Arttu Nieminen

Käyttäytymislähtöisen kehityksen työkalut

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

25.4.2014

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Arttu Nieminen Käyttäytymislähtöisen kehityksen työkalut 66 sivua + 1 liitettä 25.4.2014
Tutkinto	insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Yliopettaja Erja Nikunen Yliopettaja Auvo Häkkinen

Tässä opinnäytetyössä käsitellään käyttäytymislähtöistä ohjelmistokehitystä ja sen avuksi tarkoitettuja työkaluja. Raportin alkupuolella selvitään, mitä ovat testivetoiset ohjelmistokehitysmenetelmät, sekä käydään läpi muutama testivetoinen kehitysmenetelmä. Kehitysmenetelmistä tutustutaan testivetoiseen ohjelmistokehitykseen, hyväksymistestivetoiseen ohjelmistokehitykseen sekä käyttäytymislähtöiseen ohjelmistokehitykseen.

Lisäksi raportissa käydään läpi muutama yleisin käyttäytymislähtöisen kehityksen työkalu. Työkaluista käydään läpi niiden perustoimintaperiaatteet. Työssä läpi käytävät työkalut ovat Cucumber, JBehave, Behat ja RSpec. Työkaluista painopiste on Cucumberissa.

Työn viimeisessä osassa käydään läpi, miltä käyttäytymislähtöinen kehitys voi näyttää käytännössä. Tässä osiossa kehitetään yksinkertainen selainpohjainen projektinhallintatyökalu käyttäen käyttäytymislähtöisen kehityksen periaatteita. Kehityksessä käytetään apuna Cucumber-työkalua.

Käyttäytymislähtöinen kehitys toi tuotteen kehittämiseen järjestelmällisyyttä. Se auttoi päättämään, mitä ominaisuuksia tuotteeseen tulisi toteuttaa missäkin vaiheessa sekä antoi yksittäisen ominaisuuden toteuttamiselle selkeän työskentelykaavan. Käyttäytymislähtöinen kehitys tuntui soveltuvan erinomaisesti web-sovelluksien tuottamiseen.

Avainsanat	Käyttäytymislähtöinen kehitys, Testivetoinen kehitys, Käyttäytymislähtöisen kehityksen työkalut

Author(s) Title Number of Pages Date	Arttu Nieminen Tools for Behavior-driven Development 66 pages + 1 appendices 25 April 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Erja Nikunen, Principal Lecturer Auvo Häkkinen, Principal Lecturer

This study discusses behavior-driven development and its tools. The first parts of the report explain different test-driven development methods. Development methods that were chosen for this study are test-driven development, acceptance-test-driven development and behavior-driven development.

The study also presents the most popular tools for behavior-driven development, Cucumber, JBehave, Behat and RSpec. The main focus is on Cucumber.

The last part of the study goes through behavior-driven development process in practice. This part explains how a simple project management tool can be developed by using tools and principles of behavior-driven development. The main tool used in this project is Cucumber.

Behavior-driven development brought systematic nature into the development process. It helped in deciding which features to develop and in which order. Behavior-driven development gave a clear workflow for the development of individual features. Behavior-driven development suited well for the development of web-applications.

Keywords	Behaviour-driven development, Test-driven Development, Tools for Behaviour-driven development

Sisällys

Lyhenteet

ATDD	Acceptance test driven development. Hyväksymistestivetoinen kehitys.
BDD	Behavior-driven development. Käyttäytymislähtöinen kehitys.
MVC	Model-View-Controller. Malli-Näkymä-Kontrolleri. Suunnittelumalli, joka koostuu kolmesta erilaisesta komponentista, mallista, näkymästä ja kontrollerista.
TDD	Test-driven development. Testivetoinen kehitys.
XP	Extreme programming. Ketterän ohjelmistokehityksen metodologia.

1 Johdanto

Ketterien menetelmien yleistyessä on noussut esille useita testivetoisen ohjelmistokehityksen menetelmiä. Tunnetuimpia näistä ovat testivetoinen kehitys (TDD, Test-driven development) ja hyväksymistestivetoinen kehitys (ATDD, acceptance-test-driven development). Kolmas, mutta ideologialtaan hieman eroava menetelmä on käyttäytymislähtöinen ohjelmistokehitys (BDD, Behavior-driven development). Nämä kolme menetelmää eivät ole niinkään testausmenetelmiä vaan ohjelmistokehitysmenetelmiä. Erityisesti käyttäytymislähtöinen ohjelmistokehitys painottaa sitä, että se ei ole testausmenetelmä, vaan se on tapa ajatella ohjelmistokehitystä.

Tässä työssä tutustutaan käyttäytymislähtöiseen ohjelmistokehitykseen ja, tutkitaan, kuinka se eroaa testivetoisesta ohjelmistokehityksestä ja miten käyttäytymislähtöistä kehitystä voi käyttää hyväksi erilaisissa projekteissa erilaisia työkaluja apuna käyttäen. Perehdytään käyttäytymislähtöisen kehityksen sanastoon ja periaatteisiin sekä tutustutaan erilaisiin käyttäytymislähtöisen kehityksen työkaluihin. Työkaluista tutustutaan Cucumberiin, JBehaveen, Behatiin ja RSpeciin.

Työn loppuosassa kehitetään käyttäytymislähtöisesti yksinkertainen projektinhallintatyökalu. Projektinhallintatyökalun kehittämisessä käytetään apuna Cucumber-työkalua.

Työn tavoitteena on oppia, mitä käyttäytymislähtöisellä kehityksellä tarkoitetaan, ja ymmärtää, kuinka sitä varten kehitettyjä työkaluja voi käyttää apuna käytännön sovelluskehityksessä.

2 Ketterä ohjelmistokehitys

Ketterillä ohjelmistokehitysmenetelmillä tarkoitetaan yleensä sellaisia menetelmiä, jotka perustuvat iteratiiviseen ja inkrementaaliseen kehitykseen. Näissä menetelmissä kehitysprosessi jaetaan useisiin muutamia viikkoja kestäviin iteraatioihin, joissa toteutetaan etukäteen valitut ominaisuudet. Ketterässä ohjelmistokehityksessä pyritään olemaan asiakkaan kanssa tiiviissä yhteistyössä. Yksi iteraatio sisältää ”minikoossa” vesiputousmallin mukaiset tehtävät: määrittelyn, suunnittelun, koodauksen ja testauksen.

Yksi suurimmista ketterän kehityksen eduista on nopea virheisiin reagoiminen. Koska ollaan asiakkaan kanssa tiiviisti yhteydessä, asiakkaan vaatimuksiin pystytään reagoimaan nopeasti. Mitä pidemmällä ohjelmistokehitysprosessia muutostarve tulee esille, sitä kalliimpaa sen toteuttaminen on. [1.]

Vuonna 2001 merkittävimmät ketterän ohjelmistokehityksen pioneerit kokoontuivat Utahiin keskustelemaan ketteristä menetelmistä ja niiden yhteisestä perustasta. Tämän tapaamisen tuloksena oli ketterän ohjelmistokehityksen julistus (engl. The agile manifesto). Julistus kuuluu seuraavasti. [2.]

> Löydämme parempia tapoja tehdä ohjelmistokehitystä, kun teemme sitä itse ja autamme muita siinä. Kokemuksemme perusteella arvostamme:
>
> Yksilöitä ja kanssakäymistä enemmän kuin menetelmiä ja työkaluja
>
> Toimivaa ohjelmistoa enemmän kuin kattavaa dokumentaatiota
>
> Asiakasyhteistyötä enemmän kuin sopimusneuvotteluja
>
> Vastaamista muutokseen enemmän kuin pitäytymistä suunnitelmassa
>
> Jälkimmäisilläkin asioilla on arvoa, mutta
>
> arvostamme ensiksi mainittuja enemmän.

Suosituimpia ketteriä kehitysmenetelmiä ovat Scrum ja Extreme Programming (lyh. XP). Näitä käsitellään seuraavaksi.

2.1 Scrum

Scrum on projektinhallintamenetelmä, jota käytetään yleisesti ketterässä ohjelmistokehityksessä. Scrumissa monitaitoinen ryhmä suorittaa kehitysprosessin alusta loppuun iteratiivisesti. Scrum-kehitysprosessi jakaantuu iteraatioihin, joita kutsutaan sprinteiksi. Yksi sprintti kestää 1-4 viikkoa, jonka aikana tuotetaan sprinttiin valitut ominaisuudet valmiiksi. Sprintin alussa valitaan ominaisuudet, jotka sprintin aikana toteutetaan. Sprintin lopussa tehdään katselmus, jossa sprintin tulos esitetään asiakkaalle. Ennen

seuraavaa sprinttiä tarkastellaan kuluneen sprintin prosessia, mietitään, mikä meni hyvin ja missä voisi parantaa. Sprintin aikana pidetään joka päivä lyhyt kokous, Daily Scrum, jossa jokainen kehitysryhmän jäsen kertoo, mitä teki eilen, miten se hänen mielestään meni ja mitä aikoo tehdä tänään. [3.]

2.2 Extreme Programming

Extreme Programming ei ole niin tarkoin määritelty menetelmä kuin Scrum vaan enemmänkin kokoelma hyviä ohjelmistokehityskäytäntöjä. Näitä XP:n käytäntöjä ovat esimerkiksi pariohjelmointi, jossa kaksi henkilöä työskentelee saman näytön ääressä, sekä koodin yhteisomistajuus, jolla tarkoitetaan sitä, että kaikki ovat vastuussa koodista yhdessä. Koodista tehdään sellaista, että kaikki ymmärtävät, mitä se tekee, ja ohjelmakoodia käydään yhdessä läpi. Näin estetään se, että joku tietty koodi olisi vain yhden henkilön ylläpidettävissä (lomatilanteet, työpaikan vaihto jne. aiheuttaisivat ongelmia). [4.]

Yksi tärkeimmistä käytännöistä, joita XP:ssä seurataan, on testivetoinen kehitys. Tämänkaltainen ”testaa ensin” -ajattelutapa on tärkeä osa XP:tä.

3 Testivetoiset kehitysmenetelmät

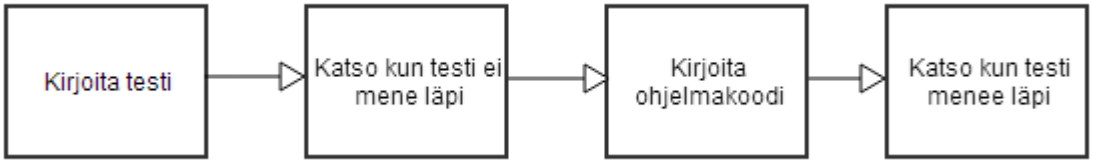
Tässä luvussa kerrotaan testivetoisista kehitysmenetelmistä ja niihin liittyvistä käytännöistä. Ensimmäisessä aliluvussa käydään läpi testivetoista ohjelmistokehitystä. Sen jälkeen tutustutaan esimerkkien avulla määrittelemisen käsitteeseen, johon perustuvat sekä hyväksymistestivetoinen että käyttäytymislähtöinen kehitys.

3.1 Testivetoinen kehitys

Ensimmäistä kertaa testivetoinen kehitys esiteltiin erillään XP:stä vuonna 2003 Kent Beckin kirjassa ”Test-Driven Development by Example”. [23.]

Testivetoisen kehityksen ideana on, että tehdään ensiksi testit ja tämän jälkeen kirjoitetaan toiminnallisuus. Testivetoisen kehityksen tuloksena on useita automatisoituja yksikkötestejä (ja mahdollisesti toiminnallisia testejä) ja ohjelmakoodia, jota on helppo

testata. Jos testit kirjoitettaisiin vasta, kun ohjelmakoodi on jo kirjoitettu, voisi testin kirjoittaminen olla vaikeaa johtuen ohjelmakoodin toteutuksesta. Jos testit kirjoitetaan ensin, on ohjelmakoodista tehtävä helposti testattavaa.



Kuva 1. Testivetoisen kehityksen perusperiaate

Testien kirjoittaminen ennen ohjelmakoodia auttaa ehkäisemään virheiden syntymistä heti alkuunsa. Mitä pidemmälle ohjelma on edennyt, sitä kalliimmaksi virheiden korjaaminen tulee. TDD:n avulla suuri osa virheistä huomataan heti ja korjataan samantien. TDD:n toteuttaminen auttaa monesti tuottamaan parempaa, modulaarista, koodia. [5.]

Otetaan esimerkki, jossa asiakas tilaa imurin. Asiakas ei sen tarkemmin määrittele, miten imurin tulisi toimia, vaan hän olettaa, että imurin kehittäjä kyllä tietää, miten imuri toimii. Kehittäjällä itsellään onkin jonkinlainen kuva siitä, millainen imuri on. Kehittäjä päättää lähestyä imurin kehitystä testivetoisesti. Kehittäjä arvelee, että kun käynnistysnappia painetaan, tulisi imurissa kulkea virtaa. Hän kirjoittaa testin, joka mittaa virrankulkua imurissa napin painalluksen jälkeen. Kun testi on kirjoitettu, painaa kehittäjä käynnistysnappia ja toteaa, että mitään ei tapahdu. Testi epäonnistuu. Kehittäjä ratkaisee ongelman rakentamalla imuriin toiminnallisuuden, että se käynnistyy, kun nappia painetaan. Tämän jälkeen testi menee läpi. Imurin kehittäjä toistaa vaiheita testin toteutus, epäonnistunut testi, toiminnallisuuden toteutus ja läpi menevä testi, kunnes imuri on valmis.

Esimerkki on tarkoituksellisesti kärjistetyn yksinkertainen. Todellisuudessa asiakkaalla voisi olla tarkat määritykset siitä, miten he haluavat imurin toimivan. Asiakas voisi esimerkiksi antaa kehittäjälle määrittelydokumentin, jossa kerrotaan, minkälainen imurista halutaan.

3.2 Määrittely esimerkkien avulla

Esimerkkien avulla määrittely (engl. Specification by Example) on tapa määritellä tuotteen vaatimukset käytännön esimerkkien avulla. Ohjelmistoalalla tämä tarkoittaa sitä, että määritellään asiakkaan kanssa sovelluksen toiminta tarkkojen yksityiskohtaisten esimerkkien avulla. Tarkoituksena on pienentää kuilua asiakkaan, joka ei ymmärrä teknistä toteutusta, ja kehittäjän, joka ei osaa lukea asiakkaan ajatuksia, välisessä kommunikaatiossa. Kun ohjelman toiminta on määritetty tarkkojen esimerkkien avulla, voidaan niiden pohjalta tehdä hyväksymistestit, jotka läpäistyään sovellus on valmis. Nämä automatisoidut testit toimivat myös ”elävänä” dokumentaationa. Pyritään tekemään tuote, joka vastaa mahdollisimman hyvin asiakkaan tarpeita. Sekä hyväksymistestivetoinen että käyttäytymislähtöinen kehitys perustuvat ajatukseen esimerkkien avulla sovelluksen määrittelystä. [6.]

Palataan imuriesimerkkiin. Asiakas, joka ei tiedä, miten imuri teknisesti toimii, tulee tilaamaan imurinkehittäjältä uutta imuria. Imurinkehittäjä pyytääkin asiakasta kertomaan esimerkkejä siitä, miten imurin tulisi toimia. Asiakas antaa esimerkkinä, että jos lattialla on kilo hiekkaa, imurin suuttimen osoittaa kohti hiekkakasaa ja painaa nappia, jossa lukee ”Start”, niin viiden sekunnin kuluttua lattialla ei tulisi olla enää yhtään hiekkaa. Tällä tavalla esimerkkien avulla imurin määritteleminen auttaa kehittäjää ymmärtämään, mitä asiakas haluaa.

3.3 Hyväksymistestivetoinen kehitys

Hyväksymistestivetoinen kehitys on tapa tehdä sovellukselle määrittely, jota vasten voidaan todeta sovelluksen toiminnallisuus automaattisesti. Toiminnallisuuden testaava, automaattisesti suoritettava testi tehdään ennen itse toiminnallisuuden kirjoittamista. Toiminnallisuus on valmis, kun testi menee läpi. Asiakas on tiiviisti mukana hyväksymistestien suunnittelussa. Näin saadaan varmistettua, että ohjelma tekee oikeita asioita oikein.

Hyväksymistestit ovat siis toiminnallisia testejä. Testataan, että ohjelma toimii, kuten asiakas on määritellyt. Testien toteuttamiseen löytyy useita eri työkaluja, kuten esimerkiksi Fit ja Selenium. [24; 25.]

Käytännössä asiakkaan suostutteleminen hyväksymistestivetoiseen kehitykseen voi olla vaikeaa. Asiakkaalla täytyy olla jo ennestään jonkun verran teknistä osaamista. Asiakas ei monesti suostu täyspainoiseen hyväksymistestivetoiseen kehitykseen, koska se vaatii asiakkaalta paljon aikaa ja vaivaa. Tapauksessa, jossa asiakas ei suostu ATDD:hen, tehdään määrittely asiakasrajapinnassa kuten normaalisti. Määrittelyn pohjalta tehdään hyväksymistestit. Huonona puolena, että asiakkaan varmistus testin oikeellisuudesta jää puuttumaan. [7.]

Imuriesimerkissä asiakas otettaisiin mukaan suunnittelemaan erilaisia testitapauksia, jotka läpäistyään imuri on valmis. Tämä vaatisi käytännössä kuitenkin asiakkaalta ainakin jonkin verran teknistä osaamista. Jos asiakas ei tiedä imureiden toiminnasta mitään eikä täten kykene osallistumaan hyväksymistestien suunnitteluun, voitaisiin määrittely asiakasrajapinnassa tehdä kuten normaalisti. Tällöin kehittäjä itse suunnittelee määrittelyn pohjalta hyväksymistestit.

3.4 BDD (Behavior-driven development) eli käyttäytymislähtöinen kehitys

Testivetoinen kehitys on ideana hyvä, mutta sen toteutuksessa on usein joitain selviä heikkouksia. Monesti on sanottu, että BDD onkin vain ”TDD:tä oikein tehtynä”, mutta BDD-asiantuntijat pitävät tätä sanontaa epätarkkana ja harhaanjohtavana [8]. Suurimmat erot TDD:n ja BDD:n välillä ovatkin termistössä ja ajattelutavassa. Monilla on vaikeuksia sisäistää testivetoista ohjelmistokehitystä. Yksi suuri kysymys, mikä yleensä herää testivetoiseen kehitykseen tutustuvalla on ”Mitä pitäisi testata?”. Mitä pitäisi testata, jos ei ole vielä ollenkaan toiminnallisuuksia eikä koodia mitä testata. Käyttäytymislähtöisessä ohjelmistokehityksessä ei niinkään puhuta testin toteuttamisesta vaan ominaisuuksien määrittelemisestä. Kun tarkemmin ajatellaan, niin siitähän TDD:ssäkin on testin kirjoittamisvaiheessa kyse: määritellään kuinka testattavan ominaisuuden tulisi toimia. [10; 16.]

Testivetoinen kehitys on yleensä hyvin kehittäjäpohjaista. Koodin kirjoittaja kirjoittaa myös siihen liittyvät testit (joskus myös eri henkilö kirjoittaa testit ja toinen toteuttaa toiminnallisuudet). BDD:ssä ominaisuudet kirjoitetaan yleensä tavallisena luettavana tekstinä, niin että myös muut kuin kehittäjät voivat olla mukana suunnittelemassa toiminnallisuutta. Kun ominaisuudet määritellään kaikkien ymmärrettävällä kielellä arkikielellä (engl .ubiquitous language), pääsevät asiakkaatkin lähemmin mukaan ohjelmiston

kehitykseen. Tämä auttaa siinä, että kehitetään oikeanlaista sovellusta ja vältetään suurimmat väärinkäsitykset siitä, mitä ohjelman pitäisi tehdä. [9; 10, s. 125-148]

Ominaisuuksien määrittelyt muistuttavat käyttäjätarinoita (engl. user story). Eri työkalut kutsuvat näitä käyttötarinoita eri nimillä; Cucumberissa ja muissa Gherkiniä, Cucumberin lanseeraamaa ominaisuuksien määrittelyyn tarkoitettua kieltä, käyttävissä näitä käyttötarinoita kutsutaan ominaisuuksiksi (engl. feature). JBehavessa niitä kutsutaan tarinoiksi (engl. story). Ominaisuudet (tai tarinat) koostuvat skenaarioista, jotka ovat yksityiskohtaisia esimerkkejä siitä, miten ominaisuuden tulisi toimia. [11; 12.]

Otetaan jälleen esimerkiksi imuri, mutta tällä kertaa se kehitetään käyttäytymislähtöisesti. Käyttäytymislähtöinen kehittäminen yhdistää esimerkkien avulla määrittelyn ja hyväksymistestivetoisen kehityksen ominaisuuksia. Asiakas voi esimerkkien avulla määritellä, miten hän haluaa imurinsa toimivan. Näiden esimerkkien pohjalta kehittäjä voi määritellä testit, jotka testaavat, toimiiko imuri juuri niin kuin esimerkissä on määritelty. Sovelluskehityksessä kuilu esimerkin ja testin välissä voidaan käytännössä poistaa erilaisten käyttäytymislähtöisen kehityksen työkalujen, esimerkiksi Cucumberin, avulla. Cucumberissa näitä esimerkkejä kutsutaan ominaisuuskuvauksiksi (koska niissä määritellään ominaisuuksia, jotka tuotteessa on), ja ne koostuvat askelista, jotka toimivat kuten testit. Tällöin saadaan asiakas mukaan hyväksymistestien suunnittelemiseen.

3.5 Hyviä BDD-käytäntöjä

BDD:n avulla on tarkoituksena toteuttaa tärkein toiminnallisuus ensin. Uutta ominaisuutta määriteltäessä tuleekin miettiä, miksi sitä tarvitaan. Yksi hyvä tapa on viiden peräkkäisen miksi-kysymyksen esittäminen (engl. 5 whys). [13.]

Esimerkki: Laskintoiminnallisuus
Miksi? Jotta pystytään laskemaan yksinkertaisia laskuja.
Miksi? Ettei anneta liian paljon rahaa kassalla.
Miksi? Jotta yrityksen toiminta on voitollista.
Miksi? Että yrityksellä on varaa maksaa palkat.
Miksi? Ihmiset tarvitsee rahaa eläkkeen.

Toinen hyvä käytäntö ominaisuuksien määrittelyssä on niin sanottu ominaisuusinjektio (engl. feature injection), jossa kerrotaan, miksi ominaisuutta tarvitaan (ja kuka sitä tarvitsee). [14.]

Yksi käytetty tapa määritellä ominaisuuksia on niin kutsuttu Connextra-formaatti [9, s. 255]. Connextra-formaatti sopii hyvin ominaisuusinjektioon.

In Order To <business value is derived> As a <role> I want <some feature>

Nimi-ominaisuusinjektio tulee siitä, että saman arvo voidaan tuottaa useilla eri tavoilla, eli ominaisuus voidaan ”injektoida” tähän sapluunaan. Seuraavassa ominaisuusinjektioesimerkissä haluttu asia on bottien pois saaminen sivustolta.

In order to stop bots spamming my site As the forum moderator I want users to have to fill in a captcha to comment.

Boteista voidaan päästä eroon esimerkiksi kuvavarmennuksen (engl. CAPTCHA, Completely Automated Public Turing test to tell Computers and Humans Apart) avulla.

In order to stop bots spamming my site As the forum moderator I want users to have to answer a logic question.

Samaan lopputulokseen voidaan päästä myös toteuttamalla jokin muu ominaisuus, kuten esimerkiksi laittamalla käyttäjät vastaamaan logiikkakysymykseen.

Toinen tapa esittää ominaisuusinjektio on kolmiaskelinen

1. Etsi tuotettava arvo.
2. Injektoi ominaisuus.
3. Paikanna esimerkit.

Arvoa, eli miksi ominaisuutta tarvitaan, etsiessä voi käyttää apuna useita eri menetelmiä, mm. aikaisemmin mainittua viiden miksi-kysymyksen kysymistä.

3.6 BDD vs. ATDD

Molemmissa sekä käyttäytymislähtöisessä kehityksessä että hyväksymistestivetoisessa kehityksessä tähdätään samaan asiaan. Asiakkaan kanssa pyritään määrittelemään, mitä sovelluksen tulisi tehdä ja toteuttaa samalla automatisoidut testit, jotka läpäistyään ominaisuus on valmis.

Suurin ero on termistössä ja työkaluissa. Käyttäytymislähtöinen kehitys keskittyy, kuten nimestä voi päätellä, sovelluksen käyttäytymiseen, kun taas testivetoiset kehitysmenetelmät keskittyvät enemmän testien suunnittelemiseen. TDD:ssä ja ATDD:ssä puhutaan testeistä, BDD:ssä puhutaan ominaisuuksista. [15.]

BDD-työkalut käyttävät monesti selkokielisiä ajettavia määrittelyitä, jotka käyvät dokumentaatiosta ja jotka toimivat automatisoituina testeinä.

Kaikki nämä termit (BDD, TDD, ATDD) voivat tuntua hyvin sekavilta, mutta niissä on hyvin paljon samaa.

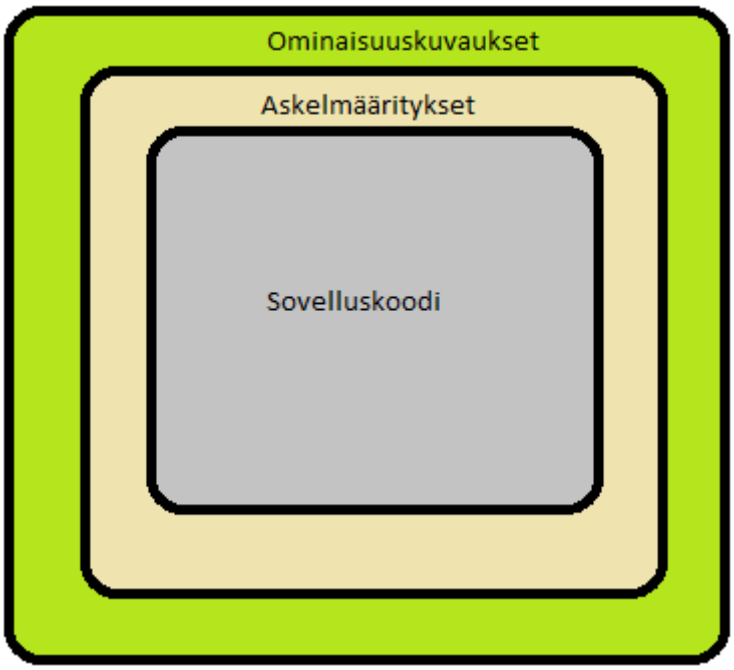
4 Käyttäytymislähtöisen kehityksen työkalut

Kun idea käyttäytymislähtöisestä kehityksestä on yleistynyt, on sen toteuttamisen avuksi tehty erilaisia työkaluja. Käyttäytymislähtöinen kehitys ei kuitenkaan välttämättä tarvitse mitään työkalua, eikä BDD-työkalun käyttöönotto automaattisesti tarkoita sitä, että toteuttaa BDD:tä. Työkaluista voi kuitenkin olla todella suuri apu BDD:n oikeaoppisessa toteuttamisessa.

Työkaluja löytyy lähes jokaiselle ohjelmointikielelle. Yhteistä näillä työkaluilla yleensä on, että ne ovat ”itsensädokumentoivia” ja monesti käyttävät selkokielisiä ominaisuuskuvauksia ja tilannekuvauksia. Työssä esitetään neljä laajalti käytettyä BDD-työkalua: Cucumber, JBehave, RSpec ja Behat. Nämä neljä valittiin, koska jokainen niistä soveltuu hieman erilaisiin projekteihin. Cucumber ja RSpec soveltuvat Ruby-projekteihin, JBehave on ensisijaisesti tarkoitettu Java-kehitykseen, ja Behat toimii PHP-projekteissa.

Perusperiaate näillä työkaluilla on yleensä, että kirjoitetaan kuvauksia sovelluksen eri toiminnoista. Kuvauksissa kerrotaan, miten ominaisuuden tulisi toimia erilaisissa tilanteissa. Ominaisuuskuvaukset toimivat myös ajettavina testeinä. Tarkoituksena on, että ominaisuuskuvauksia kirjoitetaan yhdessä asiakkaan kanssa, joten ne on yleensä toteutettu hyvin yksinkertaisesti ja ovat helppokäyttöisiä. Monissa BDD-työkaluissa on ominaisuuskuvaukset ja joissain niitä voi kirjoittaa monilla eri kielillä. Selkokieliset kuvaukset helpottavat asiakkaan kanssa ominaisuuksien määrittelemistä, erityisesti jos asiakas pääsee kirjoittamaan niitä omalla äidinkielellään. Nämä ajettavat ominaisuuskuvaukset ovat ”elävää” dokumentaatiota. Dokumentaatio pysyy koko ajan ajan tasalla, eikä käy niin, että dokumentaatio kirjoitetaan kerran ja se unohtuu. Ominaisuuskuvaukset toimivat myös keskitettynä paikkana, mistä näkee, mitä sovellus tekee. [17, s. 3-7]

Käyttäytymislähtöinen kehittäminen tapahtuu ns. ulkoa sisäänpäin (engl. outside-in), jolloin sovelluskehitystä lähestytään ensiksi asiakkaan näkökulmasta. Ensimmäisenä kirjoitetaan ominaisuuskuvaukset, jonka jälkeen kirjoitetaan askelmäärityksetja viimeisenä itse ohjelmakoodi.



Kuva 2. Ulkoa sisäänpäin -lähestymistapa käyttäytymislähtöiseen kehitykseen

Tässä luvussa tutustutaan neljään tunnettuun käyttäytymislähtöisen kehityksen työkaluun, Cucumberiin, Jbehaveen, Behatiin ja RSpeciin. Työkaluista käydään läpi niiden tärkeimmät ominaisuudet.

4.1 Cucumber

Cucumber on Rubylla toteutettu työkalu, joka suorittaa tekstitiedostoon määritellyt toiminnalliset kuvaukset automatisoituina testeinä.

4.1.1 Gherkin ja ominaisuuskuvaukset

Cucumberissa toiminnalliset määrittelyt eli ominaisuuskuvaukset kirjoitetaan Gherkin-kielellä. Gherkin-kieli on selväkielinen ja helposti luettava. Gherkinillä määritellään ominaisuuksia (engl. feature), jossa kerrotaan, miksi ominaisuus on olemassa ja miten sen tulee toimia erilaisissa tilanteissa (engl. scenario). Cucumberin tilannekuvaukset ajavat kahta asiaa: ne ovat itsensä dokumentoivia ja niitä voi ajaa automatisoituina testeinä. Ominaisuuskuvaukset sijaitsevat .feature-loppuisissa tiedostoissa. Yhdessä ”.feature”-tiedostossa on yksi ominaisuuskuvaus. [17, s. 29-30]

Ominaisuuskuvaukset koostuvat selkokielisestä tekstistä, jonka seassa on avainsanoja. Jokainen ominaisuuskuvaus alkaa Feature -avainsanalla, jota seuraa vapaa kuvaus ominaisuudesta. Kuvauksen jälkeen tulee tapaus, joka alkaa Scenario-avainsanalla. Scenario-avainsanan jälkeen kirjoitetaan tapauksen nimi, jonka jälkeen kirjoitetaan askeleet, joista tapaus koostuu. Ominaisuuskuvaus voi sisältää useita eri tapauksia siitä, miten ominaisuuden tulisi toimia. [17, s. 29-30]

Esimerkissä näkyy ominaisuuskuvaus uuden viestin lisäämiselle. Ominaisuuskuvauksessa kerrotaan, kuinka viestin lisäys toimii.

```
Feature: Creating a post
    In order to create a post
    A Visitor
    Should fill the post creation form

    Scenario: Creates a new post
        Given I am on the new post page
        Then I should see "New post"
        And I fill in "post name" with "John"
        And I fill in "post title" with "Title"
        And I fill in "post content" with "Contents"
        When I press "Create Post"

       Then I should see "Post was successfully created."
```

Koodiesimerkki 1. Cucumberin ominaisuuskuvaus

Cucumberin tilanteet rakentuvat askelista (engl. step). Askeleita ovat esimerkiksi Given, When ja Then. Cucumberin ominaisuuskuvauksia pystyy kirjoittamaan yli neljälläkymmenellä kielellä, myös suomeksi. [18.]

Given-askeleen on tarkoitus saattaa järjestelmä tunnettuun tilaan, ennen muiden askelien suorittamista. When-askel kuvaa skenaarion avaintoimintoa, eli mitä skenaariossa tehdään. Then-askel taas kertoo, missä tilassa järjestelmän tulee olla avaintoimintojen jälkeen [11]. Näiden kolmen askeleen lisäksi voidaan käyttää myös avainsanoja And ja But, joiden avulla voidaan esimerkiksi lisätä useampi When-askel. Background-avainsanalla voidaan suorittaa useita askelia, jotka ajetaan aina ennen jokaista tapausta. Ominaisuuskuvausten sekaan voi lisätä kommenttirivejä laittamalla rivin alkuun #-merkin. [17 s. 30-31, 33, 61-64]

Tapausaihion (engl. scenario outline) avulla voidaan suorittaa monta samankaltaista tapausta eri arvoilla. Tapausaihiossa määritelty tapaus ajetaan Examples-kohdassa määritellyillä arvoilla. Jokainen rivi ajetaan erillisenä tapauksena. [17, s. 70]

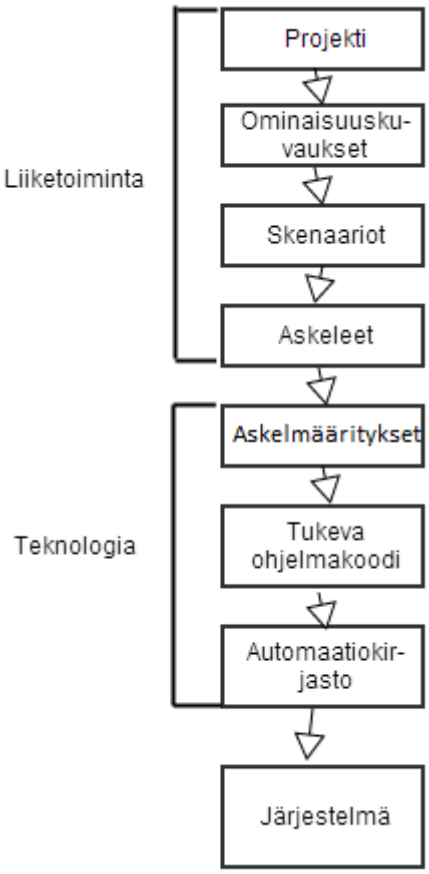
```
Scenario Outline: Sending a Message
        Given there are no messages
        Then I should see "Send a Message"
        And I fill in "post name" with <name>
        And I fill in "post message" with <message>
        When I press "Create Post"
        Then I should see <result>

        Examples:
          | name  | message           |  result                          |
          | Frank | Ensimmäinen viesti | Nimimerkki89: Ensimmäinen viesti |
          | Arttu | Toinen viesti      | Arttu: Toinen viesti             |
```

Koodiesimerkki 2. Tapausaihio

Cucumberilla kehittämisen eri vaiheet voidaan jakaa kahteen osaan: liiketoiminnalliseen osaan ja teknologiseen osaan. Liiketoiminnalliseen osaan kuuluvat projekti, ominaisuuskuvaukset, skenaariot ja askeleet.

Teknologiseen osaan kuuluvat esimerkiksi askelien käytännön toteuttaminen askelmääritysten avulla. Myös askelmääritysten toteuttamisessa avuksi toteutettu tukeva ohjelmakoodi ja erilaiset automaatiokirjastot kuuluvat teknologiseen osaan.

Ulkoa sisäänpäin -lähestymistapa toimii niin, että tuotteen kehitystä lähestytään ensin liiketoiminnallisesta näkökulmasta ja siirrytään kohta kerrallaan kohti teknologista osaa (kuva 3).



Kuva 3. Ulkoa sisäänpäin -lähestymistapa Cucumberissa. [18 s. 7-8]

Cucumberissa ulkoa sisäänpäin -kehitys näkyy niin, että ensiksi kirjoitetaan sovelluksen toiminnan määrittelyyn liittyvät ominaisuuskuvaukset, jonka jälkeen kirjoitetaan askelkuvaukset, jotka toimivat ns. testeinä.

4.1.2 Askelmääritykset

Kun ominaisuuskuvaus on kirjoitettu, toteutetaan askeleet kirjoittamalla askelmääritys (engl. step definition) Rubylla. Askelmäärityksissä ohjelmoidaan, mitä tehdään, kun suoritetaan tietty askel. Koodiesimerkissä 2 esitetyille kuudelle viimeiselle askeleelle askelmääritykset voitaisiin toteuttaa seuraavan koodin avulla.

```
Given /^I am on (.+)$/ do |page_name|
  visit path_to(page_name)
end

Then /^I should see "([^\"]*)"$/ do |text|
  page.should have_content(text)
end

When /^I fill in "([^\"]*)" with "([^\"]*)"$/ do |field, value|
  fill_in(field.gsub(' ', '_'), :with => value)
end

When /^I press "([^\"]*)"$/ do |button|
  click_button(button)
end
```

Koodiesimerkki 3. Cucumberin askelkuvaus

Näitä askelkuvauksia voi hyödyntää useissa eri ominaisuuskuvauksissa, eikä niitä tarvitse kirjoittaa jokaisen ominaisuuden kohdalla uudelleen. Kun askelia on kirjoitettu suuri määrä, on uusien ominaisuuksien kirjoittaminen helppoa, kun ei tarvitse kirjoittaa jokaista askelta uudelleen. Askelmäärityksiä tehdessä käytetään yleensä apuna erilaisia automaatiokirjastoja kuten Seleniumia tai Capybaraa. Askelien järkevä toteuttaminen onkin erittäin tärkeää, jotta ne olisivat mahdollisimman hyvin käytettävissä myöhemmin. Hyvä tapa toteuttaa uudelleenkäytettäviä askelkuvauksia on käyttää säännöllisiä lausekkeita. Askelmääritelmiä kirjoittaessa kannattaakin käyttää DRY-periaatetta (don’t repeat yourself). [17, s. 39-45]

4.1.3 Hakemistorakenne

Cucumberissa hakemistorakenne on muokattavissa, mutta hyvä tapa on kuitenkin käyttää oletushakemistorakennetta (taulukko 1). [17, s. 23-24]

Taulukko 1. Cucumberin oletushakemistorakenne

Hakemisto	
features	Sisältää .feature-tiedostot, joissa on ominaisuuskuvaukset.
features/step_definitions	Sisältää askelmääritykset.
features/support	Sisältää sovelluksen apukoodit, koukut ja env.rb:n

Sovelluksen juurihakemistosta löytyvät kaksi Cucumberille tärkeää hakemistoa: features ja support. Features-hakemisto sisältää ominaisuuskuvaukset. Cucumber etsii oletuksena askelmäärityksiä features-hakemiston alta löytyvästä step_definitions-hakemistosta.

Support-hakemistossa säilytetään Cucumberin käyttöä tukevaa koodia. Cucumber yrittää aina ensimmäisenä ajaa support-hakemistosta env.rb-tiedostoa. Koukut sijoitetaan myös support-hakemistoon. Support-hakemistoa ei välttämättä joissain yksinkertaisimmissa sovelluksissa tarvitse ollenkaan, mutta kun sovellukselle tulee kokoa ja Cucumberilla tarvitsee testata monimutkaisempia ominaisuuksia, on sen käyttäminen välttämätöntä. [17, s. 128-132]

4.1.4 Cucumberin ajaminen

Kun askeleet on toteutettu, ajetaan Cucumberilla testit. Kun huomataan, että testit eivät mene läpi, toteutetaan koodia sen verran, että testit menevät läpi (kuva 4).

```
  Scenario: Creates a new post with too short title                    # feature
s\post.feature:27
    Given I am on the new post page                                    # feature
s/step_definitions/navigation_steps.rb:1
    Then I should see "New post"                                       # feature
s/step_definitions/navigation_steps.rb:5
    And I fill in "post name" with "nimi"                              # feature
s/step_definitions/navigation_steps.rb:9
    And I fill in "post title" with "a"                                # feature
s/step_definitions/navigation_steps.rb:9
    And I fill in "post content" with "Sisältö"                        # feature
s/step_definitions/navigation_steps.rb:9
    When I press "Create Post"                                         # feature
s/step_definitions/navigation_steps.rb:13
    Then I should see "1 error prohibited this post from being saved:" # feature
s/step_definitions/navigation_steps.rb:5

4 scenarios (4 passed)
23 steps (23 passed)
0m2.070s
Coverage report Rcov style generated for Cucumber Features to Z:/innodev/example
-projects/rails-shell/project/coverage/rcov
```

Kuva 4. Cucumberin tulostus.

Kun Cucumber ajetaan komentorivillä, voidaan komentoriviargumenteilla määrittää esimerkiksi, mitkä testit ajetaan ja minkälaisessa formaatissa tuloste saadaan. Kuvan 4 esimerkissä on ajettu neljä eri skenaariota, jotka kaikki menevät läpi. Tämän lisäksi Cucumberia on käsketty antamaan tuloste myös rcov-tyyppisenä (rcov on testikattavuustyökalu Rubylle) raporttina. [17 s. 17-19, 194-196]

4.1.5 Cucumberin Tagit

Cucumber tarjoaa ominaisuuden, tagit, jonka avulla voidaan helposti organisoida ominaisuuksia ja tapauksia. Ominaisuudelle tai tapaukselle annetaan tagi laittamalla sitä edeltävälle riville @-merkillä varustettuna tagin nimi. [17, s. 80-82]

```
@fast
Feature: Sending a message
...
```

Koodiesimerkki 4. Cucumberin tagi-ominaisuus

Koska jotkut ominaisuudet ja tapaukset voivat olla hitaita suorittaa, niin voi olla hyvä tapa esimerkiksi merkitä niistä tageilla nopeasti ajettavat. Kun Cucumberia ajetaan komentoriviltä, voidaan ”--tags”-komentoriviargumentilla määritellä ne ominaisuudet tai tapaukset, jotka halutaan ajaa. Esimerkiksi, jos halutaan ajaa vain nopeasti suoritettavat tapaukset voidaan sanoa ”cucumber --tags @fast”, jos nopeasti ajettavat on merkitty ”@fast”-tagilla. Komentoriviltä Cucumberia ajettaessa pystyy antamaan useamman tagin, joita pystyy yhdistelemään ”AND” ja ”OR” -operaatioilla. Esimerkiksi, jos ajetaan ”cucumber --tags @fast, @ui”, ajaa cucumber kaikki ominaisuudet tai tapaukset, jotka on merkitty joko ”@fast” tai ”@ui” -tagilla. Jos taas ajetaan ”cucumber --tags @fast --tags @ui”, ajaa cucumber kaikki, jotka on merkitty sekä ”@fast” että ”@ui”-tageilla. Tageja voi käyttää myös negaatioina. Komento ”cucumber --tags ~@fast” ajaa kaikki ominaisuudet tai tapaukset, jotka eivät ole merkitty @fast-tagilla. [17, s. 193]

Cucumberissa on myös joitain erityisiä tageja, kuten @javascript. Kun Cucumber ajetaan @javascript -tagin kanssa, käyttää Cucumber Javascriptia ymmärtävää järjestelmää web-kyselyiden suorittamisessa. [11.]

4.1.6 Cucumberin Maailma

Cucumber ajaa tapaukset maailmassa (eng. World), joka on Object-luokan instanssi. Kaikki askelmäärityksetajetaan nykyisen maailman kontekstissa. Esimerkiksi kaikki viittaukset askelmäärityksistä instanssimuuttujiin viittaavat maailman instanssimuuttujiin ja niihin pääsee käsiksi myös muista askelmäärityksistä. Maailmaan voi lisätä omia apumoduuleita ja apumetodeita avustamaan askelmääritysten toteuttamisessa. Koska askelmääritykset kannattaa yleensä pitää lyhyinä ja yksinkertaisina, kannattaa käyttää

apumoduuleita ja -metodeja monimutkaisempien asioiden toteuttamiseen. [17, s. 120-125]

4.1.7 Koukut

Cucumber tarjoaa mahdollisuuden tehdä toimenpiteitä ennen tapausten tai askelien ajamista koukkujen avulla. Koukkuja on muutamia erilaisia. Yleisimpiä koukkuja ovat ennen (Before), jälkeen (After) sekä ennen ja jälkeen (Around) ajettavat koukut. Koukkujen avulla voidaan saattaa järjestelmä haluttuun tilaan ennen tapausten suorittamista. Koukuille voidaan kertoa tagien avulla, mitä tapauksia ennen ne suoritetaan.

```
Around('@fast') do
  system.logout
end
```

Koodiesimerkki 5. Ennen ja jälkeen ajettava koukku @fast-tagilla

Jokaisen askeleen jälkeen suoritettavia toimintoja varten on olemassa AfterStep-koukku. [17, s. 147-150]

4.1.8 Cucumber suomenkielisenä

Cucumberilla pystyy kirjoittamaan ominaisuuskuvauksia myös suomen kielellä, jolloin avainsanat vaihtuvat suomenkielisiksi. Seuraavassa esimerkissä on suomen kielellä toteutettu ominaisuuskuvaus.

```
# language: fi
Ominaisuus: Kertolasku
   Välttyäkseen rahan menettämiseltä on kauppiaan pystyttävä laskemaan kertolaskuja

 Tapaus: Kertolasku kokonaisluvuilla
     Oletetaan että olen syöttänyt sovellukseen luvun 5
     Ja että olen syöttänyt sovellukseen luvun 4
     Kun painan nappia ”Kerro”
     Niin ruudulla pitäisi näkyä 20
```

Koodiesimerkki 3. Suomenkielinen ominaisuuskuvaus

Näille suomenkielisille ominaisuuskuvauksille tehdään askelmääritykset aivan kuten englanninkielisille ominaisuuskuvauksillekin. [18.]

```
Given /että olen syöttänyt sovellukseen luvun (\d+)/ do |n|
  @laskin.lisaa n.to_i
end

When /painan nappia "(\w+)"/ do |op|
  @tulos = @laskin.laske op
end

Then /ruudulla pitäisi näkyä (.*)/ do |tulos|
  @tulos.should == tulos.to_f
end
```

Koodiesimerkki 4. Askelmääritys suomenkielisille askelille

4.1.9 Cucumber-jvm

Cucumber-jvm on Cucumberin Java-virtuaalikoneelle tehty versio. Cucumber-jvm tukee yleisimpiä Java-virtuaalikoneelle olevia ohjelmointikieliä, kuten Javaa, Groovya, Clojurea ja Scalaa.

Askelmäärityksiä lukuun ottamatta Cucumber-jvm toimii samoin kuin Cucumberkin. Cucumber-jvm:in askelmääritykset kirjoitetaan Javalla.

Cucumber-jvm:ää käytetään yleensä yhdessä jonkun koontityökalun kanssa. Cucumber-jvm toimii erityisen hyvin Mavenin kanssa, mutta sen saa myös toimimaan muiden, kuten esimerkiksi Antin ja Gradlen, kanssa. [19.]

4.2 JBehave

JBehave sai alkunsa vuoden 2003 lopussa, kun käyttäytymislähtöisen kehityksen pioneeri Dan North päätti toteuttaa työkalun, joka vastaisi hänen kuvaansa siitä, mitä on käyttäytymislähtöinen kehitys. Työkalun oli tarkoitus olla JUnitin korvaaja. JBehave on BDD:n toteuttamiseen tarkoitettu ohjelmistokehys. Toiminnallisuuksiltaan JBehave on hyvin samankaltainen kuin Cucumber.

4.2.1 Tarinat

Kun Cucumberissa kirjoitettiin ominaisuuskuvauksia, kirjoitetaan JBehavessa tarinoita (eng. narrative), jotka ovat käytännössä sama asia eri nimellä. JBehavessa tarinoita voidaan kirjoittaa sekä JBehaven omalla syntaksilla, että Gherkin-syntaksilla. [10; 12.]

JBehaven tarina muistuttaakin hyvin pitkälle Cucumberin ominaisuuskuvauksia ja on lähes identtinen, jos käytetään Gherkin-syntaksia. [12.]

```
Narrative:
 In order to find out the product of two numbers
 As a user
 I want to use a calculator to multiply two numbers

 Scenario: Multiply two numbers numbers

 Given there is a calculator
 When I multiply <number1> and <number2>
 Then the outcome should be <result>

 Examples:
 |number1|number2|result|
 |  10   |  10   |  100 |
```

Koodiesimerkki 5. JBehaven tarina

4.2.2 Askeleiden sitominen Javaan ja annotaatiot

JBehaven tarinoissa käytetyt askeleet yhdistetään Javaan annotaatioiden avulla. Käytetyimmät annotaatiot ovat @Given, @When ja @Then. Jokainen askel-annotaatio sisältää säännöllisen lausekkeen, johon eri tarinoissa esiintyviä askeleita verrataan. Samaan Java-metodiin voi sitoa useampia eri askelia aliaksien avulla.

Esimerkiksi metodiin gettingPaid, voidaan yhdistää kaksi eri askelta ”When customer pays $price” ja ”When customer hands over $price”).

```
@When("customer pays $price")
@Alias("customer hands over $price")
public void gettingPaid(double price){
  // ...
}
```

Koodiesimerkki 6. Askelien sitominen Java-metodiin

Jos tarinassa olevaa askelta ei ole kytketty mihinkään Java-metodiin, on se tilassa ”pending” ja odottaa toteutusta. Askeleen voi myös asettaa itse ”pending”-tilaan @pending-annotaation avulla.

Koodiesimerkissä 5 esitetyn tarinan askelille voidaan toteuttaa askelkuvaukset esimerkiksi seuraavan java-koodin avulla.

```
public class MultiplyTwoNumbersSteps {

   private Calculator calculator;

   @Given("there is a calculator")
   public void givenThereIsACalculator() {
     calculator = new Calculator();
   }

   @When("I multiply <number1> and <number2>")
   public void whenIMultiplyNumber1AndNumber2(@Named("number1")int number1,
@Named("number2")int number2) {
     calculator.multiply(number1, number2);
   }
   @Then("the outcome should be <result>")
   public void thenTheOutcomeShouldBe(@Named("result")int result) {
     assert calculator.getResult() == result;
   }
 }
```

Koodiesimerkki 6. JBehaven askelmääritys

Askelissa käytettävien annotaatioiden lisäksi JBehavessa on kaksi tapausannotaatiota: @BeforeScenario ja @AfterScenario. Niiden avulla voidaan tehdä toimenpiteitä ennen ja jälkeen tapausten. Muita annotaatiotyyppejä ovat tarina-annotaatiot (engl. story annotations) ja tarinat-annotaatiot (engl. stories annotations). Näiden avulla voidaan toteuttaa toimenpiteitä ennen tarinoita ja niiden jälkeen. [12.]

4.2.3 Tarinoiden suorittaminen

JBehave on tarkoitettu hyvin erilaisiin kehitysympäristöihin sulautuvaksi. Testejä pystyykin ajamaan usealla tavalla. JBehave-testejä voi ajaa JUnit-testeinä, jolloin niitä on helppo suorittaa graafisista kehitysympäristöistä. Tarinoita voi ajaa myös koontityökalujen, kuten Antin tai Mavenin avulla. [12.]

4.2.4 Parametrisoidut tapaukset

Parametrisoidut tapaukset muistuttavat hyvin paljon Cucumberin tapausaihioita. Parametrisoitujen tapauksien avulla pystytään suorittamaan samankaltaisia tapauksia eri arvoilla helposti. [12.]

```
Given the price is <price>
When I give <money> to the clerk
Then I should get <change> back

Examples:
|price|money | change|
| 4.0 | 10.0 | 6.0   |
| 8.0 | 25.0 | 17.0  |
```

Koodiesimerkki 7. Parametrisoitu tapaus

4.2.5 Meta-tieto

Tarinoihin ja tapauksiin on mahdollista lisätä meta-tietoa. Meta-tiedon lisäys onnistuu ”Meta”-avainsanan avulla.

```
Meta:
@author Arttu
@theme fast ui

Scenario:
..
```

Koodiesimerkki 8. Meta-tietoa tapaukselle

Meta-tietoa voi käyttää hyväkseen, Cucumberin tagien tapaan, ajettaessa tarinoita. Meta-tiedon avulla voidaan ajaa, vaikka vain nopeasti suoritettavat testit. [12.]

4.3 Behat

Behat on Cucumberin innoittamana PHP:lle suunniteltu työkalu, joka mahdollistaa BDD:n toteuttamisen. Cucumberin lailla Behat käyttää Gherkin-kieltä ominaisuuskuvauksissaan. Askelten määritykset kirjoitetaan PHP:lla.

Behatin ominaisuuskuvaukset ovat Gherkinistä johtuen lähes identtisiä Cucumberin ominaisuuskuvauksien kanssa ja niiden kirjoittaminen tapahtuu samalla lailla. Behat tukee myös tageja Cucumberin tapaan.

```
Scenario: List files
  Given I have a file called "testitiedosto.txt"
  When I run "ls"
  Then I should see "testitiedosto.txt"
```

Koodiesimerkki 8. Behatin ominaisuuskuvaus

Kun komentoriviltä ajetaan komento “behat”, tunnistaa Behat ominaisuuskuvauksen ja antaa askelien toteutukselle pohjan, kuten Cucumberkin. Myös Behatin askelkuvauksissa voi ja kannattaa käyttää säännöllisiä lauseita.

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

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

class FeatureContext extends BehatContext
{
  /**
    * @Given /^There is a file called "([^"]*)"$/
    */
      public function thereIsAFileCalled($file)
      {
        touch($file);
      }

  /**
    * @When /^I run "([^"]*)"$/
    */
      public function iRun($cmd)
      {
        exec($cmd,$result);
        $this->output = $result;
      }

      /**
       * @Then /^I should see "([^"]*)"$/
       */
      public function iShouldSee($name)
      {
        if (!in_array($fileName,$this->output)) {
          throw new Exception('File '.$name.' not found!');
        }
      }
}
?>
```

Koodiesimerkki 9. Behatin askelmääritykset

Behat tukee monia samoja ominaisuuksia kuin Cucumber. Tämmöisiä ominaisuuksia ovat esimerkiksi koukut. Behatin koukut eroavat hieman Cucumberin vastaavista, mutta toimintaperiaate on samankaltainen. [20.]

4.3.1 MINK

Behatin edut tulevat parhaiten esille testattaessa PHP:lla toteutettuja web-sovelluksia. Web-sovellusten funktionaaliseen testaukseen Behatilla käytetään yleensä apuna MINK-selainemulaattoria. MINK yhdistää selainemulaattoreiden (esim. Goutte) ja selainohjainten (esim. Selenium) parhaat puolet ja mahdollistaa web-sovellusten monipuolisen testauksen.

MINK käyttää ajureita, joiden avulla se kommunikoi erilaisten testityökalujen kanssa. MINKin mukana tulee ajurit Gouttelle, Sahille, Seleniumille ja Selenium2:lle. [21.]

4.3.2 Komentorivi-työkalu

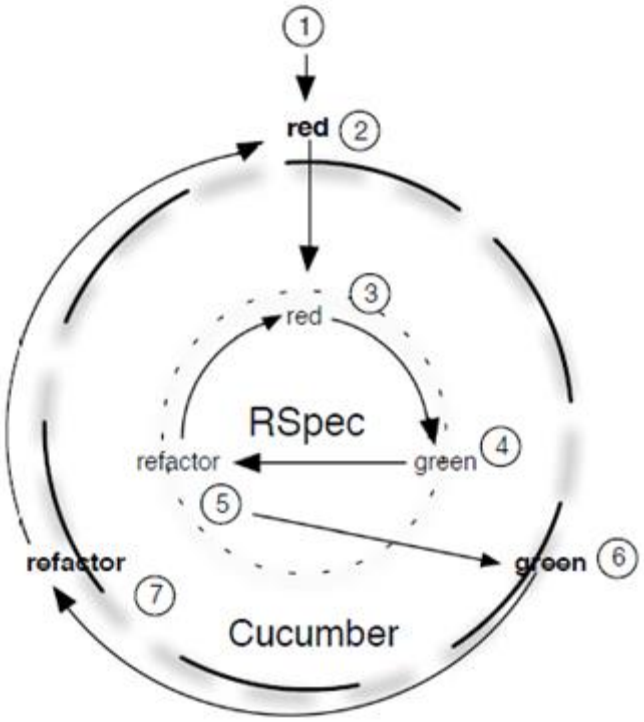
Behat-testien ajaminen onnistuu Behatin komentorivi-työkalun avulla. Komentorivin ominaisuudet muistuttavat pitkälti Cucumberin-komentorivityökalua. Komentoriviargumenttien avulla pystytään esimerkiksi valitsemaan, missä muodossa testien tulokset näytetään, mihin mahdollisesti tallennetaan testien tulokset ja millä tageilla varustetut ominaisuudet ajetaan.

```
$ behat --format html --out raportti.html
```

Koodiesimerkki 10. Behatin ajo komentoriviltä

4.4 RSpec

RSpec on testaustyökalu Rubylle, joka on alun perin kirjoitettu BDD:tä silmälläpitäen. RSpecin avulla kirjoitetaan ajettavia esimerkkejä sovelluksen oletetusta toiminnasta [9 s. 25-26]. RSpecillä testataan yleensä alemman tason koodia (vaikka sillä pystyy myös testaamaan korkean tason toiminnallisuutta), kun taas Cucumberin avulla testataan yleensä toiminnallisuutta ylemmällä tasolla. RSpeciä käytetään yleisesti yhdessä Cucumberin kanssa [9 s. 27-29]. Cucumberin ja RSpecin yhteistoimintaa voi kuvata seuraavalla kuvalla.



Kuva 5. BDD-sykli Cucumberin ja RSpecin kanssa[9 s. 19]

Kuvassa näkyy seitsemän eri vaihetta. Ensimmäisessä vaiheessa keskitytään miettimään, minkälainen ominaisuus tuotteeseen halutaan seuraavaksi. Tässä vaiheessa kirjoitetaan ominaisuuskuvaus, jossa kerrotaan, kuinka ominaisuuden halutaan toimivan. Toisessa vaiheessa kirjoitetaan epäonnistuva (red) askelmääritys ensimmäiselle askeleelle. Nämä kaksi ensimmäistä vaihetta (kuvassa ulompi kehä) liittyvät Cucumberiin.

Kuvan toisen vaiheen jälkeen siirrytään työskentelemään RSpecin kanssa (kuvassa sisempi kehä). Kolmannessa vaiheessa kirjoitetaan epäonnistuva RSpecin esimerkki. Neljännessä vaiheessa toteutetaan toiminnallisuutta niin paljon, että saadaan epäonnistuva RSpecin esimerkki läpi (green). Viidennessä vaiheessa refaktoroidaan koodia paremmaksi. Vaiheita kolme, neljä ja viisi toistetaan, kunnes Cucumberin askel menee läpi. Kuvan viimeisessä eli seitsemännessä vaiheessa refaktoroidaan jälleen koodia paremmaksi.

Seuraavassa koodissa on yksinkertaisia esimerkkejä RSpec-testeistä, jotka kertovat, miten sovelluksen tulisi toimia. Tässä on kolme esimerkkiä, joista ensimmäisessä kerrotaan, että jos @featurella ei ole nimeä, se ei ole validi. Toisessa kerrotaan, että

@featurella tulee olla kuvaus ja viimeisessä, että @featuren nimi ei saa olla yli 140 merkkiä pitkä.

```
describe "when name is not present" do
  before { @feature.name = " " }
  it { should_not be_valid }
end

describe "when description is not present" do
  before { @feature.description = " " }
  it { should be_valid }
end

describe "when name is too long" do
  before { @feature.name = "x" * 141 }
  it { should_not be_valid }
end
```

Koodiesimerkki 10. RSpec-testit

RSpecia käytetään yleisesti Cucumberin apuna askelmäärityksissä. RSpec soveltuu hyvin erilaisten Ruby-sovellusten testaamiseen ja toimii hyvin yhteistyössä esimerkiksi Cucumberin kanssa.

4.4.1 Odotukset

Odotus (eng. expectation) on sovelluksen odotettu toiminta. Odotus vastaa xUnitin termiä ”assert”. RSpec käyttää hyödykseen Rubyn tarjoamia should ja should_not-metodeita. Metodit should- ja should_not ottavat vastaan matchereita. [9, s. 150]

Yksi yleinen matcher on equal, jonka avulla voidaan tarkistaa, onko vastaus halutunlainen. RSpecissä tulee mukana useita eri matchereita kuten include ja raise_error. Merkkijonosta voidaan etsiä toista merkkkijonoa include-matcherin avulla. Jos halutaan testata, aiheuttaako metodi virheen, voidaan käyttää apuna raise_error-matcheria. [9, s. 171-176]

```
result.should equal(9)
```

Koodiesimerkki 11. should-metodi, joka käyttää equal-matcheria

4.4.2 Koodiesimerkit

Esimerkki (engl. example) vastaa xUnitin testimetodia. Esimerkki on ajettava esimerkki, siitä, miten tutkittavaa koodia voidaan ajaa ja miten sen odotetaan toimivan. BDD:ssä kirjoitetaan esimerkit ennen tutkittavaa koodia. On mahdollista luoda myös esimerkkiryhmiä, jotka koostuvat useista esimerkeistä.

Esimerkkiryhmiä luodaan describe- tai context-metodien avulla, jotka molemmat viittaavat samaan metodiin. Koodiesimerkkejä voidaan luoda it-metodin avulla. RSpecissä on usein samalle metodille useita eri aliaksia, jotta pystytään kirjoittamaan esimerkit mahdollisimman kuvaaviksi (BDD:ssä sanavalinnoilla on väliä). Eriteltävä toiminnallisuus kerrotaan merkkijonossa, joka annetaan it-metodille parametrina. Samaan it-metodiin voidaan viitata it-metodin sijasta myös specify- tai example-metodien avulla.

RSpecin testit kirjoitetaan spec-tiedostoihin, jotka voivat sisältää yhden tai useamman esimerkkiryhmän. [9, s. 150-158]

5 Käyttäytymislähtöinen kehitys käytännössä

Tässä osiossa tutustutaan käyttäytymislähtöiseen kehitykseen käytännössä kehittämällä projektinhallintatyökalu. Tarkoituksena on käydä yksityiskohtaisesti läpi projektinhallintatyökalun kehittäminen käyttäytymislähtöisen kehityksen käytäntöjä apuna käyttäen.

Tarkoituksena on kehittää yksinkertainen työkalu, joka soveltuu käyttäytymislähtöisesti kehitettävien projektien hallintaan. Monesti kun tuotetta lähdetään kehittämään, ei ole kovin tarkkaa kuvaa siitä, minkälainen tuotteesta halutaan. Näin on myös tässä tapauksessa. Käyttäytymislähtöisen kehityksen avulla yritetään saada selville, minkälainen sovellus tarvitaan. Kuva siitä, minkälainen sovelluksesta tulee, tarkentuu jokaisen uuden ominaisuuden kohdalla.

5.1 Kehityksessä käytettävät työkalut

Projektinhallintatyökalusta tulee selainkäyttöinen, ja sen kehityksessä käytetään apuna Ruby on Rails -ohjelmistokehystä. Ruby on Rails on Ruby-ohjelmointikieleen pohjautuva avoimen lähdekoodin ohjelmistokehys [25]. Ruby on Rails käyttää MVC-arkkitehtuuria [26]. MVC-arkkitehtuurissa sovellus jakaantuu kolmeen eri osaan: malliin (engl. Model), näkymään (engl. View) ja kontrolleriin (engl. Controller) [27].

Kehityksessä käytetään apuna käyttäytymislähtöisen kehityksen työkalua Cucumberia. Cucumberin askelmärityksien toteutuksissa käytetään apuna Capybara- ja RSpec-testityökaluja. Capybaran avulla simuloidaan interaktiota sovelluksen ja käyttäjän välillä [28]. RSpecistä käytetään apuna matchereita, joiden avulla voidaan tarkistaa, onko saatu tulos halutunkaltainen.

5.2 Ensimmäinen ominaisuus: Ominaisuuksien katselu

Koska halutaan mahdollisimman nopeasti aikaan hyödyllinen sovellus, täytyy miettiä, mitkä ovat tärkeimmät ominaisuudet, jotka sovellus tarvitsee. Jotta työkalusta olisi mitään apua projektinhallinnassa, tulisi sen osata kertoa, mitä on projektissa tekemättä. Koska tarkoituksena on luoda työkalu, jota käytetään nimenomaan käyttäytymislähtöisessä kehityksessä, päätetään, että sovelluksessa pidetään kirjaa ominaisuuksista, jotka projektissa kehitettävään tuotteeseen tulee. Ensimmäisenä ja tärkeimpänä sovelluksen tulisi siis osata listata ominaisuudet, jotka projektinhallintatyökalun avulla kehitettävässä projektissa kehitetään.

Kun on päätetty, mikä ominaisuus projektinhallintatyökaluun halutaan kehittää, kirjoitetaan siitä Cucumberin ominaisuuskuvaus. Ominaisuuskuvauksessa kerrotaan, että jos työkalussa luodaan ominaisuudet ”Post a Message” ja ”See Messages”, tulisi niiden näkyä ominaisuudet-sivulla.

```
Feature: See Features
  Scenario: See all the features
    Given there is a feature called "Post a message"
    And there is a feature called "See messages"
    When I visit the features page
    Then I should see
      """
      Post a message
      See messages
      """
```

Koodiesimerkki 12. features/see_features.feature

Kun ensimmäinen ominaisuuskuvaus on kirjoitettu, ajetaan Cucumber komentoriviltä. Cucumber ilmoittaa keltaisella värillä ne askelmäärityket, joita ei ole vielä toteutettu, ja ehdottaa koodinpätkän, jonka avulla askelmäärityksen voi toteuttaa.

```
arttu@arttu-VirtualBox: ~/rails_projects/example_project
arttu@arttu-VirtualBox:~/rails_projects/example_project$ cucumber
Using the default profile...
Feature: See Features

  Scenario: See all the features                    # features/see_features.feature:2
    Given there is a feature called "Post a message" # features/see_features.feature:3
      Undefined step: "there is a feature called "Post a message"" (Cucumber::Undefined)
      features/see_features.feature:3:in `Given there is a feature called "Post a message"'
    And there is a feature called "See messages"     # features/see_features.feature:4
      Undefined step: "there is a feature called "See messages"" (Cucumber::Undefined)
      features/see_features.feature:4:in `And there is a feature called "See messages"'
    When I visit the features page                   # features/see_features.feature:5
      Undefined step: "I visit the features page" (Cucumber::Undefined)
      features/see_features.feature:5:in `When I visit the features page'
    Then I should see                                # features/see_features.feature:6
      """
      Post a message
      See messages
      """
      Undefined step: "I should see" (Cucumber::Undefined)
      features/see_features.feature:6:in `Then I should see'

1 scenario (1 undefined)
4 steps (4 undefined)
0m0.375s

You can implement step definitions for undefined steps with these snippets:

Given(/^there is a feature called "(.*?)"$/) do |arg1|
  pending # express the regexp above with the code you wish you had
end

When(/^I visit the features page$/) do
  pending # express the regexp above with the code you wish you had
end

Then(/^I should see$/) do |string|
  pending # express the regexp above with the code you wish you had
end

arttu@arttu-VirtualBox:~/rails_projects/example_project$
```

Kuva 6. Cucumberin ajaminen komentoriviltä ensimmäisen ominaisuuskuvauksen jälkeen.

Ensimmäinen askelmääritys ”Given there is a feature called ’Post a message” kertoo, että tietokannassa tulisi olla ominaisuus nimeltään “Post a message”. Koska sovelluksessa ei ole vielä mitään tapaa lisätä ominaisuuksia tietokantaan, käytetään apuna FactoryGirl-työkalua, jonka avulla voidaan luoda testaamista varten ominaisuuksia tietokantaan [29]. Ensimmäisessä askelmäärityksessä luodaan FactoryGirl.create-metodin avulla uusi ominaisuus.

```
Given(/^there is a feature called "(.*?)"$/) do |feature_name|
  FactoryGirl.create(:feature, :name => feature_name)
end
```

Koodiesimerkki 13. features/step_definitions/feature_steps.rb

Kun tämän jälkeen ajetaan Cucumber, saadaan ensimmäinen askel epäonnistumaan. Tämän Cucumber näyttää punaisella värillä. Cucumber kertoo myös, minkä takia askel epäonnistuu. Tässä tapauksessa ei ole olemassa FactoryGirlin tehdasta, jolla pystyy luomaan ominaisuuksia.

```
  Scenario: See all the features                    # features/see_features.feature:2
    Given there is a feature called "Post a message" # features/step_definitions/feature_step
s.rb:1
      Factory not registered: feature (ArgumentError)
      ./features/step_definitions/feature_steps.rb:2:in `/^there is a feature called "(.*?)"$
/'
      features/see_features.feature:3:in `Given there is a feature called "Post a message"'
    And there is a feature called "See messages"     # features/step_definitions/feature_step
s.rb:1
    When I visit the features page                  # features/see_features.feature:5
      Undefined step: "I visit the features page" (Cucumber::Undefined)
      features/see_features.feature:5:in `When I visit the features page'
    Then I should see                               # features/see_features.feature:6
      """
      Post a message
      See messages
      """
      Undefined step: "I should see" (Cucumber::Undefined)
      features/see_features.feature:6:in `Then I should see'

Failing Scenarios:
cucumber features/see_features.feature:2 # Scenario: See all the features

1 scenario (1 failed)
4 steps (1 failed, 1 skipped, 2 undefined)
0m0.427s
```

Kuva 7. Epäonnistuva askelkuvaus

Seuraavaksi luodaan FactoryGirlin avulla tehdas, jonka avulla pystytään luomaan testaamista varten ominaisuuksia. FactoryGirlille pitää kertoa, mitä tietoa ominaisuus sisältää. Tässä tapauksessa ominaisuudella on nimi, joka on merkkijono.

```
require 'factory_girl'

FactoryGirl.define do
  factory :feature do |f|
    f.name 'Post a message'
  end
end
```

Koodiesimerkki 14. features/support/factories.rb

Askel epäonnistuu edelleen, mutta eri syystä. Cucumber ei tiedä, mikä Feature on.

```
  Scenario: See all the features                    # features/see_features.feature:2
    Given there is a feature called "Post a message" # features/step_definitions/feature_step
s.rb:1
      uninitialized constant Feature (NameError)
      ./features/step_definitions/feature_steps.rb:2:in `/^there is a feature called "(.*?)"$
/'
      features/see_features.feature:3:in `Given there is a feature called "Post a message"'
```

Kuva 8. Askel epäonnistuu

Askel saadaan onnistumaan luomalla Ruby on Railsin malli ominaisuudelle. Onnistuneen askeleen Cucumber ilmoittaa vihreällä värillä. Mallit ovat Railsissa Ruby-luokkia, jotka keskustelevat tietokannan kanssa ja validoivat dataa. Rails mahdollistaa mallin generoimisen komentoriviltä, jolloin se luo tarvittavat tiedostot. Tämän jälkeen täytyy siirtää muutokset tietokantaan rake db:migrate-komennolla ja valmistaa testitietokanta db:test:prepare-komennolla.

```
arttu@arttu-VirtualBox:~/rails_projects/example_project$ rails generate model Feature name:st
ring
      invoke  active_record
      create    db/migrate/20140320120346_create_features.rb
      create    app/models/feature.rb
      invoke    rspec
      create      spec/models/feature_spec.rb
arttu@arttu-VirtualBox:~/rails_projects/example_project$ rake db:migrate db:test:prepare
==  CreateFeatures: migrating =================================================
-- create_table(:features)
   -> 0.0017s
==  CreateFeatures: migrated (0.0020s) ========================================
```

Kuva 9. Mallin generoiminen

Tämän jälkeen kaksi ensimmäistä askelta menee läpi.

Seuraava vaihe ensimmäisen ominaisuuden toteuttamisessa on toteuttaa askel ”When I visit the features page”. Siinä ladataan ominaisuudet-sivu, jonka on tarkoitus sisältää

listan ominaisuuksista. Luodaan tiedosto features/step_definitions/navigation_steps.rb, jossa määritetään sivustolla navigointiin liittyviä askelia. Tiedostoon kirjoitetaan askel ominaisuudet-sivun latausta varten.

```
When(/^I visit the features page$/) do
  visit(features_path)
end
```

Koodiesimerkki 15. features/step_definitions/navigation_steps.rb

Askelmäärityksessä käytetään apuna Capybaran visit-metodia, jolle kerrotaan, mikä sivu ladataan. Kun jälleen ajetaan Cucumber, saadaan askel epäonnistumaan.

```
    When I visit the features page                    # features/step_definitions/navigation_s
teps.rb:1
      undefined local variable or method `features_path' for #<Cucumber::Rails::World:0xad4f7
10> (NameError)
      ./features/step_definitions/navigation_steps.rb:2:in `/^I visit the features page$/'
      features/see_features.feature:5:in `When I visit the features page'
```

Kuva 10. Askel epäonnistuu, koska features_path muuttujaa ei ole määritelty

Askel epäonnistuu, koska muuttujaa tai metodia nimeltä features_path ei ole vielä määritelty. Virhe saadaan korjattua määrittelemällä tiedostoon config/routes.rb Ruby on Railsin resurssi ominaisuuksille. Kun Railsissa luodaan resurssi, saadaan oletusreititys haluttavalle resurssille. Tässä tapauksessa ominaisuudelle saadaan reittejä esimerkiksi toiminnoille kuten ”new”, ”create”, ”show”.

```
ExampleProject::Application.routes.draw do

  resources :features

end
```

Koodiesimerkki 15. config/routes.rb

Resurssin luominen antaa myös käyttöön URL-avustajia (engl. URL-helper), jotka viittaavat eri polkuihin. Esimerkkejä URL-avustajista ovat esimerkiksi ”features_path”, joka viittaa polkuun ”/features” ja ”new_feature_path” joka viittaa polkuun ”/features/new”. Askelmäärityksessä käytetty ”visit(features_path)” kertoo capybaran visit-metodille, että ladattava sivu on osoitteessa ”/features”.

Taulukko 2. Resurssin luomat reitit

Polku	Toiminto	http-pyyntö
/features	index	GET
/features/new	new	GET
/features	create	POST
/features/:id	show	GET
/features/:id/edit	edit	GET
/features/:id	update	PATCH/PUT
/features/:id	destroy	DELETE

Askel ei mene vielä läpi, koska ominaisuuksille ei ole vielä määritelty kontrolleria. Kontrolleri sisältää sovelluslogiikan. Kun käyttäjä pyytää sivua ”/features”, ohjautuu pyyntö kontrollerin index-toiminnolle.

```
    When I visit the features page                    # features/step_definitions/navigation_s
teps.rb:1
      uninitialized constant FeaturesController (ActionController::RoutingError)
      ./features/step_definitions/navigation_steps.rb:2:in `/^I visit the features page$/'
      features/see_features.feature:5:in `When I visit the features page'
```

Kuva 11. Ominaisuuksille ei ole määritelty kontrolleria.

Jotta virhe saadaan korjattua, tulee luoda uusi kontrolleri FeaturesController ja sinne toiminto ”index”. Toiminto jätetään vielä tässä vaiheessa tyhjäksi.

```
class FeaturesController < ApplicationController
  def index
  end
end
```

Koodiesimerkki 16. app/controller/features_controller.rb

Tämänkään jälkeen ei askel mene läpi, vaan Cucumber valittaa, että ominaisuuksilta puuttuu näkymä. Näkymä sisältää ohjelman graafisen käyttöliittymän. Railsin tapauksessa siis lähinnä HTML- ja erb-koodia. erb-koodi on HTML:n sekaan syötettävää Ruby-koodia.

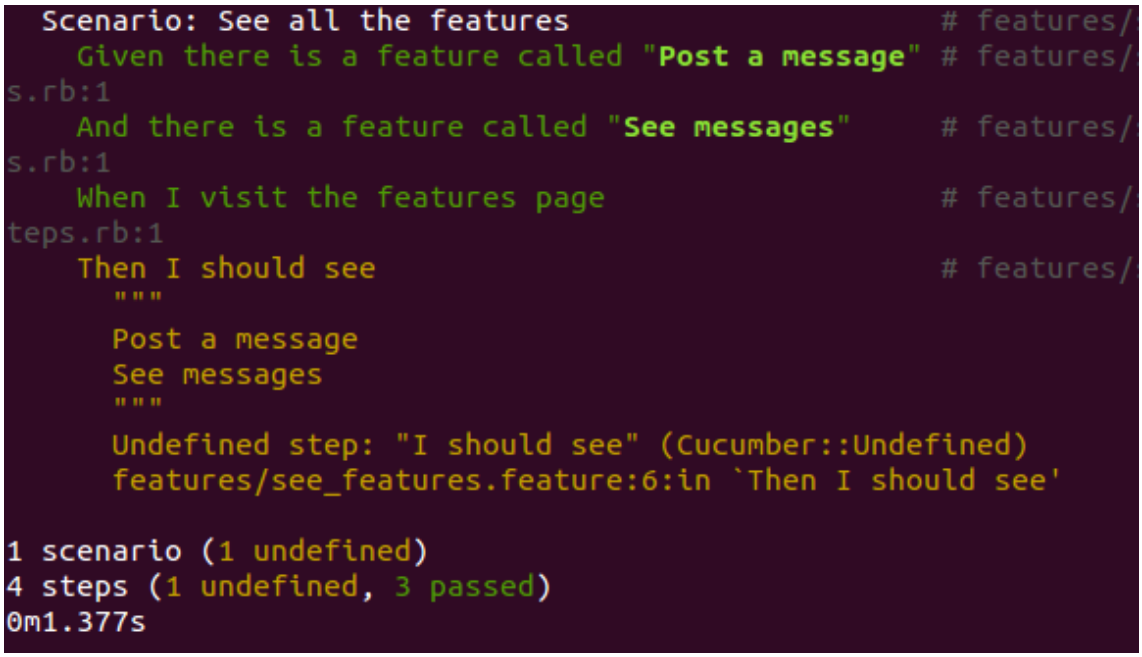
Kuva 12. Askel ei mene läpi, koska näkymä puuttuu.

Koska tarkoituksena on vain saada seuraava askel onnistumaan, luodaan näkymä ominaisuuksien index-toiminnolle väliaikaisella sisällöllä.

```
Making the step pass
```

Koodiesimerkki 17. app/view/features/index.html.erb

Tämän jälkeen näkymän lataaminen onnistuu ja kolmas askel onnistuu.



Kuva 13. Kolmas askel onnistuu

Ensimmäinen ominaisuus projektinhallintatyökalussa on yhtä askelta vaille valmis. Kirjoitetaan askelmääritys viimeiselle askeleelle, jossa etsitään ladatusta sivusta merkkijonoa, jossa on haluttujen ominaisuuksien nimet.

```
Given(/^there is a feature called "(.*?)"$/) do |feature_name|
  FactoryGirl.create(:feature, :name => feature_name)
end

Then(/^I should see$/) do |text|
   page.should have_content(text)
end
```

Koodiesimerkki 18. features/step_definitions/feature_steps.rb

Askelmäärityksen toteutuksessa käytetään apuna Capybaran page-oliota ja RSpecin have_content-matcheria. Page-olio sisältää aikaisemmassa askeleessa ladatun sivun, ja RSpecin have_content vertaa sivun sisältöä haluttuun merkkijonoon.

```
    Then I should see                                    # features/step_definitions/feature_step
s.rb:5
      """
      Post a message
      See messages
      """
      expected to find text "Post a message See messages" in "Lets get this step to fail" (RS
pec::Expectations::ExpectationNotMetError)
      ./features/step_definitions/feature_steps.rb:6:in `/^I should see$/'
      features/see_features.feature:6:in `Then I should see'
```

Kuva 14. Epäonnistuva askel

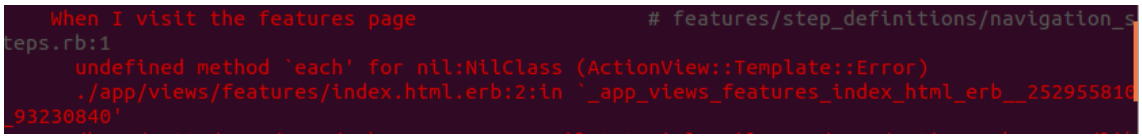
Askel epäonnistuu, koska näkymää ei ole toteutettu oikein, vaan siellä on vasta väliaikainen sisältö. Näkymää luotaessa olisi voitu jo kirjoittaa suoraan valmista koodia, mutta silloin ei nähtäisi keltainen-punainen-vihreä sykliä, joka on olennainen osa Cucumberin käyttämistä.

Kirjoitetaan näkymään todellinen koodi, mikä listaa kaikki ominaisuudet. Koodissa käydään läpi @features-olio, jonka tulisi sisältää kaikki tietokannasta löytyvät ominaisuudet.

```
<h1>Features:</h1>
<% @features.each do |f| %>
  <p><%= f.name %> </p><br />
<% end %>
```

Koodiesimerkki 19. app/views/features/index.html.erb

Askel ei kuitenkaan mene läpi, koska näkymä ei löydä @features-oliota. Saadaan virheilmoitus, että tyhjälle oliolle ei voi käyttää metodia each.



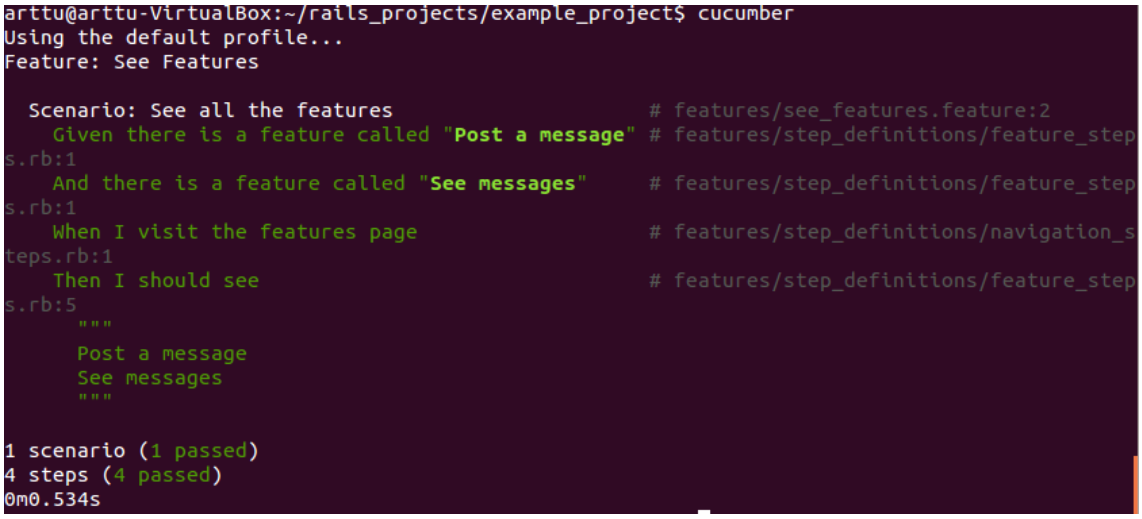
Kuva 15. Askel epäonnistuu, määrittelemättömän features-olion takia.

Viimeinenkin askel saadaan läpi, kun haetaan kontrollerissa kaikki ominaisuudet tietokannasta @features-muuttujaan.

```
class FeaturesController < ApplicationController
  def index
    @features = Feature.all
  end
end
```

Koodiesimerkki 20. app/controllers/features_controller.rb

Kun vielä kerran ajetaan Cucumber, nähdään, että kaikki neljä askelta menevät läpi. Ensimmäinen määritelty ominaisuus on siis valmis. Vielä olisi hyvä tarkistaa lisäämällä tietokantaan käsin muutama ominaisuus, käynnistää Railsin kehitykseen tarkoitettu web-palvelin ja katsoa, että kirjoitettu toiminnallisuus oikeasti toimii niin kuin oli tarkoitettu.



Kuva 16. Kaikki askeleet menevät läpi.

5.3 Ominaisuuksien lisääminen

Sovelluksesta ei vielä ole kovin paljon iloa. Jotta sovellus osaa näyttää ominaisuuksia, on ne lisättävä käsin tietokantaan. Tämä vaatii aika paljon tietoteknistä osaamista ja vaivannäköä. Tästä päästäänkin sovelluksen toiseksi tärkeimpään ominaisuuteen eli ominaisuuksien lisäämiseen. Sovellus, missä pystyy lisäämään ja tarkastelemaan ominaisuuksia, voi olla jo oikeasti hyödyllinen.

Ensimmäinen vaihe ominaisuuden toteuttamisessa on jälleen ominaisuuskuvauksen kirjoittaminen. Ominaisuuskuvaukseen lisätään tapaus uuden ominaisuuden lisäämiselle.

```
Feature: Add features
  Scenario: Add new feature
    Given I am on the create new feature page
    When I fill in the name field for the feature form with "Find users"
    And I click "Submit"
    And I visit the features page
    Then I should see "Find users"
```

Koodiesimerkki 21. features/add_features.features

Halutaan, että kun käyttäjä on uuden ominaisuuden lisäyssivulla, hänelle näkyy lomake, jossa on kenttä ominaisuuden nimeä varten. Kun käyttäjä kirjoittaa kenttään ominaisuuden nimen ja painaa Submit-painiketta, lisää sovellus tietokantaan uuden ominaisuuden. Kun käyttäjä tämän jälkeen vierailee ominaisuudet-sivulla, siellä todella näkyy tämä lisätty ominaisuus.

Jos tässä vaiheessa ajetaan Cucumber, se näyttää, että kaikki askeleet on toteuttamatta. Ensimmäinen toteutettava askel on ”Given I am on the create new feature page”, jossa navigoidaan uuden ominaisuuden lisäys sivulle.

```
Given(/^I am on the create new feature page$/) do
  visit(new_feature_path)
end
```

Koodiesimerkki 22. features/step_definitions/navigation_steps.rb

Tämän askelkuvauksen toteuttaminen saa askeleen epäonnistumaan, koska kontrollerissa ei ole toimintoa ”new”.

```
    Given I am on the create new feature page                          # features/step
_definitions/navigation_steps.rb:5
      The action 'new' could not be found for FeaturesController (AbstractController::Ac
tionNotFound)
      ./features/step_definitions/navigation_steps.rb:6:in `/^I am on the create new fea
ture page$/'
      features/add_feature.feature:3:in `Given I am on the create new feature page'
```

Kuva 17. Askel epäonnistuu, koska kontrollerissa ei ole toimintoa ”new”

Jotta askel saadaan läpi, täytyy kontrolleriin toteuttaa haluttu toiminto. Toiminnossa luodaan uusi Feature-olio ja asetetaan se muuttujaan @feature.

```
def new
  @feature = Feature.new
end
```

Koodiesimerkki 23. app/controllers/features_controller.rb

Tämänkään lisääminen ei kuitenkaan saa askelta läpäisemään, vaan Cucumber antaa virheilmoituksen puuttuvasta näkymästä.

```
    Given I am on the create new feature page                          # features/step
_definitions/navigation_steps.rb:5
      Missing template features/new, application/new with {:locale=>[:en], :formats=>[:h
tml], :handlers=>[:erb, :builder, :raw, :ruby, :jbuilder, :coffee]}. Searched in:
        * "/home/arttu/rails_projects/example_project/app/views"
       (ActionView::MissingTemplate)
      ./features/step_definitions/navigation_steps.rb:6:in `/^I am on the create new fea
ture page$/'
      features/add_feature.feature:3:in `Given I am on the create new feature page'
```

Kuva 18. Näkymä puuttuu uuden ominaisuuden lisäämiselle

Jotta askel menisi läpi, on toteutettava näkymä toiminnolle. Luodaan näkymä väliaikaisella sisällöllä.

```
Making the step pass
```

Koodiesimerkki 24. app/views/features/new.html.erb

Seuraavassa askeleessa täytetään lomakkeeseen nimikenttään luotavan ominaisuuden nimi. Askelmäärityksessä käytetään apuna Capybaran fill_in-metodia, jonka avulla pystyy kirjoittamaan lomakekenttiin.

```
When(/^I fill in the name field for the feature form with "(.*?)"$/) do |text|
   fill_in('feature_name', :with => text)
end
```

Koodiesimerkki 25. features/step_definitions/feature_steps.rb

Kun tässä vaiheessa ajetaan Cucumber, antaa se Capybaran ”ElementNotFound”-virheen. Tämä johtuu siitä, että näkymässä on vain väliaikainen teksti. Jotta tämä askel saadaan läpi, tulee oikeasti toteuttaa näkymä.

Näkymässä on tässä tapauksessa lomake uuden ominaisuuden lisäämistä varten. Lomakkeessa on kenttä ominaisuuden nimelle sekä painike lomakkeen lähettämistä varten.

```
<h1>Add new feature:</h1>

<div class="row">
  <div class="span6 offset3">
    <%= form_for(@feature) do |f| %>
    <table>
    <tr>
      <td><%= f.label :name %></td>
      <td><%= f.text_field :name %></td>
    </tr>
    <tr>
      <td><%= f.submit "Submit" %></td>
    <% end %>
    </table>
  </div>
</div>
```

Koodiesimerkki 26. app/views/features/new.html.erb

Seuraava askel liittyy lomakkeen lähettämiseen Submit-painiketta painamalla. Askeleen toteutuksessa käytetään taas apuna Capybaraa. Capybarassa on click_button-metodi, jonka avulla pystyy painamaan painiketta.

```
When(/^I click "(.*?)"$/) do |name|
  click_button(name)
end
```

Koodiesimerkki 27. features/step_definitions/feature_steps.rb

Askel ei mene heti läpi, vaan Cucumber valittaa, ettei se löydä FeatureControllerista toimintoa “create”. ”Create”-toiminnossa tallennetaan luotu ominaisuus tietokantaan.

```
def create
  @feature = Feature.new(feature_params)
  if @feature.save
  else
    render 'new'
  end
end

def feature_params
  params.require(:feature).permit(:name)
end
```

Koodiesimerkki 28. app/controllers/features_controller.rb

Toiminnon lisääminen kontrolleriin ei vielä saa askelta läpäisemään, vaan Cucumber kertoo, että puuttuu näkymä ”create”-toiminnolle. Näkymään laitetaan viesti onnistuneesta ominaisuuden lisäämisestä.

```
Feature was successfully created
```

Koodiesimerkki 29. app/views/features/create.html.erb

Tämä saa painikkeen painallusaskeleen onnistumaan. Kun ajetaan Cucumber, huomataan, että myös seuraava askel, navigoiminen ominaisuudet-sivulle, menee suoraan läpi. Yksi Cucumberin hienoista ominaisuuksista on, että askelkuvauksia voi käyttää hyödyksi useamman kerran ja useammassa eri ominaisuuskuvauksessa.

Uuden ominaisuuden lisäämisessä on jäljellä enää yksi askel. Ominaisuudet-sivulla pitäisi näkyä juuri luotu ominaisuus.

```
Then(/^I should see "(.*?)"$/) do |text|
  page.should have_content(text)
end
```

Koodiesimerkki 30. features/step_definitions/feature_steps.rb

Askelkuvauksen kirjoittamisen jälkeen myös viimeinen askel menee läpi. Kun ajetaan jälleen Cucumber, nähdään, että kaksi tapausta onnistuu ja ne sisältävät yhteensä yhdeksän onnistuvaa askelta.

```
2 scenarios (2 passed)
9 steps (9 passed)
0m0.649s
```

Kuva 19. Molemmat kaksi tapausta menevät läpi

Sovelluksen käyttäminen selaimen kautta onnistuu, mutta on vaivalloista ja vaatii tietämystä sovelluksesta. Käyttäjän täytyy navigoida sovelluksessa osoiterivin avulla sen sijaan, että sivustolla olisi linkit ominaisuudet-sivulle ja uuden ominaisuuden lisäämiseen. Linkkien toteuttaminen onkin seuraavaksi toteutettava ominaisuus.

```
Feature: Navigate
  Scenario: Navigate to New Feature Page
    Given I am on the Features page
    When I click the link for "New Feature"
    Then I should see "Add new feature"

  Scenario: Navigate to Features Page
    Given I am on the create new feature page
    When I click the link for "Features"
    Then I should see "Features:"
```

Koodiesimerkki 31. features/links.feature

Luodaan uusi ominaisuuskuvaus sivustolla navigointia varten. Ominaisuuskuvaukseen luodaan kaksi eri tapausta: uuden ominaisuuden luontisivulta ominaisuudet-sivulle navigoiminen ja ominaisuudet-sivulta uuden ominaisuuden luontisivulle navigoiminen.

Cucumberia ajettaessa nähdään, että tarvittavia askelmäärityksiä puuttuu. Kun lisätään askelmääritys ”Given I am on the Features page”, jossa Capybaran visit-metodilla ladataan ominaisuudet-sivu, saadaan ensimmäinen askel läpäisemään.

Seuraavaksi toteutetaan askel ”When I Click the link for ‘New Feature’”. Askelmäärityksessä käytetään apuna Capybaran click_link-metodia.

```
Given(/^I am on the Features page$/) do
  visit(features_path)
end

When(/^I click the link for "New Feature"$/) do
  click_link("New Feature")
end
```

Koodiesimerkki 32. features/step_definitions/navigation_steps.rb

Tämän askelmäärityksen toteuttaminen saa askeleen epäonnistumaan, koska ominaisuudet-sivulla ei ole linkkiä uuden ominaisuuden luontisivulle. Lisätään sivuille linkki niin, että lisätään se layout-tiedostoon ”app/views/layouts/application.html.erb”, jolloin linkki näkyy jokaisella sivulla automaattisesti.

```
<!DOCTYPE html>
<html>
<head>
  <title>ExampleProject</title>
  <%= stylesheet_link_tag    "application", media: "all", "data-turbolinks-track" =>
true %>
  <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
  <%= csrf_meta_tags %>
</head>
<div id="links">
<%= link_to "New Feature", new_feature_path %>
</div>
<body>

<%= yield %>

</body>
</html>
```

Koodiesimerkki 33. app/views/layouts/application.html.erb

Linkin lisääminen saa askeleen, ja siten myös tapauksen, läpäisemään. Toisestakin tapauksesta ensimmäinen askel menee läpi, koska kyseinen askelmääritys on kirjoitettu jo aikaisemmassa vaiheessa. Toisesta tapauksesta puuttuu määritys askeleelle ”When I click the link for ’Features’”. Tämä määritys voidaan toteuttaa samalla lailla kuin edellisessä tapauksessa uuden ominaisuuden lisäyssivulle navigointi toteutettiin. Järkevämpää on kuitenkin korjata aikaisempaa koodia säännöllisten lausekkeiden avulla niin, että molemmissa tapauksissa voidaan käyttää samaa askelmääritystä.

```
When(/^I click the link for "(.*?)"$/) do |link|
  click_link(link)
end
```

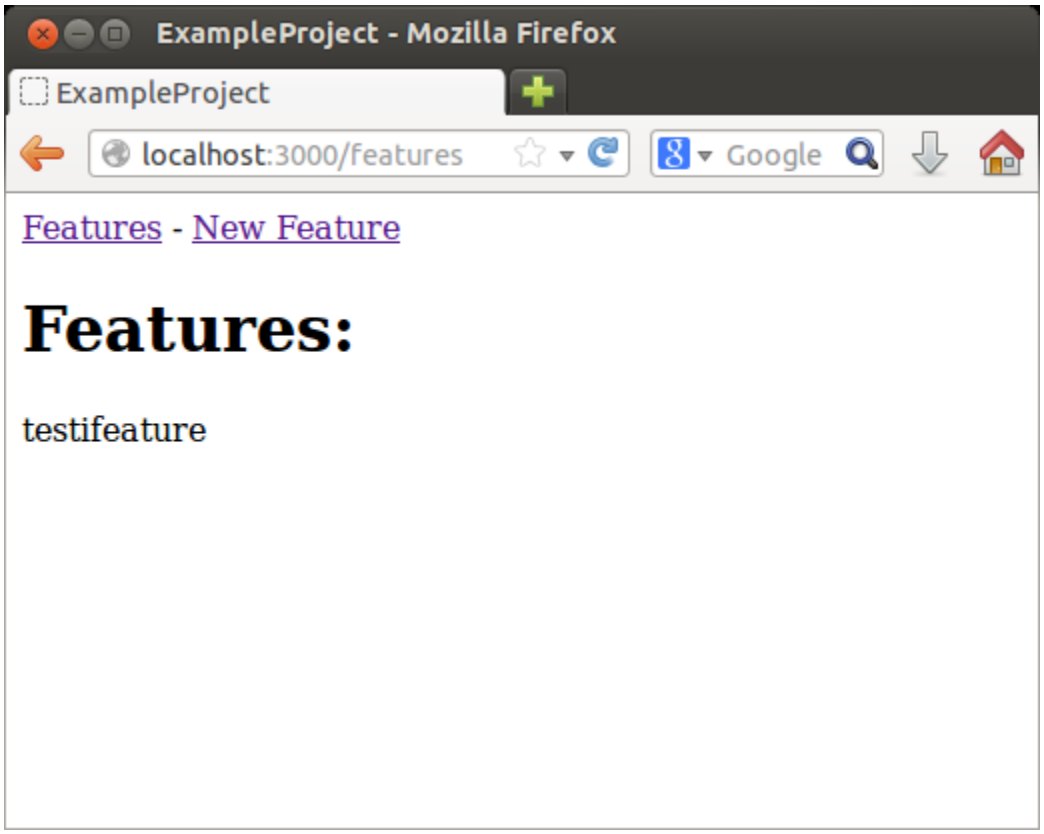
Koodiesimerkki 34. features/step_definitions/navigation_steps.rb

Tämän refaktoroinnin jälkeen askel epäonnistuu, ja saadaan tietää, että linkkiä ei löydy sivulta. Askel saadaan läpäisemään muokkaamalla jälleen sivuston pohjaa niin, että jokaisella sivulla näkyy linkki ominaisuudet-sivulle.

```
<!DOCTYPE html>
<html>
<head>
  <title>ExampleProject</title>
  <%= stylesheet_link_tag    "application", media: "all", "data-turbolinks-track" =>
true %>
  <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
  <%= csrf_meta_tags %>
</head>
<div id="links">
<%= link_to "Features", features_path %> -
<%= link_to "New Feature", new_feature_path %>
</div>
<body>

<%= yield %>

</body>
</html>
```

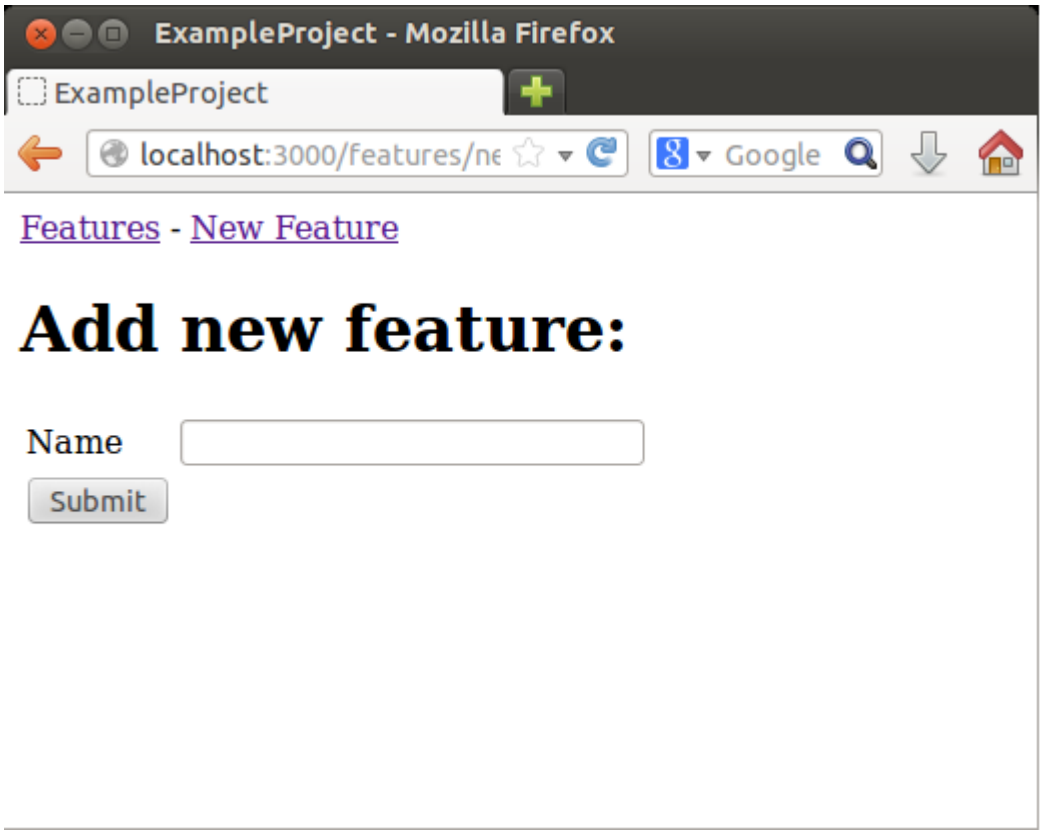
Koodiesimerkki 35. app/views/layouts/application.html.erb

Kun linkki on lisätty sivuston pohjaan, saadaan askel ja koko tapaus läpäisemään. Tässä vaiheessa viimeistään kannattaa käynnistää Railsin web-palvelin ja katsoa, miltä sovellus näyttää selaimessa.



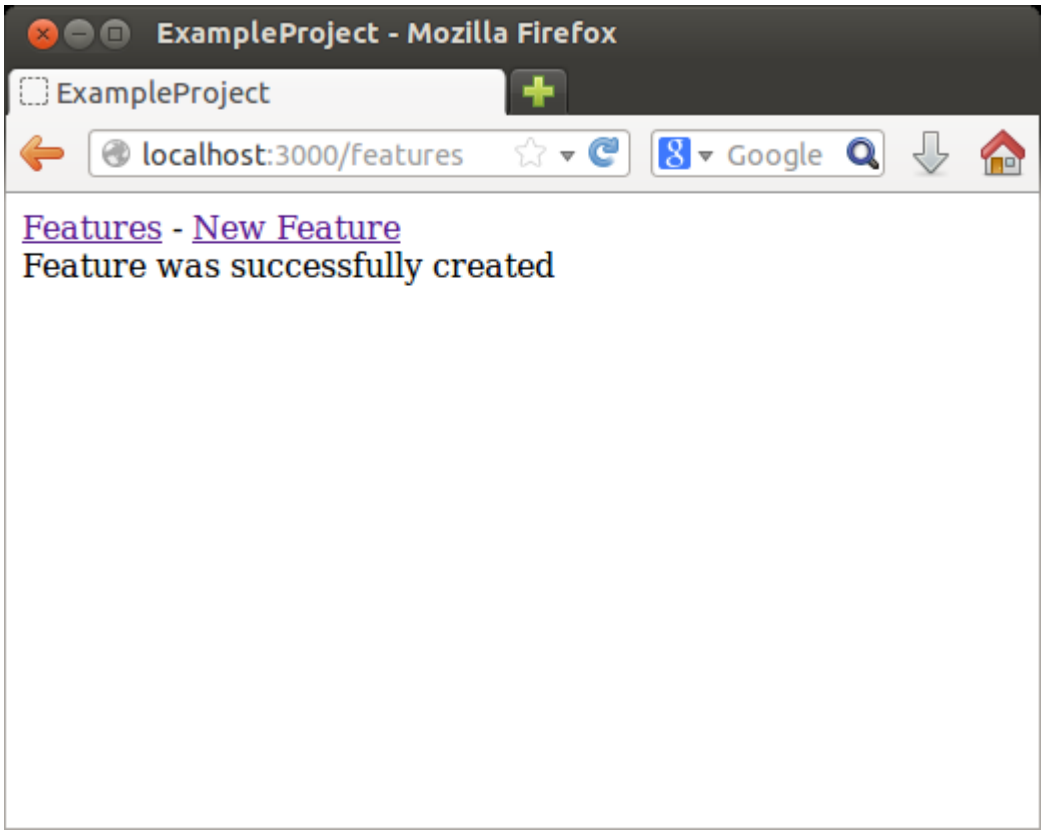
Kuva 20. Ominaisuudet-sivu

Ominaisuudet-sivulla näkyy työkaluun lisätyt ominaisuudet. Ensimmäisellä vierailukerralla ominaisuuslista on tyhjä. Sivun ylälaidassa näkyy jokaisella sivulla näkyvät linkit ominaisuudet-sivulle ja uuden ominaisuuden lisäyssivulle.



Kuva 21. Uuden ominaisuuden lisäyssivu

Uuden ominaisuuden lisäyssivulla näkyy lomake, jossa on kenttä ominaisuuden nimelle ja painike lomakkeen lähettämistä varten. Kun lomakkeen kenttään kirjoitetaan ominaisuuden nimi, painetaan Submit-painiketta, ja uuden ominaisuuden lisäys onnistuu. Käyttäjälle näkyy viesti onnistuneesta ominaisuuden lisäyksestä.



Kuva 22. Uuden ominaisuuden lisäys onnistui.

5.4 Ominaisuuksien poistaminen ja tehtävien lisääminen

Sovellus on vielä kovin yksinkertainen, eikä se vielä sellaisenaan sovellu projektinhallintaan. Sillä pystyy ainoastaan lisäämään uusia ominaisuuksia ja näkemään, mitä ominaisuuksia sinne on lisätty. Sovellus ei tiedä, missä tilassa ominaisuuden kehittäminen on. Onko ominaisuus valmis, vai tarvitseeko se kenties vielä joidenkin tiettyjen tehtävien toteuttamista? Ehkä koko ominaisuus on todettu turhaksi. Nämä ovat ongelmia, joita sovelluksen avulla ei vielä pysty ratkaisemaan, mutta jotka projektinhallintatyökalun olisi tarpeellista osata.

Seuraavaksi toteutetaan ominaisuuden poistaminen. Jos sovelluksesta pystyy poistamaan ominaisuuksia, voi sitä jo siinä vaiheessa käyttää jäljellä olevan työn tarkasteluun poistamalla valmiina olevat ominaisuudet. Ominaisuuden poistaminen tapahtuu ominaisuuden sivulta Delete-painiketta painamalla.

```
Feature: Delete features
  Scenario: Delete feature
    Given there is a feature called "Find users"
    And I am on the feature page
    And I click "Delete"
    And I visit the features page
    Then I should not see "Find users"
```

Koodiesimerkki 36. features/delete_feature.feature

Ensimmäinen askel menee läpi suoraan. Toisen askeleen määritys tapahtuu jälleen Capybaran visit-metodin avulla.

```
When(/^I am on the feature page$/) do
  visit(feature_path(Feature.first))
end
```

Koodiesimerkki 37. features/step_definitions/navigation_steps.rb

Askel epäonnistuu, koska kontrollerista ei löydy toimintoa show. Show-toiminnossa haetaan ominaisuuden tiedot tietokannasta, jotta niitä voidaan käyttää näkymässä.

```
def show
  @feature = Feature.find(params[:id])
end
```

Koodiesimerkki 38. app/controllers/features_controller.rb

Tämän jälkeen Cucumber valittaa, että näkymä puuttuu toiminnolle show. Askel saadaan menemään läpi, kun luodaan näkymä ”app/views/features/show.html.erb” väliaikaisella sisällöllä.

Kun näkymä on luotu, ominaisuuden sivu pystytään lataamaan. Tämä saa askeleen menemään läpi. Seuraava askel liittyy poistopainikkeen painamiseen. Koska aikaisemmassa vaiheessa ollaan luotu askelmääritys säännöllisten lausekkeiden avulla, sitä voidaan hyödyntää uudestaan tässä askeleessa, eikä tarvitse kirjoittaa painikkeen painamista varten uutta askelmääritystä. Askel epäonnistuu, koska näkymästä ei löydy painiketta ominaisuuden poistamiselle. Lisätään näkymään rivi, missä lukee ominaisuuden nimi ja painike ominaisuuden poistamista varten.

```
<h2><%= @feature.name %></h2>

<%= button_to "Delete", @feature, :method => :delete %>
```

Koodiesimerkki 39. app/views/features/show.html.erb

Kontrolleriin tulee vielä lisätä toiminto destroy, jonka avulla ominaisuus poistetaan. Toiminnolle pitää myös luoda näkymä, johon laitetaan viesti onnistuneesta ominaisuuden poistamisesta. Viimeinen askel ominaisuuden poistamisessa on, että poistettua ominaisuutta ei tulisi löytyä ominaisuudet-sivulta.

```
Then(/^I should not see "(.*?)"$/) do |text|
  page.should_not have_content(text)
end
```

Koodiesimerkki 40. features/step_definitions/feature_steps.rb

Askel ei kuitenkaan mene vielä läpi, vaan ominaisuudet-sivulta löytyy edelleen ominaisuus, jota yritettiin poistaa. Jotta poistaminen onnistuu, tulee kontrolleriin lisätä destroy-toimintoon ominaisuuden poistaminen. Poisto tapahtuu ActiveRecordin destroy-metodilla.

```
def destroy
  @feature = Feature.find(params[:id])
  @feature.destroy
end
```

Koodiesimerkki 41. app/controllers/features_controller.rb

Tämän jälkeen ominaisuus menee läpi. Käytännössä ominaisuuden poistaminen on edelleen hankalaa, koska sovelluksessa ei ole linkkejä, joiden avulla pääsee ominaisuuden sivulle. Lisätään navigointi-ominaisuuteen tapaus, jossa ominaisuudet-sivulla kaikkien ominaisuuksien nimet ovat linkkejä ominaisuuksien sivuille.

```
Scenario: Navigate to Feature Page
  Given there is a feature called "Post a message"
  And I am on the Features page
  When I click the link for "Post a message"
  Then I should see "Post a message"
```

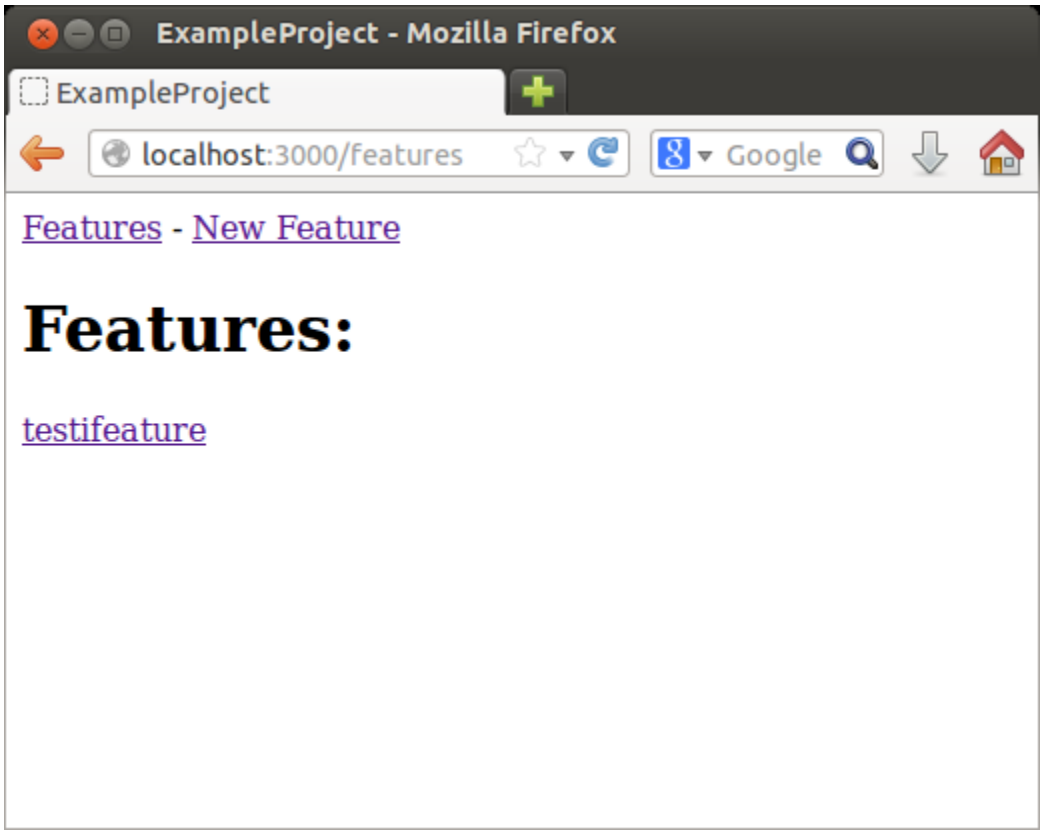
Koodiesimerkki 42. features/links_feature.feature

Cucumberia ajettaessa saadaan selville, että ominaisuudet-sivulta ei löydy haluttua linkkiä, eikä ominaisuus sen takia mene läpi. Jotta ominaisuus menisi läpi, muokataan ominaisuuksien index-näkymää niin, että siellä on linkki jokaiseen ominaisuuteen.

```
<h1>Features:</h1>
<% @features.each do |f| %>
  <p><%= link_to f.name, feature_path(f) %> </p><br />
<% end %>
```

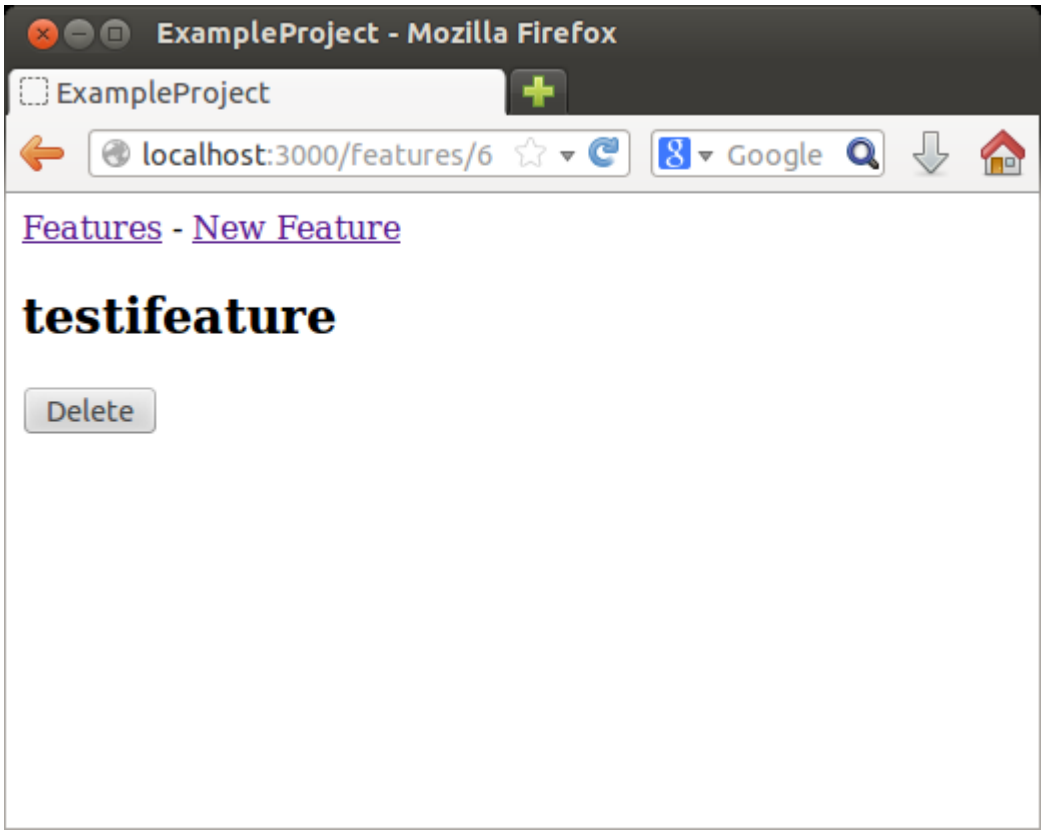
Koodiesimerkki 43. app/views/features/index.html.erb

Tämän jälkeen ominaisuus menee läpi. Sovellus voidaan avata ja todeta, että ominaisuuksien poistaminen oikeasti toimii.



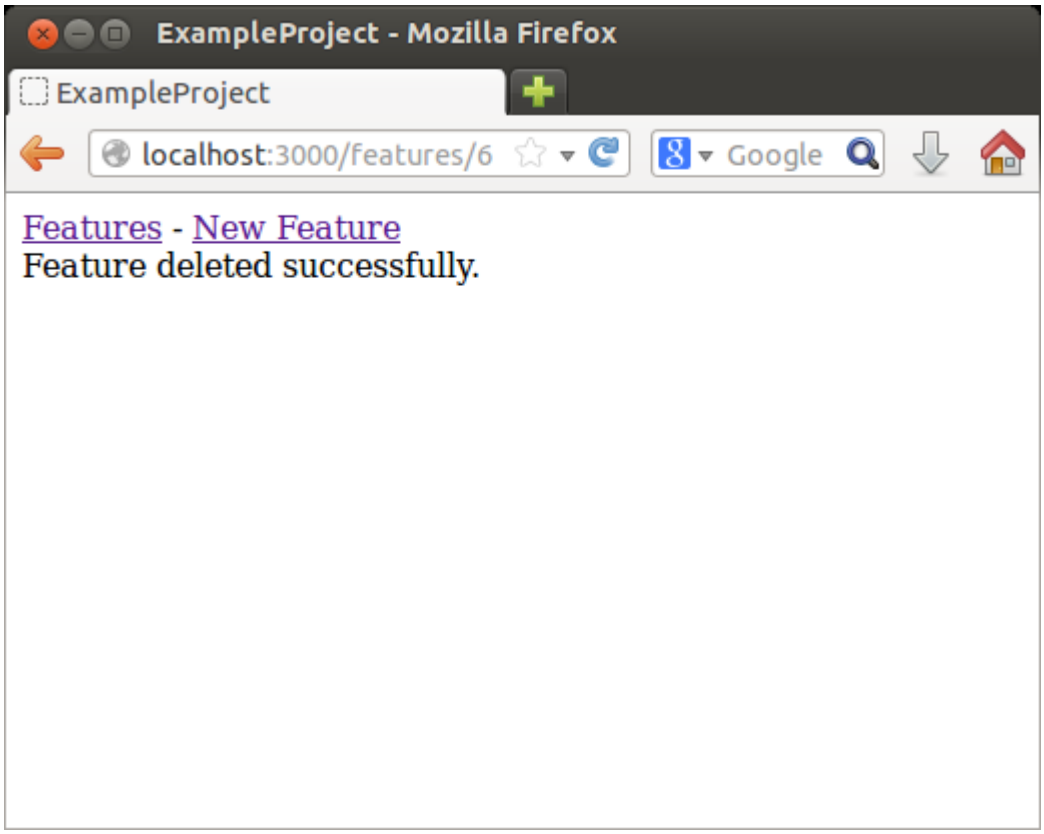
Kuva 23. Ominaisuudet-sivulla olevat ominaisuuksien nimet ovat linkkejä ominaisuuksien sivuille.

Ominaisuudet-sivulla nähdään, että luotujen ominaisuuksien nimet ovat nyt linkkejä, joita painamalla päästään ominaisuuden sivulle.



Kuva 24. Ominaisuuden sivu, missä näkyy painike poistamista varten.

Ominaisuuden omalla sivulla on Delete-painike, jota painamalla ominaisuus pystytään poistamaan.



Kuva 25. Ominaisuuden poistaminen onnistui.

Kun on painettu Delete-painiketta ja ominaisuuden poistaminen on onnistunut, nähdään viesti, joka kertoo onnistuneesta poistamisesta.

Nyt kun ominaisuuden poisto-mahdollisuus on olemassa, voisi sovellusta käyttää To-Do-listan tyyliin pitämään kirjaa vielä toteutettavista ominaisuuksista. Tämä ei kuitenkaan ole kovin tarkka tapa hallita projektia. Ei tiedetä mitään siitä, kuinka paljon mistäkin ominaisuudesta on valmiina ja mitä tulisi tehdä, että kyseessä oleva ominaisuus saadaan valmiiksi.

Seuraavaksi lisätään sovellukseen mahdollisuus lisätä tehtäviä. Ominaisuus koostuu tehtävistä, jotka ovat konkreettisia asioita, mitkä tulee tehdä, että ominaisuus saadaan valmiiksi. Ominaisuuteen liittyvät tehtävät näkyvät ominaisuuden sivulla.

```
Feature: See Tasks
  Scenario: See Tasks for a feature
    Given there is a feature called "Post a message"
    And the feature has task called "Write the code"
    And the feature has task called "Design the database"
    When I visit the feature page
    Then I should see
      """
      Tasks:
      Write the code
      Design the database
      """
```

Koodiesimerkki 44. features/see_tasks.feature

Ensimmäinen askel menee läpi, koska askelmääritys on toteutettu jo aikaisemmassa vaiheessa. Toinen ja kolmas askel sanovat, että ominaisuudella on tehtävät nimeltä ”Write the code” ja ”Design the database”. Näitä askelia varten tehdään askelmääritys, jossa käytetään apuna FactoryGirlin create-metodia. Metodille kerrotaan tehtävän nimi ja mihin ominaisuuteen se kuuluu.

```
Given(/^the feature has task called "(.*?)"$/) do |task|
  FactoryGirl.create(:task, :name => task, :feature => Feature.first)
end
```

Koodiesimerkki 45. features/step_definitions/feature_steps.rb

Kun Cucumber tämän jälkeen ajetaan, kertoo se, että ei ole luotu FactoryGirlin tehdasta tehtäville. Lisätään tehdas tehtäville, jossa kerrotaan, että tehtävä kuuluu ominaisuudelle ja tehtävällä on nimi.

```
require 'factory_girl'

FactoryGirl.define do

  factory :feature do |f|
    f.name 'Post a message'
  end

  factory :task do |f|
    f.association :feature
    f.name 'Write the code'
  end
end
```

Koodiesimerkki 46. features/support/factories.rb

FactoryGirl ei kuitenkaan pysty luomaan tehtävää, koska siltä puuttuu malli. Generoidaan malli komentorivillä Railsin tarjoaman ”generate”-ominaisuuden avulla.

```
arttu@arttu-VirtualBox:~/rails_projects/example_project$ rails g model Task name
:string feature_id:integer
      invoke  active_record
      create    db/migrate/20140402091854_create_tasks.rb
      create    app/models/task.rb
      invoke    rspec
      create      spec/models/task_spec.rb
```

Kuva 26. Tehtävä-mallin generoiminen komentoriviltä

Mallin luomisen lisäksi malliin täytyy lisätä tieto siitä, että tehtävä kuuluu aina ominaisuudelle.

```
class Task < ActiveRecord::Base
  belongs_to :feature
end
```

Koodiesimerkki 47. app/models/task.rb

Myös ominaisuuden mallissa tulee kertoa, että siihen kuuluu useita eri tehtäviä.

```
class Feature < ActiveRecord::Base
  has_many :tasks
end
```

Koodiesimerkki 48. app/models/features.rb

Kun vielä viedään muutokset tietokantaan ja päivitetään testitietokanta ajamalla komentoriviltä ”rake db:migrate db:test:prepare”, saadaan kaksi seuraavaa askelta läpäisemään.

Tässä vaiheessa meillä on yksi määrittelemätön askel, ”When I visit the feature page”, jossa ladataan ominaisuuden sivu. Tämän askelmääritys saadaan jälleen toteutettua Capybaran visit-metodilla. Kun askel on saatu läpäisemään, päästään tilanteeseen, jossa on enää yksi epäonnistuva askel, eli ominaisuuden sivulla ei näy ominaisuuteen kuuluvia tehtäviä. Jotta tehtävät saadaan näkymään, on meidän muokattava ominaisuuden näkymää ”show”-toiminnolle.

```
<h2><%= @feature.name %></h2>
<table>
  <tr>
    <td> Tasks: </td>
  </tr>
  <% @feature.tasks.each do |task| %>
    <tr>
      <td><%= task.name %> </td>
    </tr>
  <% end %>
</table>
<p><%= button_to "Delete", @feature, :method => :delete %></p>
```

Koodiesimerkki 49. app/views/features/show.html.erb

Näkymässä käydään kaikki ominaisuuteen liittyvät tehtävät läpi ja tulostetaan niiden nimi. Tämä näkymä saa viimeisenkin askeleen läpäisemään.

Tehtävien näkyminen ominaisuuden sivulla ei vielä riitä. Ominaisuuksia ei pysty lisäämään työkalun avulla. Se onkin seuraava ominaisuus, joka toteutetaan.

Sovelluksen halutaan, että sovelluksessa on toiminnallisuus uuden tehtävän lisäämiselle. Uuden tehtävän lisäyssivulla on lomake, jossa on kenttä nimeä varten ja painike

lähettämistä varten. Onnistuneen lisäyksen jälkeen ominaisuuden sivulla näkyy siihen kuuluva tehtävä.

```
Feature: Add task
  Scenario: Add a new task
    Given there is a feature called "Post a message"
    And I am on the Add new task page
    When I fill in the name form for task with "Design the database"
    And I click "Submit"
    And I visit the feature page
    Then I should see "Tasks: Design the database"
```

Koodiesimerkki 50. features/add_task.feature

Ensimmäinen toteutettava askel on uuden tehtävän lisäyssivun lataus. Askelmäärityksen toteutuksessa ladataan sivu Capybaran visit-metodin avulla. Ladattavaan sivuun viitataan metodilla ”new_feature_task_path”.

```
Given(/^I am on the Add new task page$/) do
   visit(new_feature_task_path(1))
end
```

Koodiesimerkki 51. features/step_definitions/navigation_steps.rb

Askel epäonnistuu, koska ei ole vielä olemassa metodia “new_feature_task_path”. Jotta kyseinen metodi saadaan, täytyy muuttaa sovelluksen reitityksiä. Luodaan resurssi tehtäville, niin että se on tehtävän aliresurssi.

```
resources :features do
  resources :tasks
end
```

Koodiesimerkki 52. config/routes.rb

Tämän aliresurssin luominen antaa polkuja kuten uuden tehtävän lisäämistä varten “features/:feature_id/tasks/new”. Saadaan myös apumetodeita kuten esimerkiksi askelmäärityksessä käytettävä ”new_feature_task_path”. Seuraavaksi Cucumber kertoo , että sovelluksesta puuttuu tehtäville kontrolleri. Kontrolleriin lisätään tyhjät toiminnot ”index” ja ”new”.

Tämän jälkeen Cucumber valittaa, että näkymä uuden näkymän luomista varten puuttuu. Askel saadaan läpi, kun luodaan tyhjä näkymä ”app/views/tasks/new.html.erb”.

Seuraava askel liittyy lomakkeen nimi-kentän täyttämiseen. Askelmäärityksen toteutus onnistuu Capybaran fill_in-metodilla. Askel epäonnistuu, koska vielä ei ole toteutettu näkymää, jossa olisi lomakekenttä nimeä varten.

Toteutetaan näkymä, jossa on lomake, missä on kenttä nimelle, piilotettu kenttä, mikä sisältää tiedon siitä, mihin ominaisuuteen tehtävä kuuluu ja painike lomakkeen lähettämistä varten.

```
<h1>Add a new task:</h1>

<div class="row">
  <div class="span6 offset3">
    <%= form_for [@feature, @task] do |f| %>

      <%= f.hidden_field :feature_id, :value => @feature.id %>
      <%= f.label :name %>
      <%= f.text_field :name %>

      <%= f.submit "Submit", class: "btn btn-large btn-primary" %>
    <% end %>
  </div>
</div>
```

Koodiesimerkki 53. app/views/tasks/new.html.erb

Kun jälleen ajetaan Cucumber, valittaa se, että ensimmäinen argumentti lomakkeessa ei voi olla nil. Tämä johtuu siitä, että näkymä ei tiedä olioita @feature ja @task. Korjataan tilanne muokkaamalla kontrolleria niin, että toiminnossa ”new” haetaan sen ominaisuuden tiedot muuttujaan @feature, johon luotava tehtävä kuuluu ja luodaan uusi @task-olio tehtävän luomista varten.

```
class TasksController < ApplicationController

  def index
  end

  def new
    @feature = Feature.find(params[:feature_id])
    @task = Task.new
  end

end
```

Koodiesimerkki 54. app/controllers/tasks_controller.rb

Tämän jälkeen lomakkeen nimikentän täyttäminen onnistuu, mutta seuraava askel, lähetyspainikkeen painaminen, epäonnistuu, koska kontrollerissa ei ole toimintoa create, johon ohjaudutaan lomakkeen lähettämisen jälkeen. Tässä vaiheessa luodaan vain tyhjä create-toiminto.

```
def create
end
```

Koodiesimerkki 55. app/controllers/tasks_controller.rb

Tämän jälkeen Cucumber valittaa puuttuvasta create-näkymästä. Näkymään kirjoitetaan viesti onnistuneesta tehtävän luonnista. Tämä saa viimeistä lukuun ottamatta kaikki askeleet menevät läpi. Viimeinen askel on, että ominaisuuden sivulta pitäisi löytyä siihen liittyvät tehtävät. Tämän askeleen läpi saamiseksi täytyy muuttaa kontrolleria siten, että create-toiminnossa tallennetaan lomakkeelle syötetty tehtävä tietokantaan.

```
def create
  @task = Task.new(task_params)
  if @task.save
  else
    render 'new'
  end
end

def task_params
  params.require(:task).permit(:feature_id, :name)
end
```

Koodiesimerkki 56. app/controllers/tasks_controller.rb

Tämä saa viimeisenkin ominaisuuden läpäisemään. Nyt sovelluksessa on mahdollisuus lisätä uusia tehtäviä ominaisuuksille. Ongelmana on kuitenkin, että missään ei ole linkkiä uuden tehtävän lisäyssivulle.

Lisätään tehtävän lisäyksen ominaisuuskuvaukseen tapaus, jossa navigoidaan ominaisuuden sivulta linkkiä painamalla uuden tehtävän lisäyssivulle.

```
Feature: Add task
  Scenario: Navigate to add new task page
    Given there is a feature called "Post a message"
    And I am on the feature page
    And I click the link for "Add new task"
    Then I should see "Add a new task:"
```

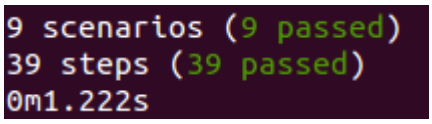
Koodiesimerkki 57. features/add_task.feature

Kaksi ensimmäistä askelta menevät suoraan läpi, mutta kolmas ei. Se epäonnistuu, koska ominaisuuden sivulla ei ole linkkiä ”Add new task”. Lisätään linkki ominaisuuden show-toiminnon näkymään.

```
<h2><%= @feature.name %></h2>
<table>
  <tr>
    <td> Tasks: </td>
  </tr>
  <% @feature.tasks.each do |task| %>
    <tr>
      <td><%= task.name %> </td>
    </tr>
  <% end %>
</table>
<p> <%= link_to "Add new task", new_feature_task_path(@feature) %> </p>
<p><%= button_to "Delete", @feature, :method => :delete %></p>
```

Koodiesimerkki 58. app/views/features/show.html.erb

Tämän jälkeen kaikki 9 tapausta ja niihin kuuluvat 39 askelta menevät läpi.



Kuva 27. Kaikki 9 tapausta menevät läpi

5.5 Projektinhallintatyökalun kehityksen yhteenveto

Nyt on kehitetty yksinkertainen projektinhallintatyökalun, jossa pystyy luomaan ominaisuuksia ja tarkastelemaan niitä. Ominaisuudet vastaavat ominaisuuksia, joita käyttäjä haluaa tuotteeseensa. Ominaisuudet koostuvat tehtävistä, jotka kuvastavat käytännön tehtäviä, joita tulee tehdä, että halutut ominaisuudet saadaan valmiiksi.

Projektinhallintatyökalussa on selviä puutteita ja paljon mahdollisuuksia jatkokehittelyyn. Työkalussa ei pysty pitämään kirjaa tehdyistä työtunneista eikä työkalussa myöskään käy selville missä tilassa mikäkin ominaisuus on. Työtehtäviä ei myöskään pysty mitenkään osoittamaan tietyille henkilöille.

Kehityksessä on käytetty käyttäytymislähtöisen kehityksen periaatteita ja nojattu Cucumber-nimiseen käyttäytymislähtöisen kehityksen työkaluun. Cucumber ohjasi koko ajan kehitystä oikeaan suuntaan kertomalla, mitkä askeleet menevät läpi ja mitkä epäonnistuvat (ja mistä syystä epäonnistuvat). Cucumberin käytön ansiosta on saatu suuri määrä toiminnallisia testejä. Testit toimivat myös elävänä dokumentaationa sovelluk-

sen toiminnasta. Jos uusi kehittäjä haluaa saada selville, mitä sovellus tekee, saa hän sen selville lukemalla ominaisuuskuvaukset. Jatkokehitystä varten on myös kohtalaisen suuri määrä erilaisia askelmäärityksiä, joita voi käyttää apuna uusissa ominaisuuskuvauksissa.

Tuotteen kehittäminen tapahtui käytännössä niin, että ensin määriteltiin haluttu ominaisuus ja kirjoitettiin esimerkki ominaisuuden toiminnasta Cucumberin ominaisuuskuvauksen avulla. Tämän jälkeen ajettiin Cucumber ja nähtiin, mitä askelia tulee toteuttaa. Ensimmäiselle askeleelle toteutettiin askelmääritys Capybaran, ja joissain tapauksissa myös RSpecin, avulla. Askelmäärityksen toteuttaminen sai askeleen epäonnistumaan. Kun askel epäonnistui, halutut toiminnallisuudet kirjoitettiin sovellukseen niin, että askel menee läpi. Tätä sykliä toistettiin, kunnes kaikki halutut ominaisuudet menivät Cucumberissa läpi. Kehitettävän tuotteen luonteesta riippuen voi olla järkevää lisätä tähän kehityssykliin myös yksikkötestejä, jolloin yksikkötestit kirjoitettaisiin heti askelmäärityksen kirjoittamisen jälkeen. Tässä tapauksessa kirjoitetaan koodia niin paljon, että yksikkötestit menevät läpi, ja kun kaikki yksikkötestit menevät läpi, pitäisi myös askeleen läpäistä.

Kun työkalua kehitettiin Cucumberin avulla, se ohjasi koko ajan kehitystä oikeaan suuntaan. Käytännössä Cucumber melkein jopa kertoi, mitä tulee tehdä seuraavaksi, että saadaan askel läpi. Tekniikkaa, jossa tehdään muutos ja katsotaan mikä hajoaa, kutsutaan kääntäjään nojaamiseksi (engl. leaning on the compiler). Tässä tapauksessa Cucumber kertoi, mikä osa ei toimi. Tästä pystyi lähes joka kerta päättelemään, mitä tulee tehdä seuraavaksi. Cucumberin ajaminen komentoriviltä antaa myös hyvän kuvan siitä, mitä tuotteessa on jo kehitetty valmiiksi ja mitkä toiminnallisuudet ovat mahdollisesti kesken. Tästä on apua, jos tuotetta kehittää useampi kuin yksi henkilö.

6 Yhteenveto

Opinnäytetyön tavoitteena oli tutustua käyttäytymislähtöiseen kehitykseen, siihen tarkoitettuihin työkaluihin ja kuinka sitä voidaan hyödyntää käytännön sovelluskehityksessä. Työssä käytiin läpi testivetoisia kehitysmenetelmiä ja tutustuttiin niiden perusperiaatteisiin. Niiden huomattiin olevan hyvin samankaltaisia ja esimerkiksi hyväksymistestivetoisen ja käyttäytymislähtöisen kehityksen erot ovat pieniä ja liittyvät lähinnä sanastoon ja tapaan ajatella.

Työn loppuosassa käytiin läpi, miltä käyttäytymislähtöinen kehitys voi näyttää käytännössä. Tämä tehtiin kehittämällä yksinkertainen projektinhallintatyökalu. Kehityksessä painotettiin sitä, että nähdään käyttäytymislähtöiselle kehitykselle ominaisia asioita, kuten esimerkiksi ”ominaisuuskuvaus-askelmääritys-toiminnallisuuden toteutus” -sykli. Nähtiin myös, kuinka Cucumberiin nojaamista voidaan käyttää ohjaamaan kehitysprosessia eteenpäin. Kehitysprojektissa nähtiin, että Cucumber toimii Ruby on Rails-ohjelmistokehyksen kanssa saumattomasti.

Käyttäytymislähtöisen kehityksen opiskelu oli hankalaa useammasta syystä. Yksi suurimpia syitä on se, että käyttäytymislähtöinen kehitys ei ole tarkasti määriteltyä, vaan lähinnä tapa ajatella ohjelmistokehitystä eri näkökulmasta. Myös hyvien ja kattavien lähdemateriaalien puute vaikeutti BDD:n opiskelua. BDD:stä kertovat materiaalit koostuvat lähinnä käyttäytymislähtöisen kehityksen pioneerien blogeista, eikä The Cucumber Bookin lisäksi aiheesta ole kirjoitettu hyviä kirjoja.

Työn painopisteenä on ollut käyttäytymislähtöinen kehitys Cucumberin avulla. Tämä valinta tehtiin siitä syystä, että se on hyvin dokumentoitu ja se toimii hyvin yhteen Ruby on Rails -ohjelmistokehyksen kanssa.

Lähteet

1 The Twelve Principles of Agile Software. Verkkodokumentti. Agile Alliance. <http://www.agilealliance.org/the-alliance/the-agile-manifesto/the-twelve-principles-of-agile-software/>.

2 Ketterän ohjelmistokehityksen julistus. Verkkodokumentti. <http://agilemanifesto.org/iso/fi/>.

3 Ken Schwaber ja Jeff Sutherland.The Scrum Guide: Scrumin määritelmä ja pelisäännöt. Verkkodokumentti. <https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/Scrum%20Guide%20-%20FI.pdf#zoom=100>.

4 The Rules of Extreme Programming. Verkkodokumentti. Extreme Programming: Don Wells. <http://www.extremeprogramming.org/rules.html>.

5 Introduction to Test Driven Development (TDD). Verkkodokumentti. <http://www.agiledata.org/essays/tdd.html>.

6 Gojko Adzic. 2011. Specification by Example.

7 Ville Harvala. 2009. Hyväksymistestivetoinen ohjelmistokehitys: Hyödyt ja haasteet empiirisen tutkimuksen valossa.

8 Neel Lakshminarayan. BDD is more than “TDD done right”. Verkkodokumentti. <http://neelnarayan.blogspot.fi/2010/07/bdd-is-more-than-tdd-done-right.html>.

9 David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, Dan North. 2010. The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends.

10 Dan North. Introducing BDD. Verkkodokumentti. <http://dannorth.net/introducing-bdd/>.

11 Cucumber Wiki. Verkkodokumentti. <https://github.com/cucumber/cucumber/wiki>.

12 JBehave.org. Verkkodokumentti. <http://jbehave.org/reference/stable/>.

13 5 Whys. Verkkodokumentti. Wikipedia. <http://en.wikipedia.org/wiki/5_Whys>.

14 Feature Injection: Putting the Value First In Your User Stories. James Carr. <http://blog.james-carr.org/2009/10/02/feature-injection-putting-the-value-first-in-your-user-stories/>.

15 Guide to Agile Practices. Verkkodokumentti. Agile Alliance. <http://guide.agilealliance.org/>.

16 Behavior-Driven Development. Verkkodokumentti. Wikipedia. <http://en.wikipedia.org/wiki/Behavior-driven_development>.

17 Matt Wynne ja Aslak Hellesøy. 2012. The Cucumber Book.

18 Cucumber i18n. Verkkodokumentti. <https://github.com/cucumber/gherkin/blob/master/lib/gherkin/i18n.json>.

19 Aslak Hellesøy. Cucumber-jvm 1.0.0. Verkkodokumentti. <http://aslakhellesoy.com/post/20006051268/cucumber-jvm-1-0-0>.

20 Quick Intro to Behat. Verkkodokumentti. <http://docs.behat.org/quick_intro.html>.

21 MINK Web acceptance testing. Verkkodokumentti. <http://mink.behat.org/>.

22 Kent Beck. 2003. Test Driven Development: By Example.

23 Fit: Framework for Integrated Test. Verkkodokumentti. <http://fit.c2.com/>.

24 SeleniumHQ Browser Automation. Verkkodokumentti. <http://docs.seleniumhq.org/>.

25 Ruby on Rails. Verkkodokumentti. <http://rubyonrails.org/>.

26 Ruby on Rails. Verkkodokumentti. Wikipedia. <http://en.wikipedia.org/wiki/Ruby_on_Rails>.

27 Model-view-controller. Verkkodokumentti. Wikipedia. <http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>.

28 Capybara: Acceptance test framework for web applications. Verkkodokumentti. <https://github.com/jnicklas/capybara>.

29 FactoryGirl: A library for setting up Ruby objects as test data. Verkkodokumentti. <https://github.com/thoughtbot/factory_girl>.

Projektinhallintatyökalun ominaisuudet

Cucumber Features

9 scenarios (9 passed)
39 steps (39 passed)
Finished in **0m0.979s seconds**
Collapse All Expand All

Feature: Add features

Scenario: Add new feature	features/add_feature.feature:2
Given I am on the create new feature page	features/step_definitions/navigation_steps.rb:5
When I fill in the name field for the feature form with "**Find users**"	features/step_definitions/feature_steps.rb:13
And I click "**Submit**"	features/step_definitions/feature_steps.rb:17
And I visit the features page	features/step_definitions/navigation_steps.rb:1
Then I should see "**Find users**"	features/step_definitions/feature_steps.rb:9

Feature: Add task

Scenario: Navigate to add new task page	features/add_task.feature:2
Given there is a feature called "**Post a message**"	features/step_definitions/feature_steps.rb:1
And I am on the feature page	features/step_definitions/navigation_steps.rb:17
And I click the link for "**Add new task**"	features/step_definitions/navigation_steps.rb:13
Then I should see "**Add a new task:**"	features/step_definitions/feature_steps.rb:9
Scenario: Add a new task	features/add_task.feature:8
Given there is a feature called "**Post a message**"	features/step_definitions/feature_steps.rb:1
And I am on the Add new task page	features/step_definitions/navigation_steps.rb:25
When I fill in the name form for task with "**Design the database**"	features/step_definitions/task_steps.rb:1
And I click "**Submit**"	features/step_definitions/feature_steps.rb:17
And I visit the feature page	features/step_definitions/navigation_steps.rb:21
Then I should see "**Tasks: Design the database**"	features/step_definitions/feature_steps.rb:9

Feature: Delete features

Scenario: Delete feature	features/delete_feature.feature:2
Given there is a feature called "**Find users**"	features/step_definitions/feature_steps.rb:1
And I am on the feature page	features/step_definitions/navigation_steps.rb:17
And I click "**Delete**"	features/step_definitions/feature_steps.rb:17
And I visit the features page	features/step_definitions/navigation_steps.rb:1
Then I should not see "**Find users**"	features/step_definitions/feature_steps.rb:21

Feature: Navigate

Scenario: Navigate to New Feature Page	features/links.feature:2
Given I am on the Features page	features/step_definitions/navigation_steps.rb:9
When I click the link for "**New Feature**"	features/step_definitions/navigation_steps.rb:13
Then I should see "**Add new feature**"	features/step_definitions/feature_steps.rb:9
Scenario: Navigate to Features Page	features/links.feature:7
Given I am on the create new feature page	features/step_definitions/navigation_steps.rb:5
When I click the link for "**Features**"	features/step_definitions/navigation_steps.rb:13
Then I should see "**Features:**"	features/step_definitions/feature_steps.rb:9
Scenario: Navigate to Feature Page	features/links.feature:12
Given there is a feature called "**Post a message**"	features/step_definitions/feature_steps.rb:1
And I am on the Features page	features/step_definitions/navigation_steps.rb:9
When I click the link for "**Post a message**"	features/step_definitions/navigation_steps.rb:13
Then I should see "**Post a message**"	features/step_definitions/feature_steps.rb:9

Feature: See Features

Scenario: See all the features	features/see_features.feature:2
Given there is a feature called "**Post a message**"	features/step_definitions/feature_steps.rb:1
And there is a feature called "**See messages**"	features/step_definitions/feature_steps.rb:1
When I visit the features page	features/step_definitions/navigation_steps.rb:1
Then I should see	features/step_definitions/feature_steps.rb:5

```
Post a message
See messages
```

Feature: See Tasks

Scenario: See Tasks for a feature	features/see_tasks.feature:2
Given there is a feature called "**Post a message**"	features/step_definitions/feature_steps.rb:1
And the feature has task called "**Write the code**"	features/step_definitions/feature_steps.rb:25
And the feature has task called "**Design the database**"	features/step_definitions/feature_steps.rb:25
When I visit the feature page	features/step_definitions/navigation_steps.rb:21
Then I should see	features/step_definitions/feature_steps.rb:5

```
Tasks:
Write the code
Design the database
```