

Tutkimuksessa löydettyjen menetelmien koetaan olevan yleiskäyttöisiä Magento 2 moduulikehityksessä, eikä esimerkiksi projektikohtaisia rajoitteita tunnistettu pysyttäessä tutkimuksessa esitetyissä työkaluissa. Työkalujen käyttöä tulee soveltaa kehitystehtävien mukaan, mutta tämä jää menetelmää hyödyntävän kehitystiimin tehtäväksi.

Opinnäytetyön koetaan täyttäneen tarkoitusperänsä ja tuottaneen suunniteltua hyötyä. Tutkimustuloksena onnistuttiin tuottamaan toimiva menetelmä jota voidaan käyttää kehitystiimin Magento 2 -projekteissa. Työssä selvitettiin myös menetelmän vaikutuksia työskentelyprosessiin, mutta jotta saataisiin täydellinen kuva menetelmän tuomista positiivisista ja negatiivisista vaikutuksista, sitä pitäisi päästä testaamaan käytännössä ja erilaisissa kehitysprojekteissa.

Lähteet

Ambler, Scott. 2002-2019. Introcution to Test Driven Design (TDD). Viitattu. 8.6.2018 http://agiledata.org/essays/tdd.html.

Barber, Derek. 2012. Why Test-driven Development? Viitattu. 10.6.2018. http://derekbarber.ca/blog/2012/03/27/why-test-driven-development/.

Beck, Kent. 2002. Test-driven development: by example. Boston: Addison-Wesley.

Bergmann, Sebastian. 2005–2011. PHPUnit Manual. Viitattu. 11.2.2019. https://phpunit.readthedocs.io/en/8.0/.

Causevic, Adnan. Punnekkat, Sasikumar. Sundmark, Daniel. 2013. TDDhq: Achieving Higher Quality Testing in Test Driven Development.

Collin, Niklas. 2010. TDD ja ATDD. Ohjelmistojen testaus -opintojakson luentomateriaali. Tampereen teknillinen yliopisto. Viitattu 31.8.2017. http://www.cs.tut.fi/~testaus/s2010/luennot/Collin.pdf.

Dooley, John 2011. Software Development and Professional Practice. E-Kirja. apress https://www.apress.com/gp/book/9781430238027#otherversion=9781430238010.

Fucci, Davide. Erdogmus, Hakan. Turhak, Burhan. Oivo, Markku. Juristo, Natalia. 2017. A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?

Gelpern, D. Hetzel, B. 1988. The growth of software testing.

Glavas, Andrija. 2018. Unit testing in Magento 2. Viitattu. 12.2.2019. https://inchoo.net/magento-2/unit-testing-magento-2.

Hautamäki, Johanna. 2015. Kehittämistutkimusta ja ongelmanratkaisua YAMK-opinnäytetöissä. Viitattu 5.3.2019. https://centriaamk.wordpress.com/2015/12/18/kehittamistutkimusta-ja-ongelmanratkaisua-yamk-opinnaytetoissa/.

Janzen, D. S. & Saiedan, H. 2008. Does Test-Driven Development Really Improve Software Design Quality? IEEE Software, 25, 2, 77-84.

Kananen, J. 2015. Kehittämistutkimus opinnäytetyönä. Kehittämistutkimuksen kirjoittamisen käytännön opas. Jyväskylän ammattikorkeakoulu.

Kopp, Vinai. 2016. The Skeleton Module Kata. Viitattu. 12.2.2019. http://vinaikopp.com/2016/02/05/01_the_skeleton_module_kata/ .

Luo, Lu. 2013. Software testing techniques, technology maturation and research strategy.

Madeyski, Lech. 2010. The impact of Test-First programming on branch coverage and mutation score indicator of unit tests: An experiment.

Magento. 2018. Magento DevDocs. Viitattu. 13.9.2018. https://devdocs.magento.com/.

Mumah, Von Larry. 2018. Myths Surrounding Unit Tests. Viitattu. 11.2.2019. https://www.swms.de/en/blog/myths-surrounding-unit-tests/.

Nyári, István. 2017. Magento Commerce: History and features of the most popular ecommerce platform. Viitattu. 8.6.2018. https://aionhill.com/magento-commerce .

Porter, Davis. 2018. What Adobe's Acquisition of Magento Means For the Future of Ecommerce. Viitattu. 14.9.2018. https://ecommerce-platforms.com/ecommerce-news/adobe-buys-magento .

Rauhala, Eija. 2010. Ohjelmistotestauksen suunnittelu - Case: A-lehdet Oy:n laskujen tulostusohjelma. Opinnäytetyö. Haaga-Helia, tietojenkäsittelyn koulutusohjelma. Viitattu. 14.2.2019. https://www.theseus.fi/bitstream/handle/10024/23687/Rauhala_Eija.pdf?sequence=1&isAllowed=y.

Storm, Alan. 2015. Magento 2 Object Manager Preferences. Viitattu. 8.5.2018. https://alanstorm.com/magento_2_object_manager_preferences/.

Shrivastava, Divya P. Jain, R. C. 2011. Unit test case design metrics in test driven development.

Trucchia, Francesco & Romei, Jacopo. 2010. Pro PHP Refactoring. Springer Science+Business Media, LLC.

Who Leads the Market of Ecommerce Platforms in 2017?. 2017. Blogiteksti verkkokauppa-alustojen markkinaosuudesta syksyllä 2017. Viitattu 12.5.2018. https://blog.aheadworks.com/ecommerce-market-2017/.

Yablonskaya, Tanya. 2018. Exploring all strengths and weaknesses of Magento CMS. Viitattu 10.9.2018. https://www.scnsoft.com/blog/magento-cms.

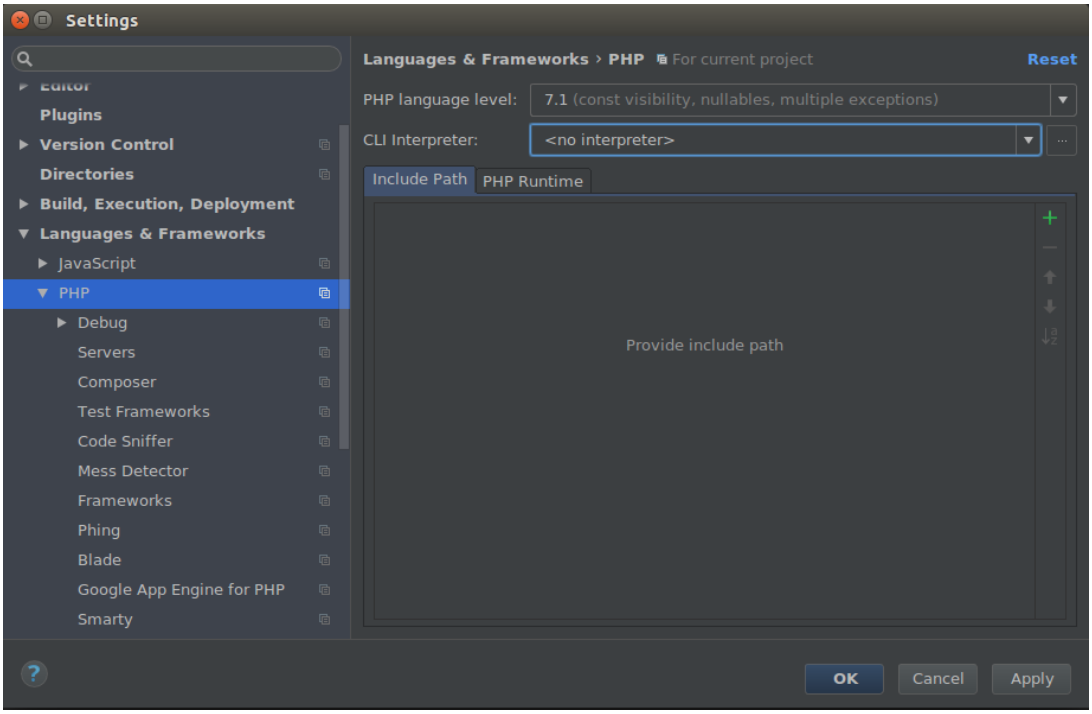
Zandstra, Matt. 2008. PHP objects, patterns and practice, second edition. Apress.

Liitteet

Liite 1. Installing and configuring the testing tools in PHPStrom

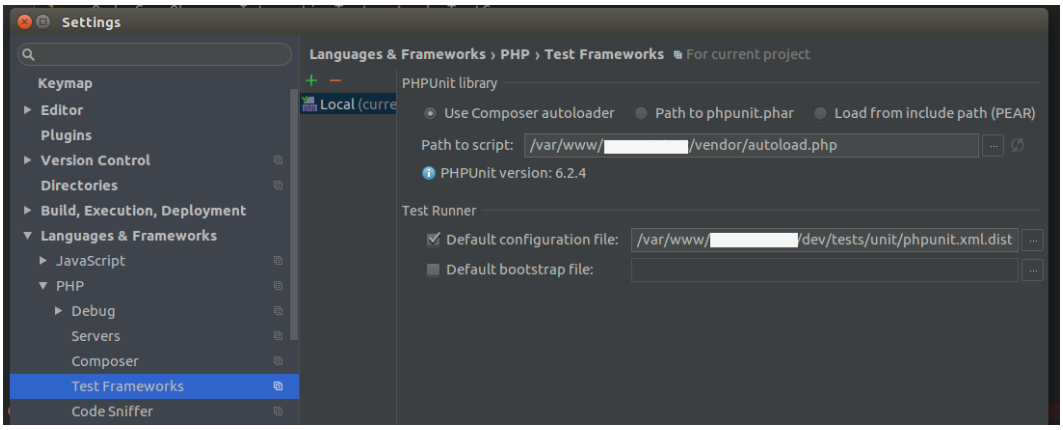
This appendix briefly explains how PHPUnit and Xdebug tools are configured for Magento 2 unit and integration testing purposes in PhpStorm IDE. The operating system used in these instructions is Ubuntu 16.04 LTS. The instructions presume that PHPStrom is already installed and configured.

1) Configure the PHP interpreter from Files->Settings->Languages & Frameworks->PHP. The interpreter is selected from the CLI interpreter dropdown menu. If the menu is empty, a new interpreter can be created from the settings button next to the dropdown.



2) The PHPUnit configuration is created in Settings->Languages & Frameworks->PHP->Test Frameworks. A new configuration can be made by clicking the + sign and choosing a local PHPUnit cofiguration. Use composer autoloader to

load the PHPUnit library, and dev/tests/unit/phpunit.xml.dist under your project root folder as the “Default configuration” file.



3) Next, create the empty integration test database and a dedicated user for it. Then create the file dev/tests/integration/etc/install-config-mysql.php and enter the following or similiar data:

```
<?php

return [
    'db-host' => 'localhost',
    'db-user' => 'root',
    'db-password' => 'root',
    'db-name' => 'magento_integration_tests',
    'db-prefix' => '',
    'backend-frontname' => 'backend',
    'admin-user' => \Magento\TestFramework\Bootstrap::ADMIN_NAME,
    'admin-password' => \Magento\TestFramework\Bootstrap::ADMIN_PASSWORD,
    'admin-email' => \Magento\TestFramework\Bootstrap::ADMIN_EMAIL,
    'admin-firstname' => \Magento\TestFramework\Bootstrap::ADMIN_FIRSTNAME,
    'admin-lastname' => \Magento\TestFramework\Bootstrap::ADMIN_LASTNAME,
];
```

4) Next, go to dev/tests/integration directory and create phpunit.xml file. Copy the contents from phpunit.xml.dist to phpunit.xml. Now you can edit the configurations for integration tests inside phpunit.xml. If you wish to only test custom module integrations, you can edit the testsuites part in the following way:

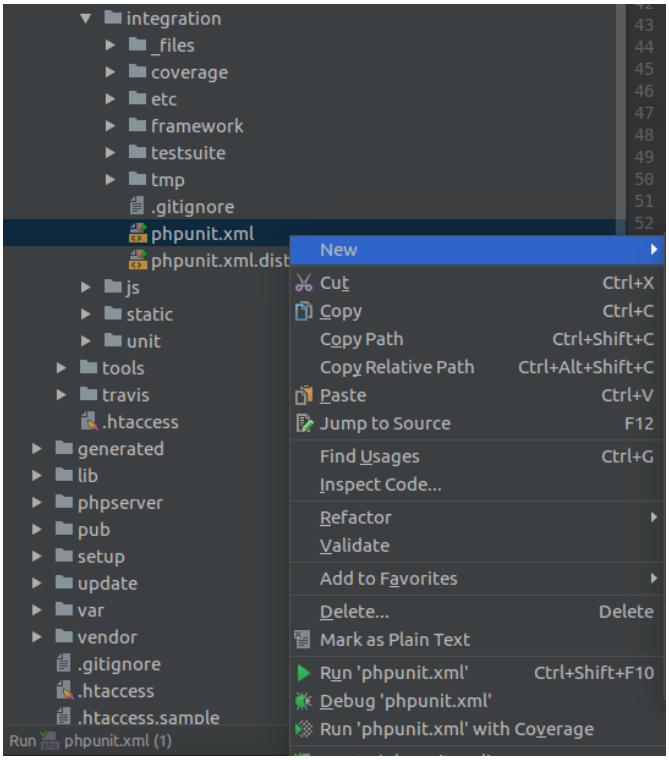
```
<!-- Test suites definition -->
  <testsuites>
    <testsuite name="Module Integration Tests">
      <directory>../../../app/code/*/*/Test/Integration</directory>
    </testsuite>
  </testsuites>
```

If you wish to start each integration with a cleaned integration test database, you can edit the TEST_CLEANUP constant by setting the value to “enabled”. However, this slows the integration tests down a bit.

```
<const name="TESTS_CLEANUP" value="enabled"/>
```

5) Now go to dev/tests/unit directory and again create phpunit.xml file. Copy the contents from phpunit.xml.dist into phpunit.xml. The testsuite can be altered in a similar manner as the integration tests.

```
<testsuite name="Module Unit Tests">
    <directory suffix="Test.php">../../../app/code/*/*/Test/Unit</directory>
</testsuite>
```



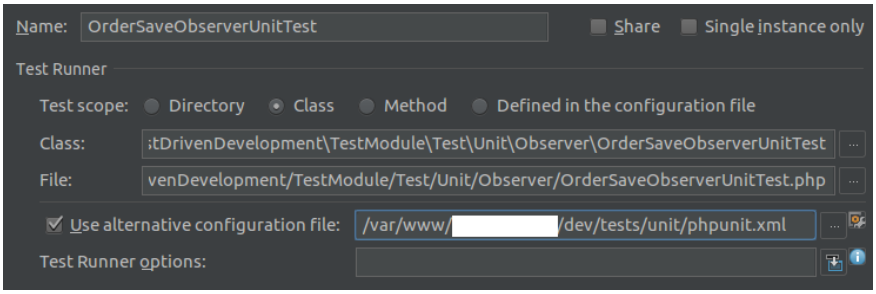
6) The integration and unit tests can be run by right clicking either of the phpunit.xml files and selecting Run 'phpunit.xml'. Alternatively, you can run the tests from terminal by navigating to either the integration or unit test directory and entering the following:

```
$ ../../../vendor/bin/phpunit
```

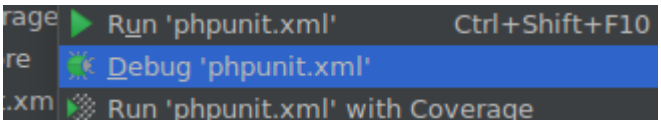
Or you run a specified testsuite by entering the following:

```
$ php ../../../vendor/bin/phpunit --testsuite "Third Party Integration Tests"
```

7) Configurations for the phpunit.xml test run and single testcase runs can be changed in Run->Edit Configurations. When running single testcases, the phpunit.xml file must be added as an alternative configuration file or the run may not work properly.



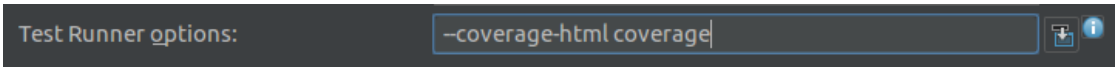
8) Xdebug can be utilized in test runs by adding a breakpoint to your preferred line, and starting a debug test run.



Xdebug can be used ie. to track the code functionality when performing integration tests that cover multiple methods. The Xdebug component installation is also required for the Code Coverage functionality.

9) The regular Code Coverage runs are executed from the **Run 'test_name' with Coverage** button. It displays test coverage data in the PHPUnit user interface with cover percentages for each class. If you wish to use the Code Coverage HTML report that covers every single line of code in the testsuite.

   You can add the option to the Test Runner options in Run→Edit Configurations by typing –coverage-html coverage:



10) The coverage logging must also be enabled by uncommenting and editing the following line in dev/tests/unit/phpunit.xml:

    <log type="coverage-html" target="coverage/test-reports/coverage" charset="UTF-8" yui="true" highlight="true"/>

    When you run unit tests, the coverage reports will be saved to the directory defined in the file. The report can be opened in your browser, and it covers any modules, classes and methods that were covered in the latest test run.

    Class view example:

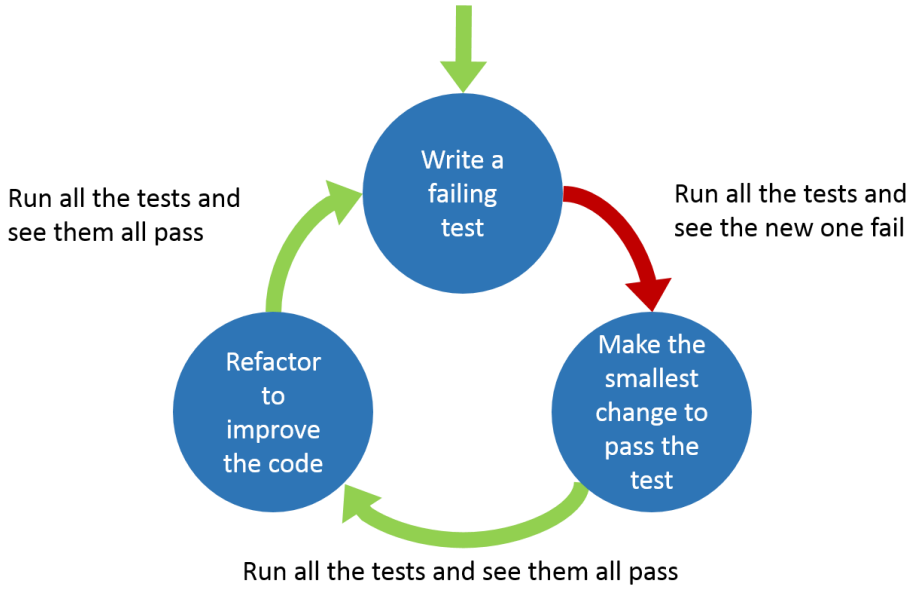
	Code Coverage								
	Classes and Traits			Functions and Methods			Lines		
Total		100.00%	1 / 1		100.00%	2 / 2	CRAP	100.00%	8 / 8
OrderSaveAfterObserver		100.00%	1 / 1		100.00%	2 / 2	4	100.00%	8 / 8
__construct					100.00%	1 / 1	1	100.00%	3 / 3
execute					100.00%	1 / 1	3	100.00%	5 / 5

Line by line code coverage:

```
33      public function getCreatedOrdersValue($customerId){
34
35          try {
36              /** @var CustomerInterface $customer */
37              $customer = $this->customer->getById($customerId);
38          } catch (NoSuchEntityException $e) {
39              throw $e;
40          }
41          $orderNum = $customer->getCreatedOrders();
42
43          if ($orderNum > 0){
44              $customer->setCreatedOrders($orderNum +1);
45          }
46          else {
47              $customer->setCreatedOrders(1);
48          }
49          $this->customer->save($customer);
50      }
```

Liite 2. Test driven module development steps

This appendix covers the steps for starting the test driven development process in Magento 2 module development. These instructions assume that the steps from appendix 1 have been followed to install the required testing tools in PHPStorm. The successful implementation of test driven module development techniques also require fundamental understanding of the PHPUnit framework.



1. Take time to plan which Magento 2 classes and interfaces are needed to run your module's supposed function, and how you will implement the usage of these components. Make sure that the component you are planning to use works the way you think.

    - Study the interface, repository of factory you are planning to use and make sure it can be implemented in the way you intend to.

2. Start by creating integration tests that test the functionality of your components in Magento's system. This includes module registration and perhaps plugin, observer or route configurations. After the tests fail, create the needed configuration files to make the tests pass.

3. Create the integration tests that test any interaction between your written code and Magento. The tests should of course fail at first, because no functionality has been created to make them pass.

    - Utilize fixtures to create predefined database content to simulate products, customers, orders, etc. These can be loaded in your integration test to create a preferred testing circumstance. Fixtures are saved in the module's Test/Integration/_files folder.
    - Use assertions to test data processing in your written functionality. Mock objects are possible but not necessary to use in integration tests.
    - When writing integration tests, keep i mind that it's more about testing user story behavior (i.e. customer registration, adding products to cart) than individual component functionality.

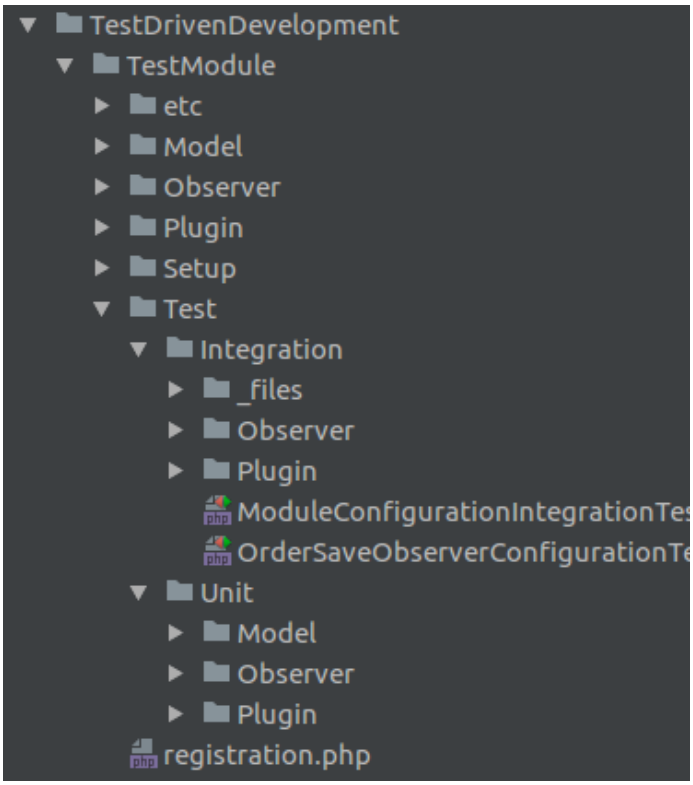
4. Begin creating the unit tests for individual module components. A single testcase should be responsible for testing a single block of code, or a method.

    - Create tests for any individual methods that need testing. Plan the expectations of the written code beforehand. What is the desired output of the functionality, and what are the results if something goes wrong?
    - Keep a fast development cycle. Write a failing test, then the code to pass the test, and lastly any possible refactoring before starting over.
    - Try to create expectations for different data types to avoid any surprises.
    - Utilize Mocks and Stubs to act as test doubles for actual objects you may need in your code.

5. When you have written the code to make all the desired tests to pass, the integration tests should also pass. You can refactor or add more integration tests at this point, otherwise you are finished.

6. All modules do not require similar type of testing. Simpler implementations may work with unit testing only. In any event, make sure to plan the tests carefully.

An example of the module directory including test folders.



Liite 3. ModuleConfigurationIntegrationTest.php tiedosto

```
<?php

namespace TestDrivenDevelopment\TestModule\Test\Integration;

use Magento\Framework\App\DeploymentConfig;
use Magento\Framework\Component\ComponentRegistrar;
use Magento\Framework\Module\ModuleList;
use Magento\TestFramework\ObjectManager;
use PHPUnit\Framework\TestCase;

class ModuleConfigurationIntegrationTest extends TestCase
{

   private $module = 'TestDrivenDevelopment_TestModule';

   /**
    * @var ObjectManager
    */
   private $objectManager;

   protected function setUp()
   {
      $this->objectManager = ObjectManager::getInstance();
   }

   public function testModuleRegistration()
   {
      $registrar = new ComponentRegistrar();
      $paths = $registrar->getPaths(ComponentRegistrar::MODULE);
      $this->assertArrayHasKey($this->module, $paths);
   }

   public function testModuleIsEnabled()
   {
      $deploymentConfig = $this->objectManager->create(DeploymentConfig::class);

      /** @var ModuleList $moduleList */
      $moduleList = $this->objectManager->create(ModuleList::class,
         ['config' => $deploymentConfig]);
      $this->assertTrue($moduleList->has($this->module));
   }


}
```

Liite 4. OrderSaveObserverConfigurationTest.php tiedosto

```
<?php

namespace TestDrivenDevelopment\TestModule\Test\Integration;

use Magento\Framework\Event\ConfigInterface;
use Magento\TestFramework\ObjectManager;
use PHPUnit\Framework\TestCase;

class OrderSaveObserverConfigurationTest extends TestCase
{
   private $observerName = 'testmodule_order_save_after_observer';

   public function testOrderSaveAfterObserverConfigurations()
   {
      /** @var ConfigInterface $observerConfig */
      $observerConfig = ObjectManager::getInstance()->create(ConfigInterface::class);
      $observers = $observerConfig->getObservers('sales_order_save_after');

      $this->assertArrayHasKey($this->observerName, $observers);
   }

}
```

Liite 5. OrderSaveObserverUnitTest.php tiedosto

```
<?php

namespace TestDrivenDevelopment\TestModule\Test\Unit\Observer;

use Magento\Customer\Api\Data\CustomerInterface;
use Magento\Framework\Event;
use Magento\Framework\Event\ObserverInterface;
use Magento\Framework\Event\Observer;
use Magento\Sales\Api\Data\OrderInterface;
use PHPUnit\Framework\TestCase;
use TestDrivenDevelopment\TestModule\Observer\OrderSaveAfterObserver;
use TestDrivenDevelopment\TestModule\Model\CustomerOrderNumCalculator;

class OrderSaveObserverUnitTest extends TestCase
{

   /** @var  Event |  \PHPUnit_Framework_MockObject_MockObject*/
   private $eventMock;

   /** @var  ObserverInterface | \PHPUnit_Framework_MockObject_MockObject */
   private $observerMock;

   /** @var  CustomerOrderNumCalculator | \PHPUnit_Framework_MockObject_MockOb-
ject */
   private $calculator;

   /** @var  OrderInterface | \PHPUnit_Framework_MockObject_MockObject*/
   private $order;

   /** @var  CustomerInterface | \PHPUnit_Framework_MockObject_MockObject*/
   private $customer;

   /** @var OrderSaveAfterObserver */
   private $orderSaveAfterObserver;


   public function setUp(){

      $methodsToStub = array_merge(
         get_class_methods(OrderInterface::class),
         ['getOrderNumIsSet', 'setOrderNumIsSet']
      );

      $this->order = $this->getMockBuilder(OrderInterface::class)
      ->setMethods($methodsToStub)
         ->disableOriginalConstructor()->getMock();
      $this->customer = $this->createMock(CustomerInterface::class);
      $this->calculator = $this->createMock(CustomerOrderNumCalculator::class);
      $this->observerMock = $this->createMock(Observer::class);
```

```
    $this->eventMock = $this->createPartialMock(Event::class, ['getData']);

    $this->orderSaveAfterObserver = new OrderSaveAfterObserver($this->calculator,
    $this->order);
}

public function testImplementsObserverInterface(){

    $this->assertInstanceOf(ObserverInterface::class, $this->orderSaveAfterObserver);
}

public function testCreatedOrdersMethodCallForExistingCustomer() {

    $this->observerMock->method('getEvent')
        ->willReturn($this->eventMock);
    $this->eventMock->method('getData')
        ->with('order')
        ->willReturn($this->order);
    $this->order->method('getCustomerIsGuest')
        ->willReturn(false);
    $this->order->method('getOrderNumIsSet')
        ->willReturn(false);
    $this->calculator->expects($this->once())
        ->method('getCreatedOrdersValue');
    $this->order->expects($this->once())
        ->method('setOrderNumIsSet');

    $this->orderSaveAfterObserver->execute($this->observerMock);

}

public function testCreatedOrdersMethodCallForGuest() {

    $this->observerMock->method('getEvent')
        ->willReturn($this->eventMock);
    $this->eventMock->method('getData')
        ->with('order')
        ->willReturn($this->order);
    $this->order->method('getCustomerIsGuest')
        ->willReturn(true);
    $this->calculator->expects($this->never())
        ->method('getCreatedOrdersValue');

    $this->orderSaveAfterObserver->execute($this->observerMock);

}

public function testCreatedOrdersNumAttributeIfAlreadySet(){

    $this->observerMock->method('getEvent')
        ->willReturn($this->eventMock);
    $this->eventMock->method('getData')
        ->with('order')
```

```
            ->willReturn($this->order);
        $this->order->method('getCustomerIsGuest')
            ->willReturn(false);
        $this->order->method('getOrderNumIsSet')
            ->willReturn(true);
        $this->calculator->expects($this->never())
            ->method('getCreatedOrdersValue');

        $this->orderSaveAfterObserver->execute($this->observerMock);
    }

}
```

Liite 6. OrderNumCalculatorUnitTest.php tiedosto

```
<?php

namespace TestDrivenDevelopment\TestModule\Test\Unit\Observer;

use Magento\Customer\Api\CustomerRepositoryInterface;
use Magento\Customer\Api\Data\CustomerInterface;
use Magento\Framework\Exception\NoSuchEntityException;
use PHPUnit\Framework\TestCase;
use TestDrivenDevelopment\TestModule\Model\CustomerOrderNumCalculator;

class OrderNumCalculatorUnitTest extends TestCase
{
   /** @var  CustomerInterface | \PHPUnit_Framework_MockObject_MockObject */
   private $customerInterfaceMock;

   /** @var  CustomerRepositoryInterface | \PHPUnit_Framework_MockObject_MockOb-
ject*/
   private $customer;

   /** @var CustomerOrderNumCalculator */
   private $model;

   /** @var NoSuchEntityException | \PHPUnit_Framework_MockObject_Stub_Exception */
   private $exceptionStub;


   public function setUp(){

      $methodsToStub = array_merge(
         get_class_methods(CustomerInterface::class),
         ['getCreatedOrders', 'setCreatedOrders']
      );

      $this->customerInterfaceMock = $this->getMockBuilder(CustomerInterface::class)
         ->setMethods($methodsToStub)->disableOriginalConstructor()->getMock();

      $this->customer = $this->createMock(CustomerRepositoryInterface::class);

      $this->exceptionStub = $this->createMock(NoSuchEntityException::class);

      $this->model = new CustomerOrderNumCalculator($this->customer);
   }

   public function testMethodSavesCustomAttributeData(){

      $customerId = 1;

      $this->customer->method('getById')
         ->willReturn($this->customerInterfaceMock);
```

```
        $this->customerInterfaceMock->expects($this->once())
            ->method('getCreatedOrders');
        $this->customerInterfaceMock->expects($this->once())
            ->method('setCreatedOrders');
        $this->customer->expects($this->once())
            ->method('save');

        $this->model->getCreatedOrdersValue($customerId);

    }

    public function testAddIncrementalValueToCreatedOrders(){

        $customerId = 1;
        $createdOrders = 1;

        $this->customer->method('getById')
            ->willReturn($this->customerInterfaceMock);
        $this->customerInterfaceMock->method('getCreatedOrders')
            ->willReturn($createdOrders);
        $this->customerInterfaceMock->expects($this->once())
            ->method('setCreatedOrders')->with($createdOrders +1);

        $this->model->getCreatedOrdersValue($customerId);
    }

    public function testCustomerEntityNotFoundException(){

        $this->customer->method('getById')->willThrowException($this->exceptionStub);
        $this->expectException(NoSuchEntityException::class);

        $this->model->getCreatedOrdersValue(null);
    }

}
```

Liite 7. OrderSaveObserver.php tiedosto

```
<?php

namespace TestDrivenDevelopment\TestModule\Observer;

use Magento\Framework\Event\Observer as Event;
use Magento\Framework\Event\ObserverInterface;
use Magento\Sales\Api\Data\OrderInterface;
use TestDrivenDevelopment\TestModule\Model\CustomerOrderNumCalculator;

class OrderSaveAfterObserver implements ObserverInterface
{
   /**
    * @var CustomerOrderNumCalculator
    */
   private $customerOrderNumCalculator;
   /**
    * @var OrderInterface
    */
   private $orderInterface;

   /**
    * OrderSaveAfterObserver constructor.
    * @param CustomerOrderNumCalculator $customerOrderNumCalculator
    * @param OrderInterface $orderInterface
    */
   public function __construct
   (
      CustomerOrderNumCalculator $customerOrderNumCalculator,
      OrderInterface $orderInterface
   )
   {
      $this->customerOrderNumCalculator = $customerOrderNumCalculator;
      $this->orderInterface = $orderInterface;
   }

   /**
    * sales_order_save_after
    *
    * @param Event $observer
    */
   public function execute(Event $observer)
   {
      /** @var OrderInterface $order */
      $order = $observer->getEvent()->getData('order');

      if (!$order->getCustomerIsGuest() && !$order->getOrderNumIsSet()){

         $this->customerOrderNumCalculator->getCreatedOrdersValue(
         $order->getCustomerId(), $order);
```

```
            $order->setOrderNumIsSet(true);
        }
    }
}
```

Liite 8. CustomerOrderNumCalculator.php tiedosto

```
<?php


namespace TestDrivenDevelopment\TestModule\Model;

use Magento\Customer\Api\CustomerRepositoryInterface;
use Magento\Customer\Api\Data\CustomerInterface;
use Magento\Framework\Exception\NoSuchEntityException;

class CustomerOrderNumCalculator
{
   /**
    * @var CustomerRepositoryInterface
    */
   private $customerRepository;
   /**
    * CustomerOrderNumCalculator constructor.
    * @param CustomerRepositoryInterface $customerRepository
    */
   public function __construct
   (
      CustomerRepositoryInterface $customerRepository
   )
   {
      $this->customerRepository = $customerRepository;
   }

   /**
    * @param  int $customerId
    * @throws NoSuchEntityException
    */
   public function getCreatedOrdersValue($customerId){

      try {
         /** @var CustomerInterface $customer */
         $customer = $this->customerRepository->getById($customerId);
      } catch (NoSuchEntityException $e) {
         throw $e;
      }
      $orderNum = $customer->getCreatedOrders();

      if ($orderNum > 0){
         $customer->setCreatedOrders($orderNum +1);
      }
      else {
         $customer->setCreatedOrders(1);
      }
      $this->customerRepository->save($customer);
   }
}
```

Liite 9. registration.php ja module.xml tiedostot

```
<?php
\Magento\Framework\Component\ComponentRegistrar::register(
   \Magento\Framework\Component\ComponentRegistrar::MODULE,
   'TestDrivenDevelopment_TestModule',
   __DIR__
);
```

```
<?xml version="1.0" ?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSche-
maLocation="urn:magento:framework:Module/etc/module.xsd">
   <module name="TestDrivenDevelopment_TestModule" setup_version="1.0.4">
   </module>
</config>
```

Liite 10. events.xml ja di.xml tiedostot

```
<?xml version="1.0"?>

<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSche-
maLocation="urn:magento:framework:Event/etc/events.xsd">
   <event name="sales_order_save_after">
      <observer name="testmodule_order_save_after_observer"
      instance="TestDrivenDevelopment\TestModule\Observer\OrderSaveAfterObserver" />
   </event>
</config>
```

```
<?xml version="1.0"?>

<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSche-
maLocation="urn:magento:framework:ObjectManager/etc/config.xsd">
   <type name="Magento\Customer\Api\CustomerRepositoryInterface">
      <plugin name="testmodule_customer_interface_plugin"
      type="TestDrivenDevelopment\TestModule\Plugin\CustomerRepositoryInterfacePlugin"
      disabled="true"/>
   </type>
</config>
```

Liite 11. customer.php fikstuuritiedosto ja tiedoston käytön esimerkki

```
<?php

namespace TestDrivenDevelopment\TestModule\Test\Integration\_files;

use Magento\Customer\Model\CustomerRegistry;

$objectManager = \Magento\TestFramework\Helper\Bootstrap::getObjectManager();

/** @var $repository \Magento\Customer\Api\CustomerRepositoryInterface */
$repository = $objectManager->create(\Magento\Customer\Api\CustomerRepositoryInter-
face::class);

/** @var \Magento\Customer\Model\Customer $customer */
$customer = $objectManager->create(\Magento\Customer\Model\Customer::class);

/** @var CustomerRegistry $customerRegistry */
$customerRegistry = $objectManager->get(CustomerRegistry::class);

$customer->setWebsiteId(1)
   ->setId(99)
   ->setEmail('testi.asiakas@example.com')
   ->setPassword('password')
   ->setGroupId(1)
   ->setStoreId(1)
   ->setIsActive(1)
   ->setFirstname('Testi')
   ->setCreatedOrders(0)
   ->setLastname('Asiakas')
   ->setDefaultBilling(1)
   ->setDefaultShipping(1)
   ->setTaxvat('12')
   ->setGender(0);

$customer->isObjectNew(true);
$customer->save();
$customerRegistry->remove($customer->getId());
```

```
public static function loadFixture(){

      include __DIR__ . '/../_files/customer.php';
 }
```