KÄYTTÄYTYMISLÄHTÖINEN KEHITYS (BDD) BEHAT-KEHYKSELLÄ

HAMK
HÄMEEN AMMATTIKORKEAKOULU

Ammattikorkeakoulun opinnäytetyö

Tietojenkäsittelyn koulutusohjelma

Visamäki, syksy 2015

Teemu Nurmi

HAMK
HÄMEEN AMMATTIKORKEAKOULU

HÄMEENLINNA, VISAMÄKI
Tietojenkäsittelyn koulutusohjelma
Systeemityö

Tekijä Teemu Nurmi **Vuosi** 2015

Työn nimi Käyttäytymislähtöinen kehitys (BDD) Behat-kehyksellä.

TIIVISTELMÄ

Opinnäytetyössä oli tavoitteena tutkia ja vertailla käyttäytymislähtöistä kehittämistä ja testivetoista kehittämistä. Aihetta suositteli työharjoittelupaikka Valo Interactive Oy, koska minun olisi hyvä ymmärtää käyttäytymislähtöistä kehittämistä työelämää silmällä pitäen.

Aihetta tutkittiin enimmäkseen teorian kautta, kirjallisiin ja elektronisiin lähteisiin perustuen. Lopuksi tehtiin pieni käytännön testaus käyttäen PHP-kielelle kehitettyä Behat-kehystä, joka mahdollistaa käyttäytymislähtöisen kehittämisen.

Työssä havaittiin, että käyttäytymislähtöinen kehittäminen ei ole vain päivitetty versio testivetoisesta kehittämisestä. Se on kommunikointiväline eri sidosryhmien väliseen kommunikointiin teknisestä osaamistasosta riippumatta. Se on siis erilaisempi ja isompi asia kuin vain evoluutio testivetoisesta kehittämisestä.

Avainsanat BDD, Käyttäytymislähtöinen kehitys, TDD, Testivetoinen kehitys, Behat

Sivut 27 s.

HÄMEENLINNA, VISAMÄKI
Degree Programme in Business Information Technology
Application development

Author Teemu Nurmi **Year** 2015

Subject of Bachelor's thesis Behavior-driven development (BDD) using Behat

ABSTRACT

The goal of this thesis was to study and compare behavior-driven development and test-driven development. Subject of thesis was recommended by Valo Interactive Oy because it would be beneficial for me to understand behavior-driven development for future career in mind.

The topic was mostly examined through theory using written and electronic sources. The practical part of thesis was done using Behat framework which makes possible behavior-driven development with PHP programming language.

As a result of thesis it was discovered that behavior-driven development is not just an updated version of test-driven development. It is a method of communication between various stakeholders regardless of their level of technical expertise. It is a different and bigger thing than just evolution of test-driven development.

Keywords BDD, Behavior-driven development, TDD, Test-driven development, Behat

Pages 27 p.

KÄSITELUETTELO

Test-driven development (TDD)	Testivetoinen kehitys.
Behavior-driven development (BDD)	Käyttäytymislähtöinen kehitys.
Unit test	Yksikkötestaus, joka testaa yksittäistä moduulia tai luokkaa.
Integration test	Integraatiotesti, joka testaa useamman moduulin yhteistoimintaa.
System test	Järjestelmätestaus, joka testaa koko järjestelmää kerralla.
Stress test	Kuormitustesti, joka testaa koko järjestelmän kuormituksen kestävyyttä.
Refaktorointi	Koodin uudelleen kirjoittamista, jotta siitä tulee teknisesti kehittyneempää. Sisäistä rakennetta muutetaan ilman ulospäin näkyviä vaikutuksia.
Step definition	Suomeksi testiaskel. Skenaario koostuu listasta, jossa kerrotaan mitä pitäisi tapahtua. Jokaista selkokielistä tekstiriviä kutsutaan testiaskeleeksi.
Core stakeholders	Suomeksi ydinosapuoli. BDD-mallissa ydinosapuoli on ryhmä, jonka ongelmaa yritetään ratkaista.
Incidental stakeholders	Suomeksi oheisosapuoli. BDD-mallissa oheisosapuoli on ryhmä, johon kuuluvat kaikki ongelmaa ratkaisevat tahot.
Domain-driven design (DDD)	Liiketoimintavetoinen kehitys.
Story	Suomeksi tarina. Selkokielinen testitapaus.
Template	Suomeksi sapluuna. Selkokielisen testitapauksen rakennemalli.
Title	Suomeksi otsikko. Tarinan otsikko, jossa kerrotaan, mistä tarina on kyse, koska tarinoita voi olla useita.
Narrative	Suomeksi kerronta. Tarinan komponentti, jossa kerrotaan, kuka tekee ja mitä.
Acceptance criteria	Suomeksi hyväksymiskriteeri. Oikea tulos, joka päättää tarinan.

SISÄLLYS

1 JOHDANTO

Tämä työ käsittelee käyttäytymislähtöistä kehittämistä (behavior-driven development), joka on yksi tuoreimmista ohjelmistokehityksen malleista. Työn aiheen keksimisestä vastasi työharjoittelupaikkani Valo Interactive Oy, joka on australialais-suomalainen ohjelmistoalan yritys. Osoitin kiinnostusta automaatiotestaukseen, johon en ehtinyt perehtyä harjoittelun aikana. Harjoittelupaikalta annettiin ymmärtää, että käyttäytymislähtöinen testaus olisi hyvä hallita työelämää silmällä pitäen.

Aihe on kiinnostava, koska minulla ei ollut aikaisempaa tietoa käyttäytymislähtöisestä testauksesta tai automaatiotestauksesta. Tämän vuoksi työssä selvennetään mitä käyttäytymislähtöinen kehittäminen on, sekä miten se eroaa testivetoisesta kehittämisestä (test-driven development). Lisäksi selvitetään, onko käyttäytymislähtöinen kehitys vain testivetoisen kehittämisen päivitetty versio. Työssä tutkitaan myös, miksi näitä menetelmiä käytetään, ja sopivatko ne jokaiseen ohjelmistoprojektiin. Aihetta selvennetään teorian ja testauksen kautta, eikä niinkään ohjelmoinnin kautta. Tämän vuoksi käytännön osuudessa keskitytään testaamiseen, eikä niinkään sovelluksen kehittämiseen.

Työn alku koostuu testauksen ja automaatiotestauksen teorian selventämisestä. Tässä osuudessa käydään mm. läpi syitä miksi käsin tehtävästä testauksesta jouduttiin osittain siirtymään automaatiotestaukseen. Seuraavassa luvussa käsitellään testivetoista kehittämistä sekä sen hyötyjä että haasteita. Näiden kolmen luvun jälkeen siirrytään varsinaiseen pääaiheeseen, joka on käyttäytymislähtöinen kehittäminen. Osuudessa selvitetään käyttäytymislähtöisen kehittämisen ja testivetoisen kehittämisen eroavaisuuksia sekä sitä, mihin testivetoisen kehittämisen ongelmiin käyttäytymislähtöinen kehittäminen pyrkii vastaamaan. Lisäksi selvennetään, onko käyttäytymislähtöinen kehittämisen malli vain päivitetty versio testivetoisen kehittämisen mallista, sekä mitä hyötyä ja haasteita se tuo. Viimeinen luku käsittelee Behat-kehystä. Ensin käydään teorian tasolla läpi sitä, mikä Behat on. Tämän jälkeen käytännön osuudessa kokeillaan Behatin pikaoppaan avustuksella, miten käyttäytymislähtöisen kehittämisen mallin mukaista testausta tehdään käytännössä.

2 TESTAUS

Ohjelmistotestauksessa on tarkoitus varmistaa, että toteuttavasta ohjelmistotuotteesta tulee toivotun kaltainen. Näin varmistetaan että tehdään oikeaa tuotetta, joka on myös tehty oikein. Lisäksi testauksessa pyritään tunnistamaan poikkeamat suunnitelmasta. (Kasurinen 2013, 10.)

Ohjelmistotestaus on yksi osa-alue ohjelmistotuotannosta, mutta se on paljon laajempi kokonaisuus kuin ohjelmointityö. Testauksen lisäksi testaaja voi joutua nimittäin tekemään muitakin työtehtäviä, kuten ohjelmoimaan, dokumentoimaan tai haastattelemaan koekäyttäjiä. Näiden syiden vuoksi testaajan ammatti saattaa vaihdella suuresti eri ohjelmistotaloissa.

Kasurinen (2013, 10) havainnollistaa esimerkein testaustyön eroja Ohjelmistotestauksen käsikirjassa. Esimerkkinä hän mainitsee pelialan yrityksen, jossa testataan graafisia elementtejä ja pelin hauskuutta. Toisena esimerkkinä hän mainitsee valtion virastolle veroilmoitusta tekevää yritystä. Testaamisessa he eivät keskity selvittämään veroilmoituksen täyttämisen hauskuutta, vaan sen helppokäyttöisyyttä. Viimeisenä esimerkkinä Kasurinen mainitsee kriittisen ajotietokonejärjestelmän testaamista autoteollisuuden näkökulmasta. Silloin ei ole edellä mainituilla testauksen osa-alueilla väliä, vaan on tärkeätä minimoida mahdolliset viiveet järjestelmässä. Äkkijarrutustilanteessa ei ole aikaa odotella, että kriittisen ABS-järjestelmän ajuri latautuu.

2.1 Testauksen synty

Vielä ennen 60-lukua tietokoneohjelmat olivat hyvin yksinkertaisia ja niitä ei juuri tarvinnut testata. Kuusikymmentäluvulle tultaessa tietokoneiden ohjelmat alkoivat kuitenkin monimutkaistua. Vuosikymmenen loppupuolella saavutettiin tilanne, jossa ohjelmia piti alkaa testaamaan.

Vuonna 1968 Saksassa pidetyssä konferenssissa tuli tutuksi käsite "ohjelmistokriisi" (Kasurinen 2013, 10). Sillä tarkoitettiin yllättävää ongelmaa, joka oli seurausta tietokoneiden suorituskyvyn räjähdysmäisestä kasvusta. Tämän myötä oli alettu kehittää isoja ja monimutkaisia ohjelmia, joiden kaikkia yksityiskohtia yksittäinen ohjelmoija ei kyennyt enää hahmottamaan. Lisäksi moni ohjelmisto ei ottanut kaikkea irti käytettävästä laitteesta. Tämä aiheutui ohjelmistobudjettien paisumista yli suunnitelmien, sekä ohjelmistojen huonoa laatua (Kasurinen 2013, 11).

Ohjelmistokriisin tärkein saavutus oli käsitteiden synty, kuten ohjelmistotuotanto, ohjelmistoprosessit sekä ohjelmistotestaus. Aina vain monimutkaistuvia tietokoneohjelmia piti siis alkaa testaamaan. Tätä varten laadittiin suunnitelmat, jossa aikataulutettiin testaus ja määritettiin testattavat alueet. Lisäksi kehitettiin tarkistuslistat, johon merkittiin jo testatut alueet. (Kasurinen 2013, 11.)

Testauksen tärkeydestä huolimatta se jää vieläkin nykyaikana liian vähälle huomiolle ohjelmistoprojektin aikana. Useasti testaus jää vasta ohjelmistoprojektin loppuvaiheille, jolloin suurin osa projektiin varatuista rahoista on jo käytetty. Tällöin virheiden korjaaminen voi jäädä toteuttamatta, koska

usein ohjelmisto alkaa tuottamaan rahaa vasta kun se siirtyy markkinoille asiakaskunnan käyttöön. (Kasurinen 2013, 16.)

2.2 Testaus ohjelmistoprojektissa

Testausprosessissa on useasti mukana kolme tahoa, jotka ovat esimies, testaajat ja kehittäjät. Esimies toimii testipäällikkönä ja mahdollisesti koko projektin päällikkönä. Testaajat sen sijaan hoitavat nimensä mukaisesti testauksen, mutta voivat tehdä projektissa muitakin työtehtäviä, kuten käyttöliittymäsuunnittelua ja ohjelmointia. Kehittäjät puolestaan tekevät ohjelman varsinaisen kehittämisen, mutta he voivat myös toimia projektin pääasiallisina testaajina. (Kasurinen 2013, 58.)

Testausprojekti aloitetaan siitä, että testipäällikkö, kehittäjät ja testaajat sopivat testaussuunnitelman. He luovat ensimmäiset testitapaukset suunnitelman pohjalta, sekä päättävät kuka tekee mitäkin. Kehittäjät alkavat toteuttaa järjestelmän komponentteja, sekä tekevät niille yksikkötestauksia. Testaajat ja kehittäjät suunnittelevat samanaikaisesti uusia testitapauksia jotka täydentävät olemassa olevaa testaussuunnitelmaa. Lisäksi testaajat testaavat komponentteja jo luoduilla testitapauksilla. (Kasurinen 2013, 59.)

Jos testi päättyy virheeseen, niin testaajat dokumentoivat virheen ja arvioivat virheen vakavuuden. Virheilmoitusten perusteella kehittäjät korjaavat järjestelmästä löytyviä virheitä, sekä jatkavat järjestelmän kehittämistä luomalla uusia komponentteja. Valmiit komponentit integroidaan järjestelmään ja ne testataan integrointitesteillä. Virheiden sattuessa tapaukset dokumentoidaan ja ongelmat korjataan. (Kasurinen 2013, 59.)

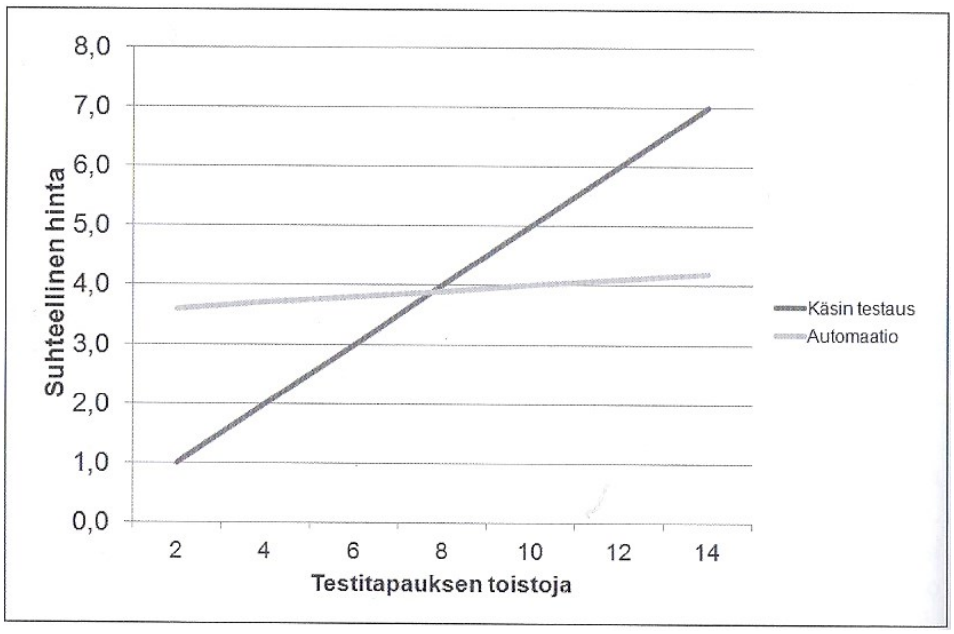
Järjestelmätestaukseen siirrytään kun kaikki komponentit on valmiita. Lopulta kun järjestelmätestauksessa ei enää löydy merkittäviä vikoja, niin siirrytään hyväksymistestaukseen. Tässä vaiheessa asiakas tekee päätöksen siitä, onko järjestelmä riittävän hyvä. Jos hän hyväksyy järjestelmän, niin järjestelmä todetaan valmistuneeksi. Muussa tapauksessa kehitystä jatketaan. Projektin lopuksi koostetaan loppuraportti, joka arkistoidaan, siltä varalta että järjestelmän pariin joudutaan vielä palaamaan jossain toisessa projektissa. (Kasurinen 2013, 59.)

3 TESTAUSAUTOMAATIO

Testausautomaatio on testauksen muoto, jossa ohjelman testaamista suoritetaan automaattityövälineellä. Tarkoituksena on automatisoida joukko toistuvasti tapahtuvia testitapauksia. Tällä pyritään vapauttamaan testaajat muihin tehtäviin. Esimerkiksi usein toistuvat testit voidaan ajaa yön aikana. Tämän ansiosta kehittäjät ja testaajat voivat aloittaa työpäivän tutkimalla testauksen tuloksia, sekä tehdä mahdollisia korjauksia ohjelmistoon. (Kasurinen 2013, 76.)

Parhaiten testausautomaatio sopii yksikkötestaukseen, jossa testataan yksittäistä moduulia. Yksikkötestauksessa lähetetään joukko kutsuja moduuliin, jonka seurauksena moduuli reagoi käskyihin. Toinen perinteinen kohde automaatiotestaukselle on käyttöliittymätestaus, jossa toimintojennauhoitusohjelma ajaa sarjan siihen nauhoitettuja käskyjä. Näitä ovat esimerkiksi tiedoston tallentaminen, hakukoneen käyttäminen jne. Automaatiotestausta voidaan tarvittaessa käyttää myös järjestelmän rasitustestauksessa. (Kasurinen 2013, 71-72, 76, 78.)

Testausautomaatio ei kuitenkaan korvaa käsin testaamista. Se on tarkoitettu vähentämään käsin tehtävää testausta mutta ei poistamaan sitä. Mitä useammin sama testitapaus ajetaan, niin sitä järkevämpää on sen automatisointi. Tällöin saavutetaan kustannustehokkuutta (kuva 1), vaikka automaatiotestauksen rakentaminen aiheuttaakin lisävaivaa. Jos testitapausta joudutaan toistamaan yli kahdeksan kertaa, niin testitapauksen automatisointi on taloudellisesti järkevää. Aina testin automatisointi ei ole kuitenkaan mahdollista. (Kasurinen 2013, 77-78.)



Kuva 1. Testauksen suhteelliset kustannukset: mitä useammin sama testitapaus ajetaan, niin sitä kannattavammaksi tulee automaatiotestaus. (Kasurinen 2013, 78.)

Lyhyesti sanottuna, automaatiotestauksella pyritään varmistamaan, että aikaisemmin toimineet osat eivät ole rikkoutuneet kehitysprosessin aikana.

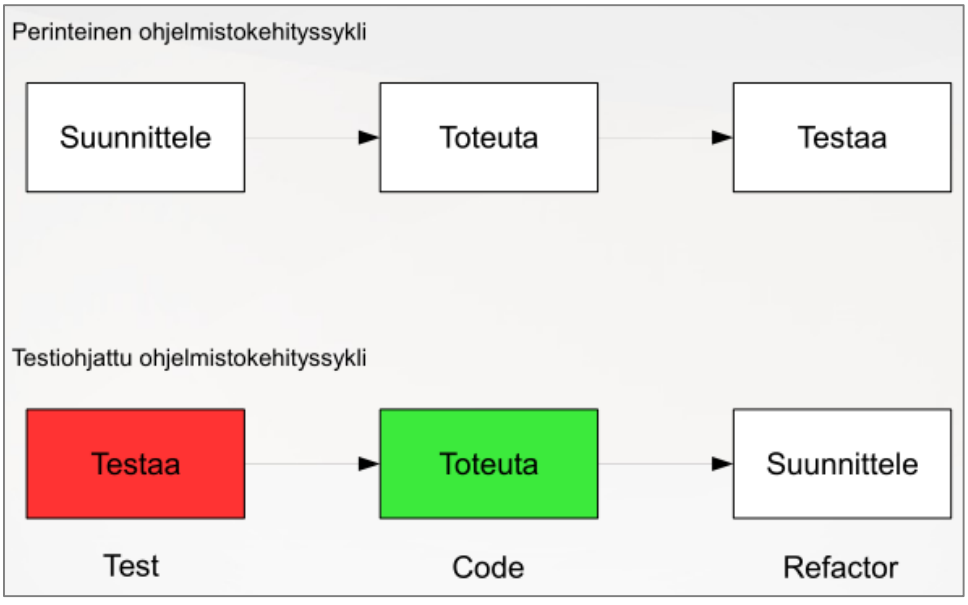
Käsin tehtävällä testauksella sen sijaan pyritään rikkomaan ohjelma, jotta löydetään piilossa olevat virheet. (Kasurinen 2013, 78.)

4 TESTIVETOINEN KEHITYS

Testivetoinen kehitys (Test-driven development) on ohjelmiston kehitysprosessi, jonka kehitti Kent Beck. Testivetoinen kehitys perustuu useisiin lyhytkestoisiin ja jatkuviin kehityssykleihin, joissa testaaminen on määräävässä asemassa. Ensimmäinen toteutettava asia on testitapaus. Tämän jälkeen kehitetään ohjelmaa niin pitkälle, että se toteuttaa testitapauksen onnistuneesti. Lopuksi koodi refaktoroidaan, eli uudelleen kirjoitetaan teknisesti paremmaksi ilman ulospäin näkyviä vaikutuksia. Lopuksi siirrytään uuteen työvaiheeseen projektissa, kun toteutus on riittävän onnistunut. Tällä yksinkertaisella työskentelytavalla kannustetaan kirjoittamaan puhtaampaa, paremmin suunniteltua, sekä helpommin ylläpidettävää koodia. Tästä seuraa huomattavasti vähäisempi virheiden määrä. TDD-mallin pääasiallisena tarkoituksena on siis laadun varmistaminen. (Kasurinen 2013, 148; Smart 2015, 12; Ye 2013, 5; Collin 2010.)

4.1 TDD-mallin perusidea ja kehityssykli

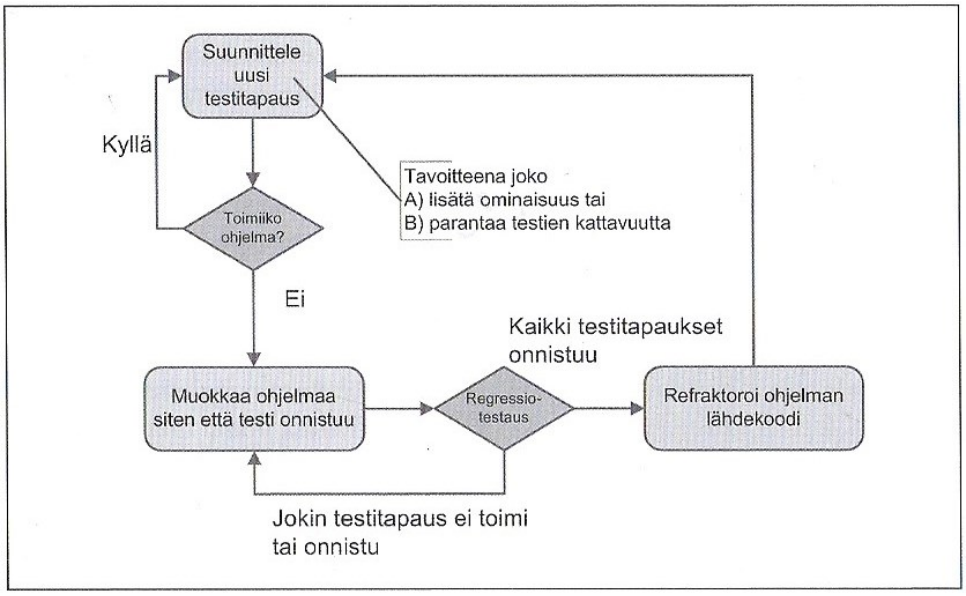
Perinteisessä ohjelmistokehityssyklissä, kuten vesiputousmallissa, ohjelmisto suunnitellaan ensin, sitten se toteutetaan ja vasta lopuksi se testataan (Kasurinen 2013, 12.). Perinteiseen ohjelmistokehityssykliin verrattuna testivetoinen ohjelmistokehityssykli tapahtuu ikään kuin väärinpäin (kuvio 1). Ensin luodaan testi, sitten toteutetaan ohjelmisto ja lopuksi suunnitellaan.



Kuvio 1. Perinteisen ohjelmistokehityssyklin ja testivetoisen ohjelmistokehityssyklin erot. (Collin 2010.)

Ohjelmistokehittäjä Robert C. Martin (2009, 122.) on kehittänyt kolme lakia, jotka hänen mukaansa kertovat testivetoisen kehityksen perusidean. Ville Karjalainen (2013) on suomentanut nämä kolme lakia. Ensimmäisessä laissa todetaan, että koodia ei saa kirjoittaa ennen kuin sitä vastaan on kirjoitettu testi. Kirjoitetaan siis ensin testi ja vasta sitten ohjelmistokoodi. Toisessa laissa todetaan että testin kirjoittaminen täytyy lopettaa heti, kun testi

testaa yhden asian eikä mene läpi. Eli kirjoitetaan testi joka testaa vain yhden testattavan asian, eikä useita asioita kerralla. Kolmannessa ja viimeisessä laissa todetaan, että koodin kirjoittaminen täytyy lopettaa heti, kun testi menee läpi. Ei siis lisätä uusia toiminnallisuuksia. Ainoastaan on sallittua koodin refaktorointi. Alla olevassa kuvassa (kuva 2) käydään vielä tarkemmin läpi testivetoisen kehittämisen toimintamalli.



Kuva 2. Tarkempi kuvaus testivetoisen kehityksen toimintamallista (Kasurinen 2013, 149.)

4.2 Hyödyt ja haasteet

TDD-mallin mukainen kehittäminen tuo mukanaan paljon hyötyä. Testivetoisen kehityksen myötä kerralla toteuttavien muutosten määrä pysyy pienenä, jotta testitapaukset pystytään ajamaan onnistuneesti. Lisäksi ohjelman rakentamiseen ei tarvita välttämättä kattavaa suunnitelmaa, koska ohjelma voidaan rakentaa ominaisuus ja toiminnallisuus kerrallaan. Tästä aiheutuu ohjelman arkkitehtuurin yksinkertaistuminen. Tietenkin tarvitaan jonkinlainen alustava suunnitelma koko ohjelmistolle, mutta ei täysin kattavaa suunnitelmaa. Lisäksi toteutettava ohjelma on jatkuvasti ajan tasalla. Projektissa ei tarvitse tehdä erillistä laajamittaista kehitysversioiden yhdistämistä tai refaktorointia, koska sitä tehdään jatkuvasti aina uuden testitapauksen myötä. Kaikki tämä kannustaa tekemään uudelleen kierrätettäviä komponentteja, joita voidaan käyttää myöhemmissä projekteissa. Tästä tulee myöhemmissä projekteissa kustannussäästöjä. (Kasurinen 2013, 149.)

Haittapuolena testivetoisessa kehityksessä on luottaminen yksikkötestaukseen, joka ei ole kattava tapa testata kokonaista tuotetta. Ongelmia aiheuttaa myös huonosti suunnitellut yksikkötestit, jolloin virheet saattavat jäädä huomaamatta. Tämä voi johtaa ongelmatilanteessa järjestelmän rakenteen huomattaviin muutoksiin, kun järjestelmää testataan myöhemmässä vaiheessa kokonaisuutena. Testivetoinen kehitysmalli tuottaa myös paljon testattavaa, koska testejä kirjoitetaan tusinoittain päivässä, sadoittain kuukaudessa ja tuhansittain vuodessa. Suurissa järjestelmissä testivetoinen kehitys

voi kasvattaa testausmäärää eksponentiaalisesti, Testejä saattaa olla yhtä paljon kuin varsinaista ohjelmistokoodia, jolloin niiden hallinnointi voi olla haastavaa. Lisäksi testivetoinen kehitysmalli ei sovellu suurelle organisaatiolle. Sen sijaan testivetoinen kehitys sopii alle viidentoista hengen tiimeille. Suuren koko luokan projektit, monimutkaiset projektit tai projektit joissa on paljon ulkoistettuja resursseja, eivät ole myöskään testivetoiseen kehitykseen sopivia. Kaikesta huolimatta testivetoiseen kehittämiseen löytyy useille ohjelmointikielille työkaluja. (Kasurinen 2013, 149-150; Martin 2009, 123.)

5 KÄYTTÄYTYMISLÄHTÖINEN KEHITYS

Käyttäytymislähtöisessä kehityksessä (Behavior-driven development) pyritään kuvaamaan ohjelman käyttäytymistä sidosryhmien näkökulmasta. Sidosryhmällä tarkoitetaan ketä tahansa, joka on kiinnostunut ohjelmasta. Sidosryhmät jaetaan kahteen ryhmään. Ydinosapuoli (core stakeholders) ja oheisosapuoli (incidental stakeholders). Ydinosapuoli on ryhmä, jonka ongelmaa yritetään ratkaista. Oheisosapuoli on sen sijaan ryhmä, johon kuuluvat kaikki jotka yrittävät ratkaista ongelmaa. Sidosryhmien avulla on tarkoitus ymmärtää maailmaa heidän näkökulmastaan. Millainen heidän liiketoimintansa on? Mitä haasteita he kohtaavat? Mitä mahdollisuuksia he kohtaavat? Millä sanoilla he kuvaavat ohjelman haluttua käyttäytymistä? Tämä perustuu liiketoimintavetoiseen kehitykseen (Domain-driven design). (Chelimsky, Astels, Dennis, Hellesøy, Helmkamp, & North 2010, 138-139.)

5.1 BDD-mallin perusidea ja kehityssykli

Chelimksy ja kumppanit toteavat (2010, 138-139.), että käyttäytymislähtöisen kehityksen perimmäisenä ideana on rakentaa ohjelma, jolla on arvoa sidosryhmille. Ville Karjalainen (2013) on suomentanut heidän kolme periaatetta, jotka tiivistävät käyttäytymislähtöisen kehityksen perusidean. Ensimmäinen periaate on ”Tarpeeksi on tarpeeksi”, jonka mukaan liiallinen suunnittelu on hukkaan heitettyä aikaa. Tehdään vain sen verran kuin on tarvetta, eikä yhtään enempää. Seuraava periaate on ”Toimita osapuolille sitä, mitä he oikeasti haluavat”. Rakennetaan vain asioita joista on hyötyä sidosryhmille, joko heti tai myöhemmin. Viimeinen periaate on ”Kaikki on käyttäytymistä”. Kaikessa on kyse käyttäytymisestä. Käytetään siis samaa ajattelutapaa ja kieltä kuvaamaan ohjelma oli se sitten koodin tasolla, sovelluksen tasolla, dokumentoinnin tasolla, asiakasrajapinnassa jne.

BDD-mallin kehityssykli ei kaikesta huolimatta juuri eroa TDD-mallista. Ensimmäiseksi luodaan testitapaus, jossa kerrotaan halutun ominaisuuden käyttäytyminen. Eli mitä pitäisi tapahtua. Seuraavaksi tehdään varsinainen toiminnallisuus koodaamalla. Lopuksi refaktoroidaan ohjelman koodi, jos testi meni onnistuneesti läpi. BDD-mallin kehityssyklin on siis melkein samanlainen kuin testivetoisessa kehityksessä. Ainoastaan erona on se, että puhutaan käyttäytymisestä eikä niinkään testeistä. (Ye 2013, 7.)

Kaikista tärkein BDD:n ominaisuus on kuitenkin yhtenäinen- ja selkeäkieli eri sidosryhmien väliseen kommunikointiin. Näin liike-elämä ja kehittäjät puhuvat samaa kieltä. Yksinkertaistettuna käyttäytymislähtöisessä kehityksessä on tarkoitus saada kaikki osapuolet tekemään parempaa yhteistyötä yhteisen kielen kautta. (Smart 2015, 4, 12.)

5.2 BDD-malli pyrkii korjaamaan TDD-mallin ongelmia

Dan North (North n.d.) kehitti käyttäytymislähtöisen kehityksen korjaamaan testivetoisen kehittämisen ongelmakohtia. Kehittäjät halusivat nimittäin tietää mistä pitäisi aloittaa, mitä pitäisi testata ja mitä ei, kuinka paljon

yhden testin pitäisi testata, miten nimetä testit, sekä miksi testi ei mene läpi (Karjalainen 2013).

Testivetoisessa kehittämisessä testien nimeämiskäytännössä sekaannusta aiheuttaa nimenomaan testi-sanan käyttö, joilla nimetään luokat ja metodit. Esimerkiksi: ”TilinHallintaTestiLuokka”. Käyttäytymislähtöisessä kehittämisessä käytetään sen sijaan lauseita kuvastamaan käyttäytymistä. Esimerkiksi: ”Tilinomistaja nostaa rahaa”. Tällä yksinkertaisella muutoksella on Dan Northin mukaan merkittävä vaikutus. Dokumentointi selkeytyy huomattavasti muiden sidosryhmän jäsenten näkökulmasta. Generoitua dokumentointia pystyvät tämän ansiosta ymmärtämään myös muut sidosryhmien jäsenet, kuten liike-elämän edustajat. (Karjalainen 2013; North n.d.)

Käyttäytymislähtöisessä kehittämisessä testit saa paremmin kohdistettua haluttuun ominaisuuteen yksinkertaisen lausesapluunan avulla. Siinä todetaan, että tämän luokan pitäisi tehdä jotakin. Erityisesti lauseessa painotetaan pitäisi-sanaa. Tämä pakottaa suunnittelemaan testin vain kyseiselle luokalle. Jos luokka tekee enemmän kuin yhden asian, niin silloin lähes aina pitää siirtää nämä ylimääräiset asiat uusiin luokkiin. Tällä tavoin testit pysyvät paremmin hallittavissa. (North n.d.)

Käyttäytymislähtöisessä kehittämisessä voidaan myös paremmin hahmottaa miksi testi ei mene läpi, koska tiedetään mitä metodin tai luokan pitäisi tehdä. Tämän ansiosta on helpompaa löytää virheen sijainti. Näin voidaan paremmin ymmärtää johtuuko virhe ohjelmistovirheestä vai onko aikaisempi oletus järjestelmän käyttäytymisestä väärä. (North n.d.)

Lisäksi käyttäytymislähtöisessä kehittämisessä tiedetään mistä kehittäminen pitäisi aloittaa. Vastaus ongelmaan löytyy kysymyksen muodossa. What’s the next most important thing the system doesn’t do? Eli mitä seuraavaksi tärkeää asiaa järjestelmä ei vielä tee. Kysymys pakottaa määrittelemään arvon ominaisuuksille ja priorisoimaan niistä tärkeimmän, jota ei ole vielä toteutettu. Tämä auttaa myös löytämään ominaisuudelle kuvaavan nimen. (North n.d.)

BDD:n kehittäminen jatkui kuitenkin vielä tästä eteenpäin, vaikka TDD:n ongelmiin oli jo löydetty vastaukset. Kehityksen lopulla syntyi yhtenäinen kieli sidosryhmien väliseen kommunikointiin. Tämä on mahdollista tarinan eli selkokielisen testin avulla. Tarinan on oltava tarpeeksi väljä, että se ei tunnu liian keinotekoiselta tai sitovalta muiden sidosryhmien jäsenten kannalta. Samalla sen pitää olla kuitenkin tarpeeksi rakenteellinen, jotta tarina voidaan paloitella pienempiin osa-alueisiin automaatiotestausta varten. Tarinan koostumuksesta kerrotaan tarkemmin seuraavassa alaluvussa. (North n.d.; Karjalainen 2013; Karkinen 2013.)

5.3 Tarina eli selkokielinen testitapaus

Käyttäytymislähtöisessä kehittämisessä testitapauksilla tarkoitetaan tarinoita. Tarinat kirjoitetaan ihmisten ymmärtämässä muodossa, joissa konkretisoidaan halutut vaatimukset. Tietokone pystyy kuitenkin lukemaan tarinat, koska ne noudattavat tiettyä syntaksia. Tämän takia automaatiotestaus

on mahdollista, kun käytetään sopivia työkaluja. (Smart 2015, 19-20; Karkinen 2013.)

Tarinan rakenne pysyy muodossa sapluunan avulla, joka koostuu useista eri komponenteista. Otsikko-komponetti (story) kertoo mistä tarinasta on kyse, koska tarinoita voi olla useita. Kerronta-komponentti (narrative) kertoo tarinan sisällön. Hyväksymiskriteeri-komponentti (acceptance criteria) lopettaa tarinan kertomalla oikean vastauksen. (Chelimsky ym. 2010, 146; North n.d.)

Seuraavassa on näyte tarinan sapluunasta, jota tullaan käyttämään tässä opinnäytetyössä:

```
01 Story: (tarinan otsikko)
02
03 Narrative:
04 In order to <hyöty>
05 As a <rooli>
06 I want <toiminnallisuus>
07
08 Acceptance Criteria: (hyväksymiskriteeri)
09
10 Scenario: (skenaarion otsikko)
11 Given <lähtökohta>
12 When  <tapahtuma>
13 Then  <lopputulos>
```

Northin (North n.d.) esimerkkiä mukaillen tarina voisi olla seuraavanlainen:

```
01 Story: Account Holder withdraws cash
02
03 Narrative:
04 In order to <get money when the bank is closed>
05 As an <Account Holder>
06 I want <to withdraw cash from an ATM>
07
08 Acceptance Criteria:
09
10 Scenario 1: Account has sufficient funds
11 Given <the account balance is "€100">
12 When  <the Account Holder requests "€20">
13 Then  <the ATM should dispense "€20">
```

Tarinan otsikon (rivi 01) pitää kuvastaa aktiviteettia, jossa joku tekee jotain. Esimerkiksi: "Tilin omistaja nostaa rahaa.". Rahan nosto ei kuitenkaan onnistu, ennen kuin tämä toiminto on rakennettu. Kun toiminto on toteutettu, niin rahan nosto onnistuu. Tämä helpottaa ymmärtämään milloin toiminto on valmis. Jos käytettäisiin otsikkoa "Tilin hallinta" niin olisi paljon vaikeampaa ymmärtää milloin toiminto on valmis. Tämän otsikon alle voi kätkeytyä monia toimintoja, kuten pankkikortin tunnuskoodin vaihtaminen. (North n.d.)

Kerronnan ensimmäisellä rivillä (rivi 04) kerrotaan mitä hyötyä aiotaan saavuttaa. Erittelemällä hyödyn tarinan kirjoittaja joutuu miettimään toiminnon hyödyllisyyden. Jos toiminnosta ei ole hyötyä, niin se tarkoittaa yleensä että

jokin tarina on unohtunut. Seuraavalla rivillä (rivi 05) rooli kertoo kuka tarvitsee toimintoa. Kolmannella rivillä (rivi 06) kerrotaan millä toiminnallisuudella saavutetaan hyöty. Kerronnan päätähtäimenä on siis hyödyn saavuttaminen. (Smart 2015, 36-37; North n.d.)

Kerronnan (rivi 03–06) sapluunassa on lähteistä riippumatta hieman eroja, sillä komponenttien järjestys voi vaihdella (Smart 2015, 36; North n.d). Tällä ei ole kuitenkaan vaikutusta, koska kerronnan osuutta automaatiotestaustyökalut eivät tulkitse mitenkään. Kerronta on kuitenkin käyttäytymislähtöiseen kehittämiseen kuuluva perusasia. Ilman kerrontaa muiden sidosryhmien olisi vaikeata tai jopa mahdotonta ymmärtää mitä ominaisuudelta halutaan. (Quick Intro to Behat n.d.)

```
08 Acceptance Criteria:
09
10 Scenario 1: Account has sufficient funds
11 Given <the account balance is \€100>
12 When  <the Account Holder requests \€20>
13 Then  <the ATM should dispense \€20>
```

Hyväksymiskriteeri (rivi 08–13) on varsinainen osuus tarinassa, jonka automaatiotyökalu lukee ja tulkitsee. Hyväksymiskriteerit muodostuvat yhdestä tai useammasta skenaarioista. Nämä ovat eräänlaisia käytännön esimerkkejä, joilla hahmotetaan mitä pitäisi tapahtua.

Skenaario aloitetaan otsikolla (rivi 10), joka kertoo mitä on tarkoitus tapahtua. Seuraavaksi (rivi 11–13) tulee joukko testiaskeleita (step definitions). Jokainen testiaskel alkaa jollakin seuraavista avainsanoista: Given, When ja Then.

Given-avainsana kertoo skenaarion lähtökohdat. Eli valmistelee testiympäristön. When-avainsana kertoo skenaarion päätehtävän. Eli mitä käyttäjä tekee. Then-avainsana kertoo lopputuloksen. Eli mitä pitäisi tapahtua. Näiden lisäksi voidaan käyttää And- ja But-avainsanoja antamaan lisämääritteitä Given, When ja Then testiaskeleihin. (Karjalainen 2013; Karkinen 2013; Smart 2015, 20; Intro to Behat n.d.)

Tarina voi myös koostua useasta skenaariosta. Tästä on hyvä esimerkki Dan Northin (North n.d.) testitapauksessa, jossa nostetaan rahaa pankkiautomaatista.

```
Story: Account Holder withdraws cash

As an Account Holder
I want to withdraw cash from an ATM
So that I can get money when the bank is closed

Scenario 1: Account has sufficient funds
Given the account balance is \$100
 And the card is valid
 And the machine contains enough money
When the Account Holder requests \$20
Then the ATM should dispense \$20
 And the account balance should be \$80
 And the card should be returned
```

```
Scenario 2: Account has insufficient funds
Given the account balance is \$10
 And the card is valid
 And the machine contains enough money
When the Account Holder requests \$20
Then the ATM should not dispense any money
 And the ATM should say there are insufficient funds
 And the account balance should be \$10
 And the card should be returned

Scenario 3: Card has been disabled
Given the card is disabled
When the Account Holder requests \$20
Then the ATM should retain the card
And the ATM should say the card has been retained

Scenario 4: The ATM has insufficient funds
...
```

Skenaarioiden pitäisi erota jo otsikkotasolla toisistaan. Esimerkissä skenaarionotsikot kertovat ainoastaan mitä eroa eri skenaarioilla on. Ei tarvitse sanoa: "Tilinomistaja nostaa rahaa tililtä jossa ei ole tarpeeksi varoja. Tämän vuoksi hänelle kerrotaan että nosto ei ole mahdollista." Otsikosta näkee jo mitä skenaariossa on tarkoitus tapahtua, sillä skenaarionotsikko toimii tarkentavan otsikkona tarinanotsikolle. Koska tarinanotsikko on: ” Tilinomistaja nostaa rahaa”, niin skenaarion otsikoksi riittää: ”Tilillä ei ole tarpeeksi rahaa”. (North n.d.)

5.4 Hyödyt ja haasteet

Käyttäytymislähtöisen kehityksen hyödyt ja haitat ovat lähes samat kuin testivetoisessa kehityksessä. Suurimpana erona on kommunikointi, joka tuo sekä hyötyä että haasteita riippuen kuinka hyvin sen käytössä onnistutaan. Kommunikoinnilla tarkoitetaan BDD-mallissa yhtenäistä ja selkeää viestintää kaikkien sidosryhmien välillä. Näin saadaan kaikki sidosryhmät sitoutumaan yhteisen vision eteenpäin viemiseen, niin että kaikki osapuolet ymmärtävät mitä tehdään. Kuitenkin tätä etua voi olla vaikea hyödyntää, jos asiakas ei ole kykenevä tai halukas kommunikoimaan vaatimuksiaan. Ilman kommunikointia on vaikeata saada irti kaikkea käyttäytymislähtöisestä kehittämisestä, koska ei voida hyödyntää selkokielisiä testitapauksia ja tämän myötä selkeää dokumentointia. (North 2013; Smart 2015, 28-30.)

Käyttäytymislähtöisessä kehittämisessä on kuitenkin yhtenäisen ja selkeän kommunikoinnin lisäksi muitakin etuja testivetoiseen kehittämiseen verrattuna. Aikaisemmassa luvussa 5.3 kerrottiin, että BDD-malli pyrkii vastaamaan TDD-mallin ongelmiin. Testivetoisen kehittämiseen nähden käyttäytymislähtöisessä kehittämisessä kehittäjät tietävät mistä pitäisi aloittaa ohjelman kehitys. On myös selvää mitä pitäisi testata ja mitä ei. Lisäksi osataan määrittää kuinka paljon yhden testin pitäisi testata. Muutenkin testauksessa tiedetään miksi testi ei mene läpi. Lisäksi testit on nimetty selkeästi. (Karjalainen 2013; North n.d.)

Yhtenäisiä etuja ja haasteitakin löytyy. Testivetoisen kehitysmallin tavoin käyttäytymislähtöinen kehitys sopii parhaiten ketterän ohjelmistokehitysmalliin. On nimittäin vaikeata tai jopa mahdotonta määritellä kaikki ohjelmiston vaatimukset etukäteen. Ketterän ohjelmistokehityksen ja automaatiotestien vuoksi voidaan vähentää turhaa työtä, säästää kustannuksissa, tehdä helpompia ja turvallisempia muutoksia sovellukseen, sekä julkaista näitä muutoksia nopeammin. (Kasurinen 2013, 149-150; Smart 2015, 28-30.)

Ketterän ohjelmistokehitysmallin vuoksi BDD-malli ei kuitenkaan toimi kunnolla isoissa organisaatioissa. Liike-elämän edustajat, kehittäjäpuoli ja testauspuoli saattavat kaikki olla omissa lokeroissaan jolloin kommunikointia ei juuri synny. Samaa ongelmaa esiintyy myös organisaatioissa joissa kehittämistä on osittain tai kokonaan ulkoistettu. Parhaiten BDD toimii pienissä tiimeissä, joissa kaikki osapuolet osallistuvat ohjelmiston suunnitteluun alusta alkaen. (Kasurinen 2013, 149-150; Smart 2015, 28-30.)

Testivetoisessa kehittämisessä voi tämän lisäksi ongelmia aiheuttaa huonosti suunnitellut testit. Käyttäytymislähtöinen kehitys ei ole poikkeus tässä. Huonosti kirjoitetut testit hidastavat varsinkin isoja ja monimutkaisia ohjelmistoprojekteja. Jos testejä ei ole suunniteltu huolellisesti, niin niiden ylläpitäminen voi olla vaikeata. (Kasurinen 2013, 149-150; Smart 2015, 28-30.)

BDD-malli on siis osittain samanlainen kuin TDD-malli, mutta erojensa vuoksi ei voida kuitenkaan puhua TDD-mallin evoluutiosta. Dan Northin (2013) mukaan BDD-malli näyttää silloin samanlaiselta kuin TDD-malli, jos kaikki projektin jäsenet ovat ohjelmoijia. Tällöin projektin jäsenet puhuvat samaa kieltä ja tietävät silloin mitä pitää tehdä. Ongelmia tulee sen sijaan heti kun yhtälöön tuodaan yksikin ulkopuolinen edustaja, kuten liike-elämän edustaja. Hän ei nimittäin puhu samaa kieltä ohjelmoijien kanssa. Tällöin BDD-malli näyttää todellisen luonteensa. Siitä tulee kommunikointiväline, jonka avulla luodaan yksi yhtenäinen visio projektinjäsenten kesken. Tätä yhteistä visiota käydään tavoittelemaan. Eli BDD-malli on erilainen ja isompi asia kuin vain evoluutio TDD-mallista.

6 BEHAT

Behat on avoimen lähdekoodin automaatiotestaustyökalu, joka mahdollistaa käyttäytymislähtöisen kehityksen. Työkalu on tarkoitettu käytettäväksi PHP:n versioiden 5.3 ja 5.4 kanssa. Työkalulla kirjoitetaan BDD-mallin mukaisia selkokielisiä tarinoita, joita sitten testaan. (Behat Documentation n.d.; Quick Intro to Behat n.d.)

Behat-työkaluun päädyttiin harjoittelupaikan suosituksesta, koska käytetty ohjelmointikieli on PHP. Toinen vaihtoehto olisi ollut Codeception, mutta se sisälsi paljon ylimääräisiä ominaisuuksia joille ei ollut tässä työssä käyttöä (Codeception: Modern PHP testing for everyone. n. d.). Valinta kohdistui tämän vuoksi Behatiin, koska se keskittyy ainoastaan BDD-mallin mukaiseen testaukseen. Näin pysyttiin paremmin opinnäytetyön rajauksen sisällä.

Behat on komentokehotepohjainen työkalu. Ajettaessa Behat lukee spesifikaatiot selkokielisestä tekstitiedostosta nimeltään features. Behat etsii tiedostosta skenaariot testausta varten ja ajaa ne. Jokainen skenaario on lista testiaskeleita (step definitions), joita Behatin pitää käydä läpi. Syntaksin pitää noudattaa perussyntaksisääntöjä, jotta Behat voi ne ymmärtää. Perussyntaksi on Behatin kohdalla lähes samanlainen kuin mitä luvussa 5.4 on esitelty selkokielisten tarinoiden kohdalla.

Skenaariot koostuvat testiaskeleista, jotka kertovat selkokielisesti mitä kussakin askeleessa pitäisi Behatin tehdä. Jos testiaskeleen koodissa ei ole virhettä, niin Behat siirtyy seuraavaan askeleeseen skenaariossa. Jos Behat pääsee skenaarion loppuun ilman virheitä, niin se antaa skenaariolle hyväksynnän. Jos kuitenkin jossain testiaskeleessa on virhe, niin Behat hylkää kyseisen skenaarion ja siirtyy seuraavaan skenaarioon. Testien lopuksi Behat antaa tuloksen, jossa näkyy testin läpäisseet skenaariot ja hylätyt skenaariot. Näiden perusteella voidaan tarkalleen tietää missä on virhe.

6.1 Behat BDD-testauksen alustana

Opinnäytetyön käytännön osuus toteutettiin läpikäymällä Behatin pikaoppaan mukainen harjoite (Quick Intro to Behat n.d.). Pikaohjeessa oli kuviteltu tilanne, jossa rakennetaan alkuperäistä UNIX-komentoa ls. Tällä komennolla listataan hakemiston sisältö. Ennen esimerkkiprojektin aloittamista piti kuitenkin asentaa Behat. Kohde käyttöjärjestelmänä toimi Linux-pohjainen Ubuntu, johon oli jo ennestään asennettu PHP 5.3.1. Behatin asennuksen apuna käytettiin Composeria, joka on paketinhallintajärjestelmä PHP:lle. Asennus aloitettiin tekemällä projektia varten hakemisto, johon Behat asennettiin. Tämän hakemiston juureen luotiin tiedosto composer.json komennolla:

```
$ nano composer.json
```

Composer.json on Composerin konfigurointitiedosto, jonka sisällä määritettään mitä Behatin versiota käytetään. Composer lataa ja asentaa näitä tietoja käyttäen halutun version Behatista. Konfigurointitiedoston rakenne on seuraava:

```
{
    "require-dev": {
        "behat/behat": "~2.5"
    },
    "config": {
        "bin-dir": "bin/"
    }
}
```

Konfiguraatiotiedoston luomisen jälkeen suoritettiin Composerin lataus ja asennus seuraavalla komennoilla:

```
$ curl -sS https://getcomposer.org/installer | php
$ php composer.phar install
```

Seuraavaksi ajettiin komento:

```
$ bin/behat --init
```

Tämä komento ei kuitenkaan toiminut, vaikka se oli täsmälleen sama kuin Behatin pikaoppaassa. Kokeilujen jälkeen Behatin ajo onnistui, kun sen sijainti määritettiin tarkemmin bin-hakemistoon komennolla:

```
$ bin/behat --init
```

Komento loi tämän jälkeen onnistuneesti hakemiston features, jonka sisään Behat asensi peruskomponentteja projektin aloittamista varten. Näiden toimenpiteiden jälkeen Behat oli asennettu onnistuneesti projektin hakemistoon.

6.2 Testitapauksen luonti

BDD-mallin kehityssyklin mukaisesti ensimmäisenä oli testitapauksen luonti. Behatin pikaoppaan avustuksella luontiin tarina. Edellä oli jo tehty esimerkkiprojektia varten oma hakemisto, joten seuraavaksi siirryttiin kerronnan määrittämiseen. Behatin kohdalla kerronnasta käytetään nimitystä ominaisuus (feature). Tässä tapauksessa ominaisuus oli ls-komento, joka listaa hakemiston sisällön. Kerronnan luonti aloitettiin luomalla tiedosto features/ls.feature. Tiedoston sisään määritettiin ominaisuuden nimi, siitä saatava hyöty, kenen näkökulmasta ominaisuutta halutaan, sekä mitä ominaisuuden pitäisi tehdä. Eli luotiin tarinan ensimmäinen komponentti, kerronta.

```
# features/ls.feature
Feature: ls
  In order to see the directory structure
  As a UNIX user
  I need to be able to list the current directory's contents
```

Tässä tapauksessa UNIX-käyttäjänä halutaan listata hakemiston sisältö, jotta nähdään hakemistonrakenne. Mitkään näistä tiedoista eivät ole kuitenkaan välttämättömiä Behatin toimivuuden kannalta, mutta ne ovat BDD-malliin kuuluvia perusasioita. Ilman näitä muiden sidosryhmien olisi vaikeata tai jopa mahdotonta ymmärtää mitä ominaisuudelta halutaan.

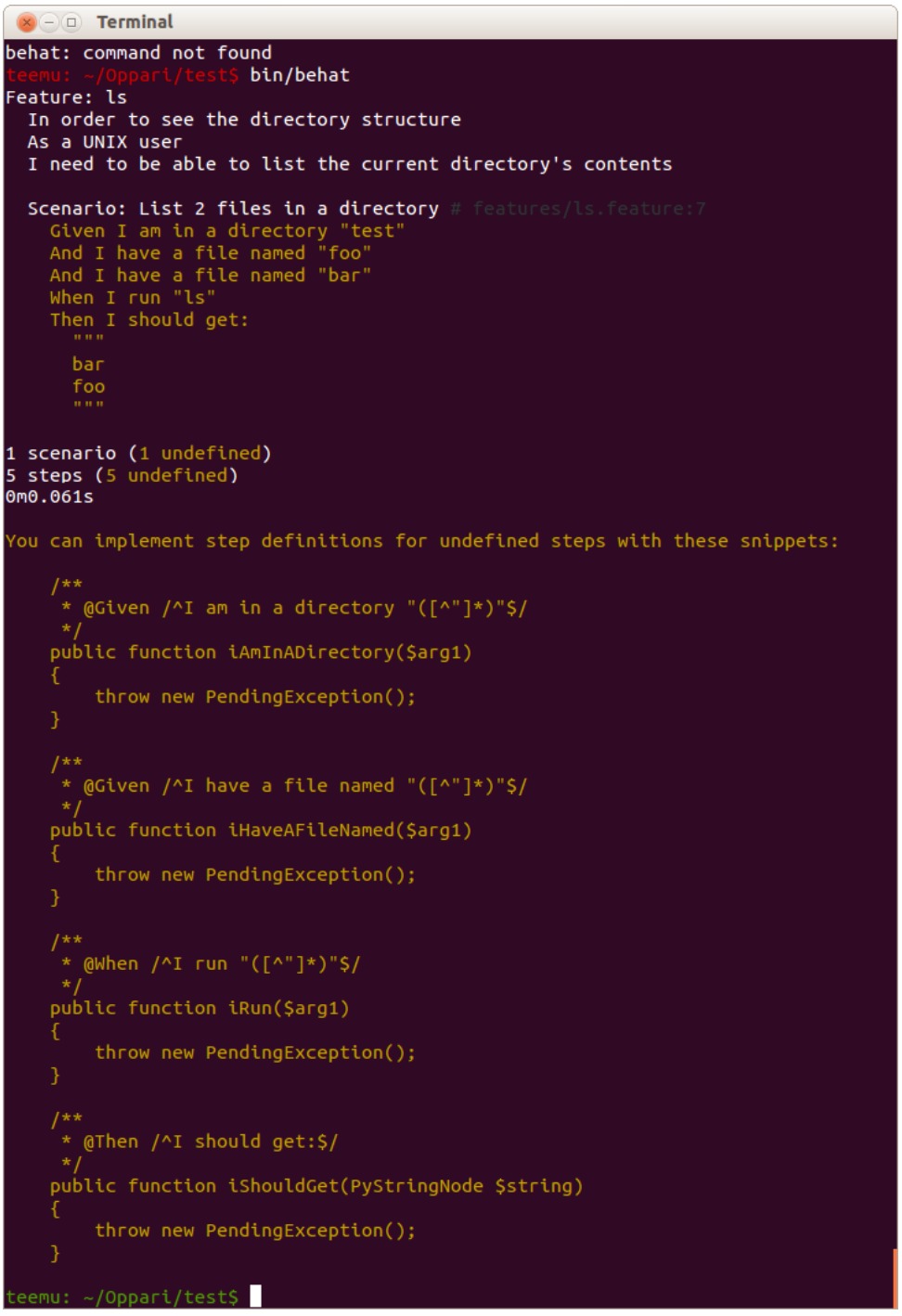
Testitapauksen luontia jatkettiin luomalla skenaario edellä mainitun features/ls.feature-tiedoston loppuun. Tämä on se varsinainen osuus tarinasta, jota Behat käyttää testeissä:

```
Scenario: List 2 files in a directory
  Given I am in a directory "test"
  And I have a file named "foo"
  And I have a file named "bar"
  When I run "ls"
  Then I should get:
    """
    bar
    foo
    """
```

Skenaariolle annettiin kuvaava otsikko: ”Listataan 2-tiedostoa hakemistossa”. Given-avainsanassa määritettiin sijainniksi test-hakemisto. Given-avainsanaa laajennettiin käyttämällä And-lisäämääritteitä. Näiden avulla kerrottiin minkä nimiset tiedostot pitäisi löytyä, kun listataan test-hakemiston sisältö komennolla ls. Kuitenkaan Given-, Then-, And-, tai muillakaan avainsanoilla ei ole mitään eroa toisiinsa nähden. Näitä vain käytetään sen vuoksi, että kaikki skenaariot olisivat helpommin luettavissa muiden sidosryhmien näkökulmasta.

6.3 Sovelluksen luonti

Näin oli luotu ensimmäinen testitapaus, joten kehityssyklin mukaisesti oli vuorossa testitapauksen ajo. Behat löysi automaattisesti tiedoston features/ls.feature ja yritti ajaa siellä olevan skenaarion testinä. Tämä päättyi kuitenkin virheeseen (kuva 3), koska varsinaista ohjelmistokoodia ei ollut vielä tehty. Esimerkiksi Behat ei tiennyt miten toimia testiaskeleen ”Given I am in a directory "test"” kohdalla. Seuraavaksi piti kertoa miten Behatin pitäisi toimia tässä tapauksessa. Tässä kohtaa Behat auttoi ehdottamalla, että esimerkiksi lausekkeesta ”Given I am in a directory "test"” tehtäisiin testiaskel (kuva 3).



Kuva 3. Testin ajaminen ennen kuin varsinaista sovelluskoodia on tehty.

Behatin ohjeissa neuvottiin avaamaan tiedosto features/bootstrap/FeatureContext.php, jonka sisältö näkyy alla.

```
#features/bootstrap/FeatureContext.php
<?php

use Behat\Behat\Context\ClosuredContextInterface,
    Behat\Behat\Context\TranslatedContextInterface,
    Behat\Behat\Context\BehatContext,
    Behat\Behat\Exception\PendingException;
use Behat\Gherkin\Node\PyStringNode,
    Behat\Gherkin\Node\TableNode;
```

```
//
// Require 3rd-party libraries here:
//
//   require_once 'PHPUnit/Autoload.php';
//   require_once 'PHPUnit/Framework/Assert/Functions.php';
//

/**
 * Features context.
 */
class FeatureContext extends BehatContext
{
    /**
     * Initializes context.
     * Every scenario gets its own context object.
     *
     * @param array $parameters context parameters (set them
up through behat.y$
     */
    public function __construct(array $parameters)
    {
        // Initialize your context here
    }

//
// Place your definition and hook methods here:
//
//    /**
//     * @Given /^I have done something with "([^"]*)"$/
//     */
//    public function iHaveDoneSomethingWith($argument)
//    {
//        doSomethingWith($argument);
//    }
//
}
```

FeatureContext.php-tiedostoon tehtiin varsinainen sovelluksen koodaus. Ensimmäisenä tiedostoon lisättiin metodi iAmInADirectory(). Lisäksi koodia selkeytettiin muokkaamalla muuttujan $argument1 nimeksi $dir.

```
<?php

use Behat\Behat\Context\ClosuredContextInterface,
    Behat\Behat\Context\TranslatedContextInterface,
    Behat\Behat\Context\BehatContext,
    Behat\Behat\Exception\PendingException;
use Behat\Gherkin\Node\PyStringNode,
    Behat\Gherkin\Node\TableNode;

/**
 * Features context.
 */
class FeatureContext extends BehatContext
{
         /**
     * @Given /^I am in a directory "([^"]*)"$/
     */
    public function iAmInADirectory($dir)
    {
        if (!file_exists($dir)) {
```

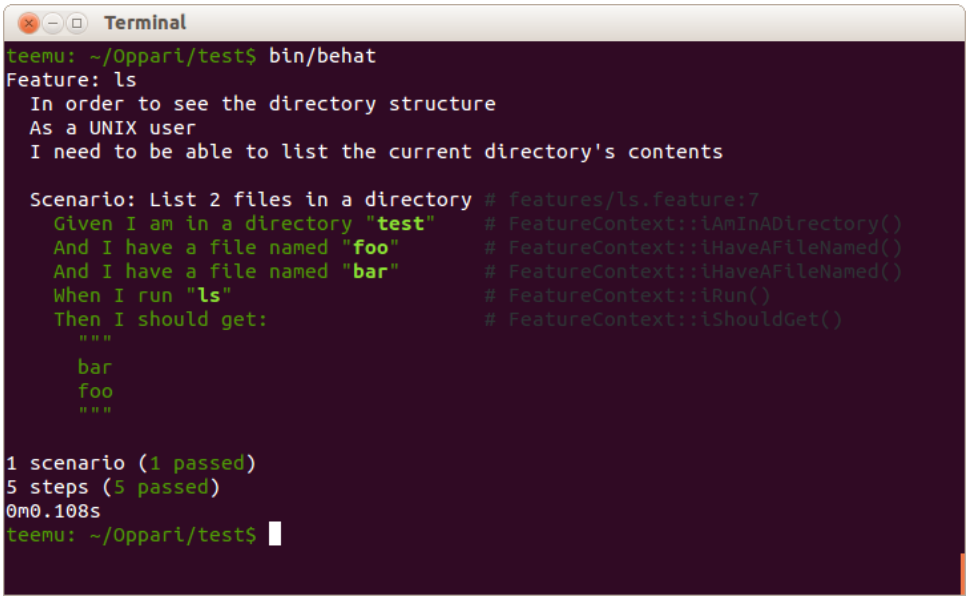
```
                mkdir($dir);
        }
        chdir($dir);
    }
}
```

Eli luotiin metodi, joka siirtyy hakemistoon tai tarvittaessa luo uuden hakemiston. Tämän jälkeen tehtiin myös muut kolme metodia Behatin neuvojen mukaisesti. Lopuksi FeatureContext.php-tiedosto näytti tältä:

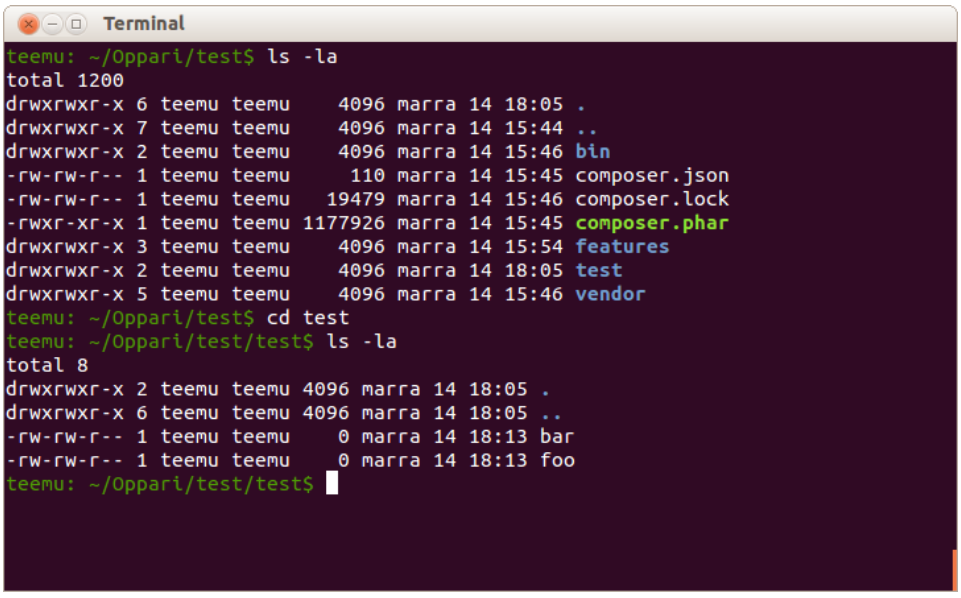
```
<?php

use Behat\Behat\Context\ClosuredContextInterface,
    Behat\Behat\Context\TranslatedContextInterface,
    Behat\Behat\Context\BehatContext,
    Behat\Behat\Exception\PendingException;
use Behat\Gherkin\Node\PyStringNode,
    Behat\Gherkin\Node\TableNode;

/**
* Features context.
*/
class FeatureContext extends BehatContext
{

        private $output;

        /**
        * @Given /^I am in a directory "([^"]*)"$/
        */
        public function iAmInADirectory($dir)
        {
                if (!file_exists($dir)) {
                        mkdir($dir);
                }
                chdir($dir);
        }

        /**
        * @Given /^I have a file named "([^"]*)"$/
        */
        public function iHaveAFileNamed($file)
        {
                touch($file);
        }

        /**
        * @When /^I run "([^"]*)"$/
        */
        public function iRun($command)
        {
                exec($command, $output);
                $this->output  =  trim(implode("\n",  $out-
put));
        }

        /**
        * @Then /^I should get:$/
        */
        public function iShouldGet(PyStringNode $string)
        {
                if ((string) $string !== $this->output) {
```

```
                        throw new Execption (
                                "Actual   output   is:\n"   .
$this->output
                        );
                }
        }
}
```

Kaikki oli tämän jälkeen valmista testitapauksen uudelleenajoa varten. Behat ajettiin siis uudelleen. Tulos oli tällä kertaa onnistunut. Testi meni läpi (kuva 4), sekä komennot oli suoritettu onnistuneesti, sillä hakemistoon oli ilmestynyt test-kansio ja sen sisään tiedostot ”bar” ja ”foo” (kuva 5).



Kuva 4. Onnistunut testitapaus.

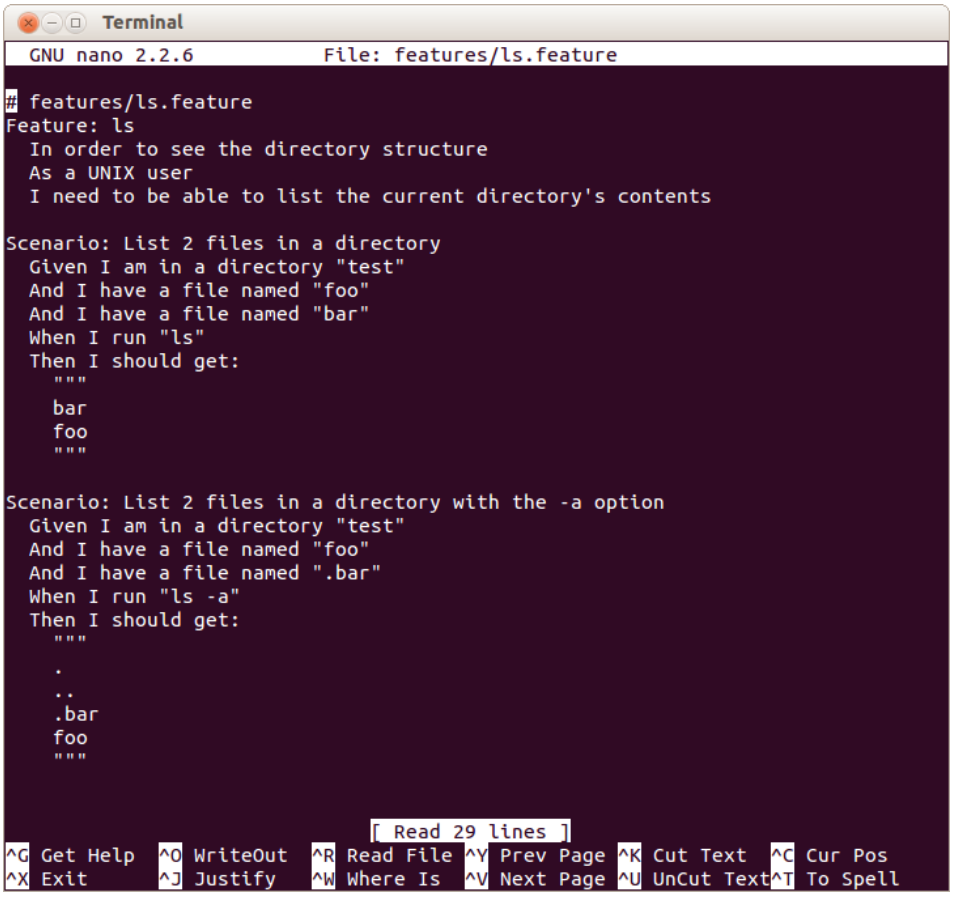


Kuva 5. Test-hakemiston sisälle luotiin tiedostot ”bar” ja ”foo”.

6.4 Toisen skenaarion lisäys

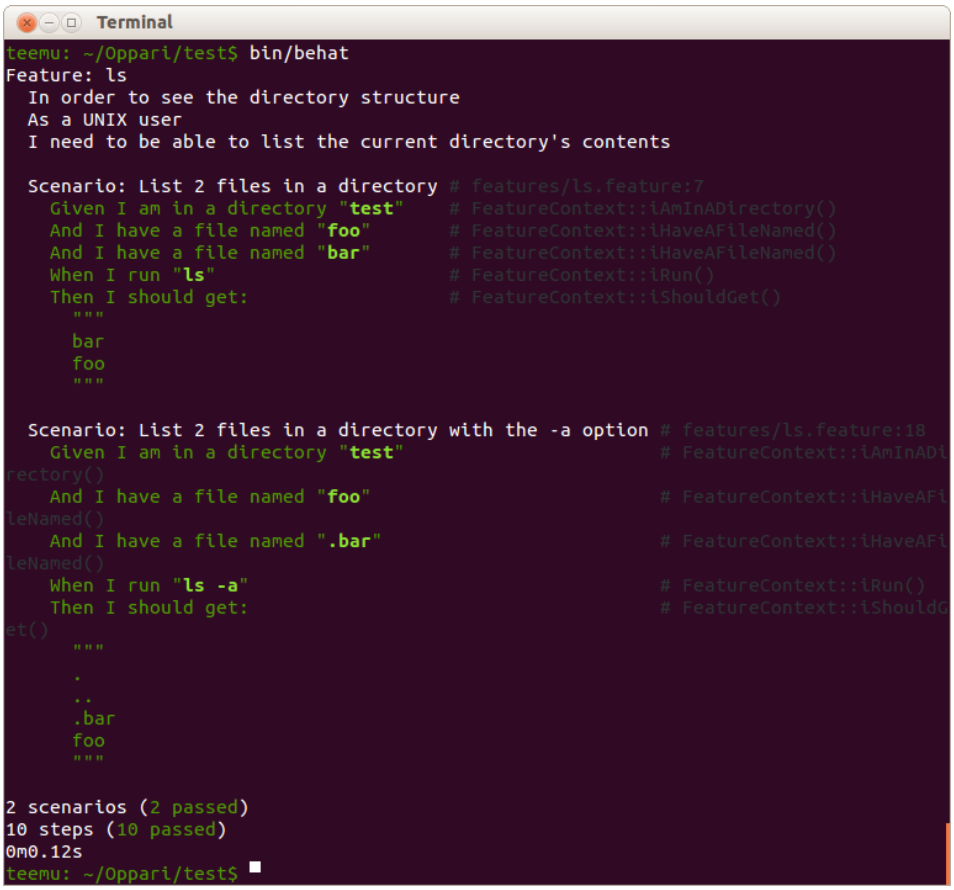
Lopuksi kokeiltiin vielä uuden skenaarion lisäämistä tarinaan. Uudessa skenaariossa listataan hakemiston sisältö käyttäen valitsinta -a. Komento ls -a listaa kaikki tiedostot, myös piilotiedostot. Tämän jälkeen tiedostossa features/ls.feature oli kaksi skenaariota (kuva 6).

```
Scenario: List 2 files in a directory with the -a option
  Given I am in a directory "test"
  And I have a file named "foo"
  And I have a file named ".bar"
  When I run "ls -a"
  Then I should get:
    """
    .
    ..
    .bar
    foo
    """
```



Kuva 6. Selkokielinen testitapaus kahdella skenaariolla.

Behat ajettiin seuraavaksi uudelleen. Tulos oli jälleen onnistunut, sillä Behat kävi kaksi eri skenaariota läpi suorittaen kaksi eri testiä (kuva 7).



Kuva 7. Onnistunut kahden skenaarion testi.

Behatin pikaoppaan avulla saatiin onnistuneesti kokeiltua BDD-mallin mukaista testausta. Lisäksi Behat tuli työkaluna tutummaksi, jonka seurauksena ymmärrettiin paremmin miten Behatia käytetään ohjelmistoprojektissa käytännön tasolla, sekä miten se asennetaan. Lopuksi olisi tietenkin pitänyt BDD-kehityssyklin mukaisesti refaktoroida koodi. Se ei ollut kuitenkaan pääasia opinnäytetyössä, joten se jätettiin tekemättä.

7 TYÖN ARVIOINTIA

Opinnäytetyö lähti liikkeelle harjoittelupaikasta saadun idean pohjalta. Käyttäytymislähtöinen kehitys olisi kuulemma hyvä hallita tulevaisuutta silmällä pitäen, koska se on saavuttamassa suurempaa tietoisuutta koko ajan.

Siksi ensin lähdettiin selvittämään mitä on testaus ja automaatiotestaus. Näihin kirjallista tietoa löytyi suhteellisen helposti, ja näin saatiin koostettua asiapitoiset luvut. Seuraavaksi siirryttiin vastaamaan ensimmäiseen tutkimuskysymykseen, mitä tarkoittaa testivetoinen kehitys (Test Driven Development). Tämä tuotti jo enemmän haasteita, mutta lopulta tarvittavat tiedot löytyivät kirjallisesta materiaalista ja aiheesta saatiin muodostettua kuva. Tämän jälkeen saatiin rakennettua riittävän kokoinen luku TDD:stä.

Ongelmat alkoivat siirtyessä eteenpäin opinnäytetyössä. Kirjallisen tiedon löytäminen käyttäytymislähtöisestä kehittämisestä oli haastavaa. Kirjallista materiaalia ei ollut opinnäytettä aloitettaessa juuri ollenkaan tai se oli hyvin hajanaista. Kyseessä on kuitenkin niin uusi asia, että kirjalliseen muotoon sitä ei ole vielä keritty painaa juuri ollenkaan. Tämän vuoksi käytettiin myös elektronisia lähteitä, kuten BDD:n kehittäjän Dan Northin nettisivuja. Kahden vuoden satunnaisen työskentelyn jälkeen ilmestyi ensimmäinen kunnon kirja aiheesta. Tässä vaiheessa opinnäytetyö oli saatu koostettua jo lähes valmiiksi, joten kirjalle ei ollut enää juuri tarvetta. Lopulta löydettiin riittävästi tietoa mitä käyttäytymislähtöinen kehitys (Behavior Drive Development) on. Lopulliset kaksi tutkimuskysymystä saivat myös vastauksensa tämän jälkeen. Miten BDD eroaa tai täydentää TDD:tä ja miksi niitä käytetään?

Vasta näiden vaiheiden jälkeen päästiin siirtymään varsinaiseen käytännön osuuteen, jossa piti selvittää miten toteutetaan pieni sovellus käyttäen BDD:tä. Tässä vaiheessa joulukuu lähestyi jo hyvää vauhtia, joten jouduttiin valitsemaan suhteellisen nopea ja yksinkertainen toteutus tapa käytännönosuuteen. Lopulta toteutettiin Behatin pikaohjeen mukainen harjoite, jossa selvisi miten BDD-mallin mukaista kehitystä tehdään käytännössä. Tämä osuus sujui oletettua helpommin. Asiaan auttoi varmasti teoriaosuuden läpikäyminen ensin, ja sen sulattelu viimeiset kaksi vuotta. Toisaalta käytännönosuuden läpikäyminen auttoi myös teoriaosuuden ymmärtämisessä.

Opinnäytetyöprosessin aikana opittiin mitä on automaatiotestaus ja mitä on TDD- ja BDD-mallin mukainen kehittäminen. Lisäksi opittiin ymmärtämään, että miksi BDD-mallin kohdalla ei voida puhua vain evoluutiosta TDD-mallista. Tämä oli se suurin yksittäinen oivallus, joka koettiin opinnäytetyön aikana.

Ennen työn aloittamista en tiennyt aiheesta juuri mitään, mutta nyt työn lopulla tietomäärä ja ymmärrys ovat kasvaneet. Yllättävää oli kuinka erilainen lähtökohta käyttäytymislähtöisessä kehittämisessä on verrattuna testivetoiseen kehittämiseen. BDD-mallissa pyritään palvelemaan kehittäjien lisäksi muitakin ohjelmistoprojektin jäseniä. Siksi on todennäköisenä että saatu

oppi ei ole turhaa, vaikka ei suuntautuisikaan it-alalla ohjelmistokehittämiseen. Käyttäytymislähtöisen kehittämisen malliin kun voi törmätä muissakin tehtävissä. Opinnäytetyö antoi siis paljon uutta ja tärkeää tietoa tulevaisuutta silmällä pitäen.

Käyttäytymislähtöinen kehitys tulee varmasti myöhemmin tutuksi yhdelle jos toiselle työelämässä, koska menetelmässä pyritään parempaan kommunikointiin eri sidosryhmien välillä. Osassa sidosryhmistä voi olla jäseniä joilla ei ole it-alasta juuri mitään kokemusta. Tähän kuitenkin auttaa onneksi BDD-mallin selkokielisyys testitapauksissa ja sitä myöten dokumentoinnissa. BDD-malli tulee varmasti saavuttamaan suurempaa tietoisuutta ajan kanssa, koska se ei ainoastaan palvele vain kehittäjiä, vaan myös muita kehitykseen osallistuvia tai siitä kiinnostuneita.

8 YHTEENVETO

Työn tarkoituksena oli tutkia käyttäytymislähtöistä kehittämistä ja vertailla sitä testivetoiseen kehittämiseen. Kummastakaan kehittämisenmallista minulla ei ollut aikaisempaa kokemusta, joten aihe oli aivan uusi. Työn teoria osuutta alustettiin ensin selventämällä testausta ja automaatiotestausta. Näiden jälkeen oli luontevaa siirtyä tutkimaan testivetoista kehitystä. Lopuksi siirryttiin tutkimaan käyttäytymislähtöistä kehittämistä, joka on työn varsinainen pääaihe.

Käyttäytymislähtöistä kehittämistä tutkittaessa huomattiin, että se pyrkii korjaamaan testivetoisen kehittämismallin puutteita. Testivetoisessa kehittämisessä törmättiin toistuvasti samoihin ongelmiin. Kehittäjät halusivat nimittäin tietää mistä pitäisi aloittaa, mitä pitäisi testata ja mitä ei, kuinka paljon yhden testin pitäisi testata, miten nimetä testit, sekä miksi testi ei mene läpi. Näihin ongelmiin BDD-mallin mukainen kehittäminen vastaa.

Teorian jälkeen käytännönosuudessa kokeiltiin BDD-mallin mukaista testausta Behat-kehyksellä. Se on PHP-kielelle tarkoitettu BDD-mallin mukainen automaatiotestaustyökalu. Behatin pikaoppaan avustuksella saatiin onnistuneesti käytännöntasolla kokeiltua mitä BDD-mallin mukainen testaaminen on. Käytännönosuus auttoi hahmottamaan vielä entistä paremmin asian.

Opinnäytetyön tavoitteisiin päästiin ja opittiin mitä on käyttäytymislähtöinen kehittäminen, sekä miten se voi palvella ohjelmistokehitysprosessissa eri sidosryhmiä. Lisäksi ymmärrettiin mitä hyötyjä ja haasteita BDD-mallista voi tulla, sekä miten se eroaa testivetoisesta kehittämisestä.

Suurin yksittäinen oppi oli kuitenkin käyttäytymislähtöisen kehittämisen luonteen tajuaminen. Kyseessä ei ole testivetoisen kehittämisen päivitetty malli, vaan erilainen ja isompi asia. BDD on kommunikointiväline eri teknisen osaamistason omaavien sidosryhmien välille, jotta voidaan luoda yhteinen näkemys siitä mitä pitäisi tehdä.

Tämän vuoksi käyttäytymislähtöinen kehitys tulee varmasti myöhemmin tutuksi yhdelle jos toiselle työelämässä, koska menetelmässä pyritään parempaan kommunikointiin eri sidosryhmien välillä. Osassa sidosryhmistä voi olla jäseniä joilla ei ole it-alasta juuri mitään kokemusta. Tähän kuitenkin auttaa onneksi BDD-mallin selkokielisyys testitapauksissa ja sitä myöten dokumentoinnissa. BDD-malli tulee varmasti saavuttamaan suurempaa tietoisuutta ajan kanssa, koska se ei ainoastaan palvele vain kehittäjiä, vaan myös muita kehitykseen osallistuvia tai siitä kiinnostuneita.

LÄHTEET

Behat Documentation. n.d. Organisaatio. Behat. Viitattu 26.3.2014. http://docs.behat.org/

Quick Intro to Behat. n.d. Organisaatio. Behat. Viitattu 29.11.2015. http://docs.behat.org/quick_intro.html

Chelimsky, D., Astels, D., Dennis, Z., Hellesøy, A., Helmkamp, B. & North, D. 2010. The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends. Raleigh & Dallas: Pragmatic Bookshelf.

Codeception: Modern PHP testing for everyone. n. d. Organisaatio. Codeception Team. Viitattu 27.11.2015, http://codeception.com

Collin, N. 2010. TDD ja ATDD. Ohjelmistojen testaus -opintojakson luentomateriaali. Tampereen teknillinen yliopisto. Viitattu 12.3.2014. http://www.cs.tut.fi/~testaus/s2010/luennot/Collin.pdf

Karjalainen, V. 2013. Testivetoisen kehityksen menetelmät ja työkalut testauksen eri tasoilla. Spring- ja Ruby on Rails -web-sovelluskehityksessä. Helsingin yliopisto. Matemaattis-luonnontieteellinen tiedekunta. Tietojenkäsittelytieteen laitos. Tietojenkäsittelytiede. Pro gradu -tutkielma.

Karkinen, S. 2013. Käyttäytymislähtöinen ohjelmistokehitys (BDD). Metropolia Ammattikorkeakoulu. Tietotekniikan koulutusohjelma. Opinnäytetyö.

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

Martin, R. 2009. Clean code. A Handbook of agile software craftsmanship. New Jersey: Prentice hall.

North, D. n.d. What's in a story. Dan North & Associates. Viitattu 26.3.2014. http://dannorth.net/whats-in-a-story/

North, D. n.d. Introducing BDD. Dan North & Associates. Viitattu 28.9.2015. http://dannorth.net/introducing-bdd/

North, D. 2013. Introducing BDD. Dan North & Associates. Viitattu 21.11.2015. http://dannorth.net/2012/05/31/bdd-is-like-tdd-if/

Smart, J. F. 2015. BDD in Action. Behavior-Driven Development for the whole software lifecycle. New York: Manning.

Ye, W. 2013. Instant Cucumber BDD How-to. Birmingham: Packt Publishing Ltd.