Skenaariokuvausten automatisonnin vaikutus käyttäytymisohjatussa ohjelmistokehityksessä

Timo Mikkilä

Pro gradu -tutkielma

ITÄ-SUOMEN YLIOPISTO

Tietojenkäsittelytieteen laitos

Tietojenkäsittelytiede

Kesäkuu 2018

ITÄ-SUOMEN YLIOPISTO, Luonnontieteiden ja metsätieteiden tiedekunta, Kuopio
Tietojenkäsittelytieteen laitos
Tietojenkäsittelytiede

Mikkilä, Timo Hannu: Skenaariokuvausten automatisonnin vaikutus käyttäytymisohjatussa ohjelmistokehityksessä
Pro gradu -tutkielma, 48 s.
Pro gradu -tutkielman ohjaaja: FT Virpi Hotti
Kesäkuu 2018

Tiivistelmä: Käyttäytymisohjattu kehitys (behavior-driven development, BDD) kuvaa menetelmät ohjelmiston määrittämiseen skenaarioilla. Tutkielmassa selvitettiin, miten skenaariokuvausten automatisointi vaikuttaa käytäytymisohjattuun ohjelmistokehitykseen. Lisäksi arvioitiin skenaariokuvausten automatisoinnin vaikutusta määrittelyihin, hyväksymis- ja regressiotestaukseen sekä yksikkötason kehitykseen ja testaukseen.

Skenaario on esimerkinomainen kuvaus siitä, miten ohjelmisto käyttäytyy tietyssä tilanteessa. Skenaariokuvauksia voidaan hyödyntää hyväksymistestivetoisen kehityksen (acceptance test-driven development, ATDD) tapaan kehityksessä - skenaariokuvaus muunnetaan automaattiseksi testiksi ja sen pohjalta tehdään toteutus ja testaus. Skenaarioiden automatisointi vaatii, että skenaariot kirjoitetaan määrittelyvaiheessa sellaiseen muotoon, että automatisointityökalut voivat tulkita skenaariokuvauksia. Automatisoidut skenaariokuvaukset muodostavat kattavan testiverkon, jota voidaan tehokkaasti hyödyntää ohjelmiston yksikkö-, hyväksymis- ja regressiotestauksessa.

Avainsanat: BDD, käyttäytymisohjattu kehitys, skenaario, TDD, testiohajttu kehitys, ATDD, hyväksymistestiohjattu kehitys, ketterät menetelmät

ACM-luokat (ACM Computing Classification System, 1998 version): D.2.1, D.2.5

UNIVERSITY OF EASTERN FINLAND, Faculty of Science and Forestry, Kuopio
School of Computing
Computer Science

Mikkilä, Timo Hannu: Effect of automated scenarios on behavior-driven development
Master's Thesis, 48 p.
Supervisors of the Master's Thesis: PhD Virpi Hotti
June 2018

Abstract: Behavior-driven development (BDD) describes means to define a software using scenarios. In this thesis it was studied how automated scenarios affect on behavior-driven software development. Also, the effect of the automated scenarios was evaluated for definitions, acceptance and regression testing and for unit development and testing phases.

A scenario is an example-like description of how software should behave in certain situation. Scenarios can be utilized on the same way how tests are utilized on acceptance test-driven development (ATDD) by automating them and making the implementation based on the automated scenarios. For scenarios to be automatable they must be written in certain format on the planning phase so that the automation tools can properly interpret them. Automated scenarios form an extensive net of tests which can be effectively utilized for the unit, acceptance and regression testing of the software.

Keywords: BDD, Behavior-driven development, scenario, TDD, test-driven development, ATDD, acceptance test-driven development, agile methods

CR Categories (ACM Computing Classification System, 1998 version): D.2.1, D.2.5

Esipuhe

Aikansa se otti, mutta tässä se nyt on. Suuret kiitokset teille kaikille, joilla riitti uskoa ja jotka jaksoitte kärsivällisesti kannustaa minua loppuun asti.

Sisältö

1 Johdanto

Käyttäytymisohjattu kehitys, englanniksi behavior-driven development tai lyhyesti BDD, on alun perin Dan Northin esittelemä ohjelmistokehitysmenetelmä. Menetelmä syntyi, kun North etsi tapoja, joiden avulla opiskelijoiden olisi helpompi omaksua yksikkötestiohjattu kehitys (Test-driven development, TDD). Yksikkötestiohjatussa kehityksessä ennen toteutusta tehdään testi, joka kuvaa haluttua toiminnallisuutta ja varsinainen toteutus tehdään vasta testin jälkeen. Aluksi North yritti siirtyä pois testi-sanan käytöstä kehittämällä uuden nimeämiskäytännön yksikkötesteille. Tässä käytännössä testien nimet aloitetaan “should” eli “pitäisi” -sanalla testi-sanan asemasta. Lisäksi North ohjeistaa, että nimien tulisi kuvata kokonaisella lauseella, mitä testattavan ohjelman osan tulisi tehdä tai kuinka sen tulisi käyttäytyä. Kuvaavat nimet auttavat myös siinä vaiheessa, kun testi jostain syystä epäonnistuu. Tällöin jo testin nimestä näkee, mitä ominaisuutta ongelma koskee ja sen mukaan voidaan päätellä, onko vika testissä vai testattavassa ohjelmassa. (North 2006)

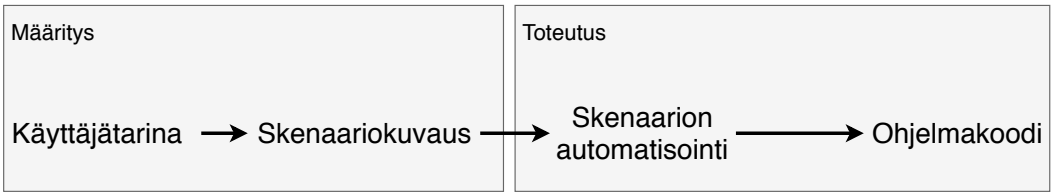
Myöhemmin North havaitsi, että samaa mallia voidaan soveltaa myös hyväksymistestiohjatussa kehityksessä (Acceptance test-driven development, ATDD), joka on yksikkötestiohjattua kehitystä korkeammalla abstraktiotasolla toimiva testiohjattu kehitysmalli. Hän kehitti yhdessä Chris Mattsin kanssa rakenteellisen kielen, jota voidaan käyttää vaatimusten määrityksessä. Kieli mahdollistaa kehitettävää ohjelmistoa koskevien esimerkkiskenaarioiden kuvaamisen ja odotetun lopputuleman määrittämisen. Nämä määritykset soveltuvat esimerkiksi ketterissä menetelmissä käytettyjen käyttäjätarinoiden tarkentamiseen. Sellaisenaan skenaariokuvauksia voidaan käyttää ihmisen suorittamiin testeihin. Rakenteellinen muoto kuitenkin mahdollistaa sen, että tietokoneen on helppo tulkita niitä, jolloin skenaariokuvauksista voidaan edelleen tehdä automaattisesti suoritettavia hyväksymistestejä. (North 2006)

Sen lisäksi, että käyttäytymisohjattu kehitys antaa välineet toteuttaa hyväksymis- ja yksikkötestiohjattua kehitystä, pyrkii se myös parantamaan vaatimusmäärittelyä. Se pyrkii auttamaan niiden ominaisuuksien löytämisessä, joilla on suurin liiketoiminnallinen arvo (Smart 2015, s. 16). Lisäksi se kannustaa ohjelmiston määritysten laatimiseen yhdessä asiakkaan, testaajan ja kehittäjän kesken (Smart 2015, s. 17). Määrityksinä käytetään selkeitä ja uskottavia esimerkkejä, jotka kuvaavat todellisia tilanteita (Smart 2015, s. 19). Lopuksi nämä määritykset voidaan hyödyntää tehokkaasti hyväksymistestauksessa ja sen jälkeen toistuvissa regressiotesteissä (Smart 2015, s. 21). Käyttäytymisoh-

jattu kehitys mahdollistaa myös tehokkaan, lähes reaaliaikaisen edistymisen seurannan ja teknisen dokumentaation, joka elää muutosten mukaan (Smart 2015, s. 26). Käyttäytymisohjattu kehitys on ohjelmistokehityksessä mukana niin vaatimusmäärittelyssä kuin yksikkö- ja hyväksymistestauksessa.

Käyttäytymisohjattu kehitys tunnetaan myös monilla muilla nimillä, kuten "esimerkeillä määrittäminen" (Specification by Example). Tätä termiä käyttää Gojko Adzic samannimisessä kirjassaan (Adzic 2011). Myös esimerkeillä määritys pitää käyttäytymisohjatun kehityksen tapaan sisällään esimerkkien hyödyntämisen vaatimusmäärittelyssä ja esimerkkien muuntamisen suoritettavaan muotoon automaattisiksi testeiksi (Adzic 2011, s. 26-27). Kolmas käytetty nimi on "hyväksymistestiohjattu kehitys", johon Lasse Koskela paneutuu kirjassaan Test Driven (Koskela 2008).

Skenaariokuvaukset ovat keskeisessä asemassa käyttäytymisohjatussa kehityksessä. Ne ovat esimerkkejä, joita ihminen manuaalisesti tai tietokone automatisoidusti voivat suorittaa testauksessa testien tapaan. Tämän tutkielman tavoitteena on selvittää, miten skenaariokuvausten automatisointi vaikuttaa käyttäytymisohjattuun ohjelmistokehitykseen. Manuaalisesti suoritettavia skenaarioita ei käsitellä. Tutkielmassa arvioidaan skenaariokuvausten automatisoinnin vaikutusta määrittelyihin, hyväksymis- ja regressiotestaukseen sekä yksikkötason kehitykseen ja testaukseen.

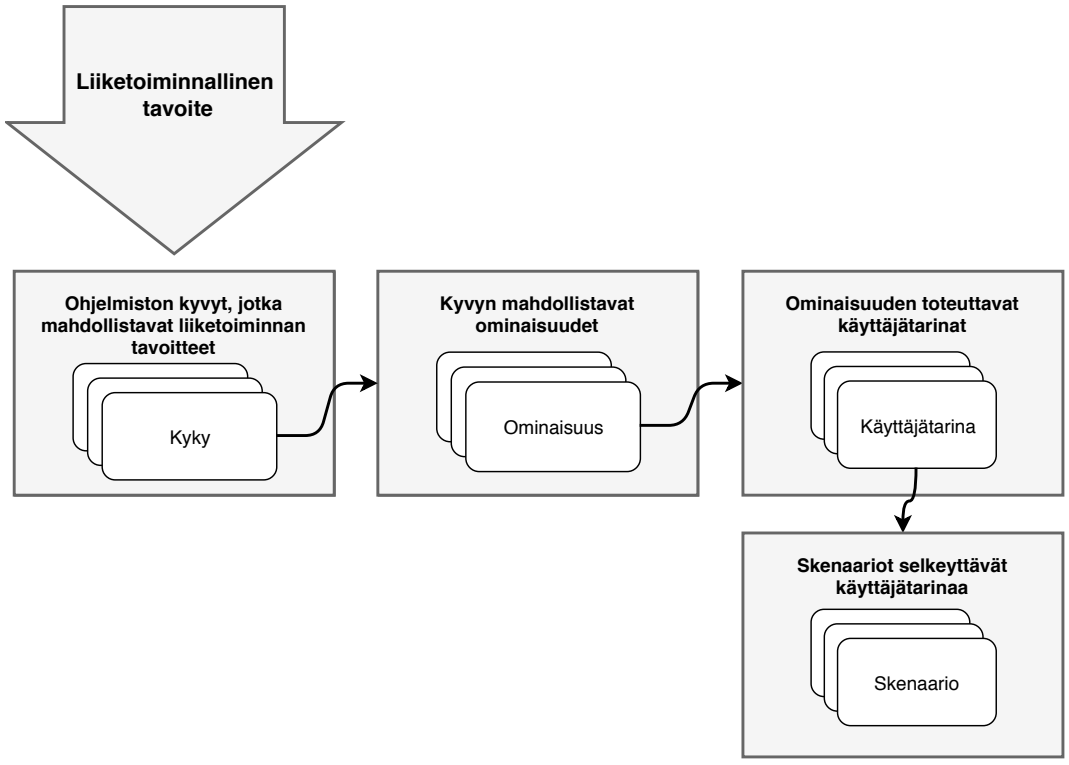


Kuva 1: Käyttäytymisohjatun kehityksen vaiheet.

Käyttäytymisohjattu kehitys voidaan jakaa neljään eri vaiheeseen kuvan 1 mukaisesti. Ensimmäinen vaihe on käyttäjätarinoiden määritys. Käyttäjätarinoita tarkennetaan edelleen skenaariokuvauksilla. Näihin kahteen vaiheeseen tutustutaan luvussa 2. Seuraavat kuvan esittämät vaiheet ovat skenaarioiden automatisointi ja halutun käyttäytymisen toteutus ohjelmistoon. Näihin vaiheisiin tutustutaan luvussa 3. Lisäksi luvussa 4 tarkastellaan käyttäytymisohjatun kehityksen hyödyntämistä yksikkötasolla. Luvussa 5 arvioidaan, kuinka skenaariokuvausten automatisointi vaikuttaa käyttäytymisohjattuun ohjelmistokehitykseen ja luvussa 6 ovat yhteenveto ja pohdinta.

2 Käyttäjätarinat käyttäytymisohjatussa kehityksessä

Skenaariokuvaukset muodostavat käyttäytymisohjatun kehityksen tukirangan: niiden avulla kuvataan ohjelmiston käyttäytymistä. Ne toimivat myös kommunikaatiovälineenä, jonka avulla eri osapuolet pääsevät yhteisymmärrykseen ohjelmistolle asetetuista toiminnallisista vaatimuksista. Ketterissä menetelmissä skenaariokuvaukset ovat myös väline tarkentaa käyttäjätarinoita. Ominaisuuden valmistuttua niitä ei kuitenkaan heitetä käyttäjätarinoiden tapaan pois vaan hyödynnetään testeinä.



Kuva 2: Liiketoiminnan tavoitteet jalostetaan käyttäjätarinoiksi, mukaillen (Smart 2015, s. 90).

Kuva 2 esittää, kuinka ohjelmistoprojektissa lähdetään liikkeelle liiketoiminnan tavoitteista, joita konkretisoidaan ja tarkennetaan vaiheittain. Ensin määritellään ohjelmiston kyvyt, jotka tukevat asetettuja liiketoiminnallisia tavoitteita. Ohjelmisto mahdollistaa nämä kyvyt ominaisuuksien muodossa. Ominaisuudet ovat kuitenkin usein vielä liian suuria kokonaisuuksia toteutettavaksi sellaisenaan, joten ne jaetaan edelleen pienempiin osiin. Ketterissä menetelmissä tähän käytetään käyttäjätarinoita. Ominaisuuksia ja käyttäjätarinoita tarkastellaan lisää luvussa 2.1.

Lähtökohtaisesti käyttäjätarinat sisältävät usein hyvin vähän tietoa. Kehitystyön ede-

tessä käyttäjätarinoita kuitenkin tarkennetaan jatkuvasti tarpeen mukaan. Käyttäytymisohjattu kehitys tuo tähän avuksi skenaariokuvaukset, jotka esimerkinomaisesti kuvaavat vuorovaikutuksen ohjelmiston kanssa ja ilmaisevat myös odotetun tuloksen. Skenaariokuvauksia tarkastellaan luvussa 2.2. Skenaariokuvaukset ovat myös tärkeä elementti kommunikaatiossa ja niitä laativat yhdessä asiakkaan edustaja, testaaja ja kehittäjä. Kommunikaatiota käyttäytymisohjatussa kehityksessä tarkastellaan tarkemmin luvussa 2.3.

2.1 Ominaisuudet ja käyttäjätarinat

Toteutuksen näkökulmasta ominaisuus voidaan tulkita ohjelmiston osaksi, joka täyttää jonkin liiketoiminnallisen tarpeen. Ominaisuuksien olisi hyvä olla riippumattomia toisistaan, jotta ne olisivat testattavissa ja julkaistavissa toisistaan erillään. Tarvittaessa joitain ominaisuuksia voidaan myös jättää pois lopullisesta julkaistavasta ohjelmistotuotteesta ilman suuria muutoksia muihin ominaisuuksiin. (Smart 2015, s. 93-94)

Ominaisuudet ovat usein liian laajoja kokonaisuuksia sellaisenaan toteutettaviksi, jonka vuoksi ne on pilkottava pienempiin osiin eli käyttäjätarinoihin. Myös käyttäjätarinoita voidaan pilkkoa siten, että niistä saadaan riittävän pieniä kokonaisuuksia. Käyttäjätarinat ovat työväline, jonka avulla ominaisuus saadaan toteutettua. Siinä missä ominaisuus jää toteutuksen jälkeen osaksi ohjelmistoa, käyttäjätarinat ovat väliaikaisia ja ne usein hylätään niiden toteuttamisen jälkeen (Leffingwell 2010, s. 102). Loppukäyttäjää kiinnostavat ominaisuudet, eivät niinkään tavat, joilla ohjelmisto on toteutettu. (Smart 2015, s. 98-99)

Käyttäjätarinoista saadaan suurin hyöty silloin, kun tarkkaa määritystä ei tehdä liian aikaisin, vaan vasta tarpeen mukaan (Wiegers ja Beatty 2013, s. 164). Näin ollen, kun uusi käyttäjätarina luodaan, ei siihen vielä välttämättä lisätä lyhyen kuvauksen lisäksi muuta tietoa. Alussa käyttäjätarinoiden tehtävänä on toimia muistuttajina siitä, mitä seikkoja on myöhemmin otettava huomioon. Täten riittää, että ne sisältävät vain sen verran tietoa, että ymmärretään mitä toiminnallisuutta ne koskevat, jotta ne osataan priorisoida toistensa suhteen. Varsinainen tarkennus tehdään usein vasta kehitysiteraatioiden alussa ja kehityksen aikana. Tällöin käyttäjätarinoihin lisätään tarvittavia tarkennuksia, erikoistapauksia ja hyväksymiskriteerejä. (Wiegers ja Beatty 2013, s. 199-200)

2.2 Käyttäjätarinoiden tarkentaminen

Käyttäjätarinoita tarkennetaan yksinkertaisesti lisäämällä tietoa käyttäjätarinan kuvaukseen. Usein käyttäjätarinoihin myös lisätään hyväksymis- tai hylkäyskriteerejä. Hyväksymiskriteerit (acceptance criteria) antavat selkeät ehdot, jotka toteutuksen on täytettävä ennen kuin käyttäjätarina voidaan katsoa toteutetuksi. Hylkäyskriteeri vastaavasti antaa ehdot, joiden täyttyminen tarkoittaa, että toteutus ei vielä vastaa käyttäjätarinan kuvausta. Seuraava esimerkki kuvaa, kuinka hyväksymiskriteerejä voidaan kirjoittaa. (Leffingwell 2010, s. 104; Wiegers ja Beatty 2013, s. 348-349)

```
1: Kuluttajana haluan nähdä päivittäisen energiankulutukseni, jotta voin vähentää
       ↪ kulutustani ja kustannuksiani.
2: Hyväksymiskriteeri:
3: - Lue sähkönkulutusmittarin tiedot 10 sekunnin välein.
4: - Näytä luetut sähkönkulutustiedot portaalissa 15 minuutin ajalle
5: - Näytä viimeisin kulutustieto kotinäytössä
6: - Ei vielä monen päivän trendejä (toinen tarina)
```

Hyväksymis- ja hylkäyskriteerit ovat sen verran epämääräisiä, että niitä ei voida käyttää testeinä. Ne ovatkin ennemmin toteutuksen ja testauksen tukena olevia ehtoja. Kriteereistä voidaan jalostaa edelleen toiminnallisia testejä, joilla tarinan toiminnallisuudelle voidaan tehdä hyväksymistestaus. (Leffingwell 2010, s. 104)

Käyttäytymisohjatussa kehityksessä käytetään käyttäjätarinoiden tarkemmassa määrityksessä skenaariokuvauksia. Skenaariot ovat konkreettisia esimerkkejä, jotka kuvaavat, kuinka käyttäjätarinan kuvaama ominaisuus toimii tietyssä tilanteessa. Ajatuksena on, että käytettäessä konkreettisia esimerkkejä määrityksessä, saadaan tehokkaasti varmistettua, että kaikilla osapuolilla on yhtenäinen käsitys käyttäjätarinan sisällöstä. Tällä tavalla vältytään jo varhain väärinkäsityksiltä. (Smart 2015, s. 38)

Skenaarioiden tulisi olla tarkkoja ja yksiselitteisiä. Epäselviä määreitä, kuten "alle 10", tulisi välttää. Skenaarioiden tulisi myös olla mahdollisimman todenmukaisia. Liian yksinkertaistetut skenaariot eivät vastaa tarpeeksi todellista tilannetta ja näin ollen toteutuksessa ei osata ottaa kaikkia tarpeellisia asioita huomioon. (Adzic 2011, s. 99-100)

Skenaariokuvauksissa ei ole hyvä ottaa kantaa toteutukseen eli kuinka ohjelman tulisi toimia. Sen sijaan niiden tulisi kuvata, mitä ohjelmistolla liiketoiminnan näkökulmasta halutaan saavuttaa. Näin kehittäjille annetaan vapaus etsiä paras mahdollinen toteutustapa, jota voidaan myöhemmin edelleen parantaa muuttamatta skenaariokuvausta.

(Adzic 2011, s. 121)

Skenaariot voidaan kirjata ylös parhaaksi katsotulla tavalla. Yleisesti käytetty tapa on käyttää niin sanottua Given-When-Then -rakennetta tai suomeksi Oletetaan-Kun-Niin -rakennetta. Tässä muodossa skenaario koostuu allekkaisista lauseista, jotka alkavat yleensä jollakin seuraavista sanoista: `Oletetaan` (Given), `Kun` (When), `Niin` (Then) tai `Ja` (And). Seuraava yksinkertainen esimerkki skenaariosta kuvaa, kuinka näitä avainsanoja voidaan käyttää.

```
1: Tapaus: Kun asiakkaana ostan tuotteen, haluan suuremmalla rahamäärällä tehdystä
       ↪ maksusuorituksesta vaihtorahat, jotta minun ei tarvitse maksaa
       ↪ tasarahalla.
2: Oletetaan että tuote maksaa 13,50€
3: Kun asiakas antaa 20€
4: Niin annetaan takaisin asiakkaalle 6,50€
```

`Oletetaan`-avainsanalla alkavalla lauseella kuvataan tapauksen alkutilaa ja asiayhteyttä. Esimerkiksi `Oletetaan, että tuote maksaa 13,50€`. `Kun`-avainsanalla alkavalla lauseella kuvataan jotakin toimintaa, josta tapauksessa ollaan kiinnostuneita. Esimerkiksi `Kun asiakas antaa 20€`. `Niin`-avainsanalla alkavalla lauseella kuvataan odotettu lopputulos esimerkiksi `Niin annetaan takaisin asiakkaalle 6,50€`. `Ja`-avainsanalla voidaan korvata toistuvia perättäisiä `oletetaan`, `kun` ja `niin` -avainsanoja, jotta skenaarion kielestä saataisiin luonnollisempaa. Esimerkiksi kun `Ja`-lause on `Kun`-lauseen jälkeen käsitetään sekin `Kun`-lauseena. (Smart 2015, s. 122)

Seuraavassa esimerkissä kuvataan yksinkertainen skenaario, jossa käyttäjälle valitaan sopiva linja-autoyhteys lähtöajan mukaan.

```
1: Oletetaan että linja-autoja kulkee keskustasta sairaalalle 7:58, 8:02, 8:08,
       ↪ 8:11
2: Kun haluan lähteä klo 8:00
3: Niin minulle tulisi ehdottaa klo 8:02 lähtevää vuoroa
```

Aluksi `Oletetaan`-lauseessa määritellään olemassa olevat lähtevät yhteydet. `Kun`-lauseella määritellään, koska käyttäjä haluaa lähteä. Lopuksi `Niin`-lauseessa esitetään odotettu tulos, joka on tässä tapauksessa klo 8:02.

Tällainen löyhästi rakenteellinen kuvauskieli antaa selkeän muotin, jota skenaariokuvausten kirjoittajien on hyvä noudattaa. Voi kuitenkin olla tilanteita, joissa rakenteel-

linen kieli voi olla kömpelöä, jolloin kannattaa hyödyntää muita tapoja mahdollisuuksien mukaan. Testauksen kannalta rakenteellinen kieli on hyvä, koska testaaja voi sen avulla vaihe vaiheelta suorittaa testin kuvaamat askeleet. Suurin hyöty rakenteellisuudesta saavutetaan silloin, kun skenaarioiden testaus automatisoidaan. Tällöin skenaariokuvaukset ovat sekä ihmisen että tietokoneen ymmärtämässä muodossa ja käytetyt työkalut sanelevat, millaista kieltä voidaan käyttää (Smart 2015, s. 39). Aiemmin esitetyt esimerkit ovat täysin automatisoitavissa sellaisenaan.

Siinä missä toiminnalliset vaatimukset, myös ei-toiminnalliset vaatimukset tuovat liiketoiminnallista lisäarvoa. Ei-toiminnallisia vaatimuksia ovat esimerkiksi tehokkuus, vakaus, saavutettavuus ja turvallisuus. Myös monia ei-toiminnallisia vaatimuksia voidaan määrittää skenaariokuvausten avulla. Aluksi on määritettävä luotettava ja konkreettinen taso, joka ohjelmistolta vaaditaan, jotta se täyttää vaatimuksen. Esimerkiksi tehokkuudessa prosessorin tai muistin kuormitus ei kerro todellista liiketoiminnallista tavoitetta. Tällaiset matalan tason kuvaukset ovat myös epämääräisiä. Esimerkiksi prosessorin kuorma saattaa olla korkea ympäristöstä johtuvista syistä eikä sen vuoksi, että ohjelma toimisi väärin. Sen sijaan parempi lopputulos saadaan, jos skenaario ilmaistaan konkreettisesti seuraavan skenaariokuvauksen esittämään tapaan.

```
1: Ominaisuus: Ohjelman tehokkuus
2: Tapaus: Lennon tilapäivityksen tehokkuus
3: Oletetaan että palvelin on käynnissä
4: Kun reittikyselyitä saapuu 5 kertaa se määrä, joka tulee tavallisesti
       ↪ ruuhkaisimman tunnin aikana
5: Niin 95% kyselyistä tulisi käsitellä 10s kuluessa
6: Ja 100% kyselyistä tulisi käsitellä 20s kuluessa
7: Ja 100% kyselyiden käsittelystä tulisi onnistua
```

Jotta tällaiset skenaariot ja niistä johdetut testit olisivat luotettavia, on niissä käytettävä todenmukaista tai todellista dataa. Aina tämä ei kuitenkaan ole mahdollista, jos kyse on täysin uudesta järjestelmästä. (Smart 2015, s. 255-257)

2.3 Käyttäjätarinat ja skenaariokuvaukset yhteistyön välineenä

Käyttäytymisohjatun kehityksen keskeisiin periaatteisiin kuuluu yhteinen yleiskieli (ubiquitous language), jonka kehittävät kehitystyössä mukana olevat osapuolet yhteistoimin (Solis ja Wang 2011). Kieltä kootaan tietoisesti siitä luonnollisesti puhutusta kielestä, jota muutenkin käytetään kommunikoinnissa (Sathawornwichit ja Hosono

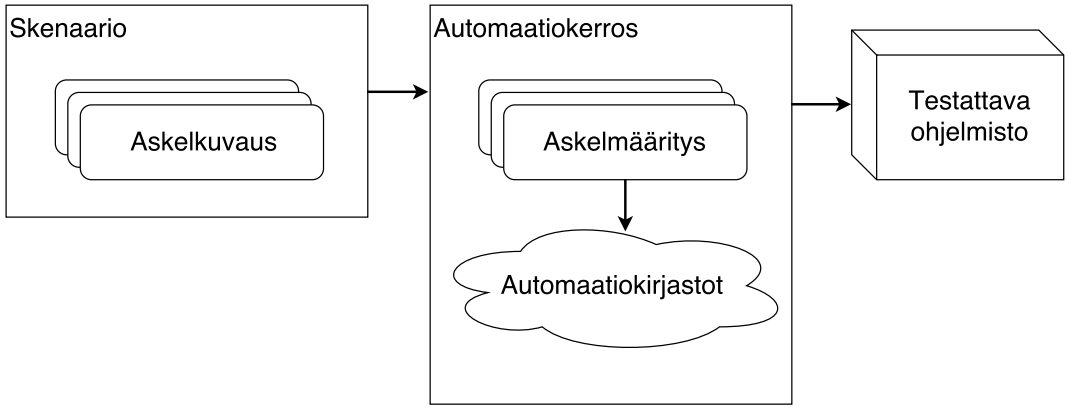
2012). Kieli ei synny hetkessä, vaan sanastoa rakennetaan vähitellen projektin edetessä. Sanaston on hyvä olla mahdollisimman yksiselitteistä, jotta vältytään väärinymmärryksiltä (Adzic 2011, s. 187). Väärinymmärrysten riski kasvaa, kun kehitystyöhön osallistuu henkilöitä, jotka eivät ole olleet alusta alkaen mukana kehittämässä yhteistä kieltä eivätkä näin ollen tunne sanastoa välttämättä riittävän hyvin (Soeken, Wille ja Drechsler 2012).

Koska skenaariokuvausten tärkeä tehtävä on toimia kommunikaatiovälineinä, käytetään niissä samaa yhteistä kieltä. Samoja esimerkkejä käyttävät niin liiketoiminnan edustajat, asiakas kuin kehitystiimikin ja kaikkien osapuolten on kyettävä ymmärtämään, mitä näillä tarkoitetaan. Skenaariokuvausten kielessä tulisi myös välttää liian teknistä, yleensä vain testaajien ja kehittäjien käyttämää, termistöä. (Adzic 2011, s. 132)

Skenaarioiden perimmäinen ja ensisijainen tehtävä on kuvata todelliset tarpeet, joita ohjelmistolla ollaan täyttämässä (Baillon ja Bouchez-Mongardé 2010). Ne ovat konkreettisia esimerkkejä, joiden avulla hälvennetään epäselvyyksiä ja poistetaan epävarmuutta. Skenaarioita ovat määrittämässä yhdessä eri sidosryhmien edustajat, joista kukin tuo oman näkökulmansa. Määritystyössä olisi hyvä olla mukana ainakin testaaja, kehittäjä, tuoteomistaja sekä asiakkaan edustaja (Smart 2015, s. 101-102). Skenaarioiden avulla luodaan yhteinen käsitys siitä, miten ohjelmiston tulisi toimia eri tilanteissa. Kun alustava yhteisymmärrys on saavutettu, voidaan uusilla skenaarioilla laajentaa ja tarkentaa käsitystä edelleen. (Adzic 2011, s. 99-103)

3 Skenaariokuvausten automatisointi

Kuva 3 esittää, kuinka käyttäjätarinoita tarkentavat skenaariot linkittyvät automaatiokerroksen välityksellä varsinaiseen testattavaan ohjelmistoon. Skenaariokuvausten ilmaisuun käytetään yleensä rakenteellista kieltä, joka mahdollistaa niiden helpon automatisoinnin. Rakenteellisuudesta huolimatta skenaariot koostuvat selkeistä ihmisen luettavista lauseista, joten myös vähemmän teknisesti orientoituneet ihmiset voivat vaivatta lukea niitä.



Kuva 3: Skenaariokuvausten automatisoinnin kerrokset.

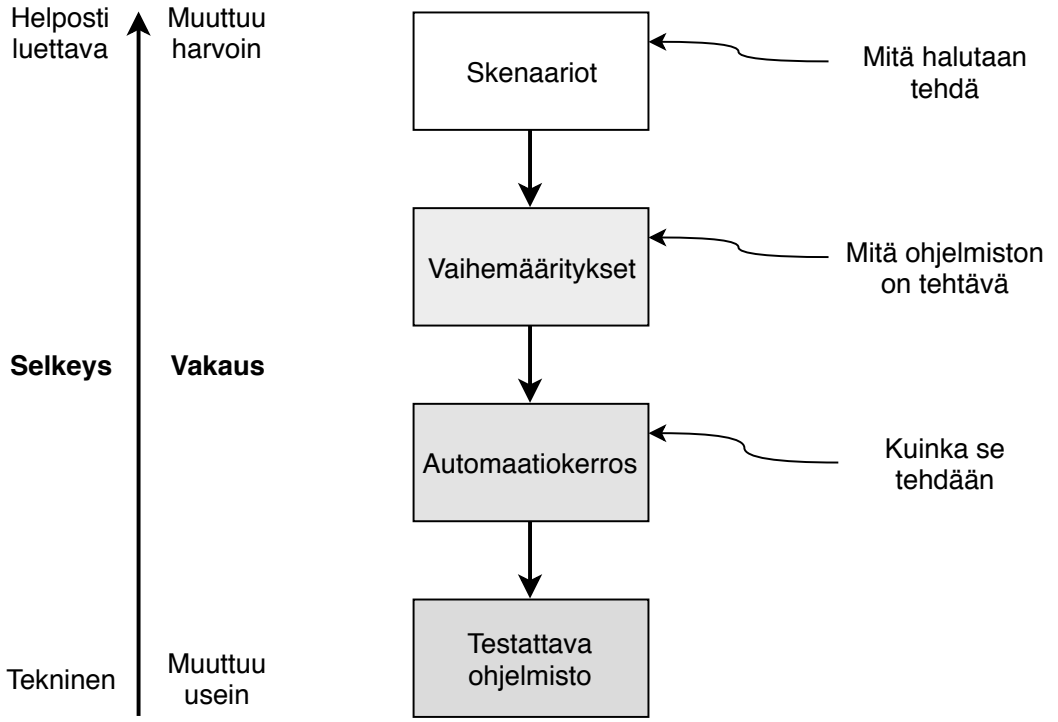
Skenaariokuvausten automatisoinnissa skenaarion askelille tehdään ohjelmoimalla askelmäärityksset. Nämä määritykset vuorovaikuttavat automaatiokirjastoja hyödyntäen varsinaisen testattavan ohjelmiston kanssa. Tätä automaatiokerrosta tarkastellaan luvussa 3.1

Luvussa 3.2 tutustutaan skenaarioiden kirjoittamiseen Gherkin-syntaksilla käyttäen Oletetaan-Kun-Niin -rakennetta. Luvussa 3.3 tarkastellaan, kuinka skenaariot automatisoidaan hyödyntäen Cucumber-JVM testauskirjastoa. Lopuksi luvussa 3.4 perehdytään automatisoitujen skenaarioiden suorittamiseen.

3.1 Automaatiokerros

Hyvin toteutettu testauksen automatisointi koostuu useista arkkitehtuurikerroksista kuvan 4 mukaisesti. Ylimmällä tasolla ovat skenaariokuvaukset ja alimmalla tasolla itse testattava ohjelmisto. Skenaarioiden tulee olla ihmisen ymmärtämässä muodossa ja kertoa liiketoiminnan termein, mitä ohjelmistolla halutaan saavuttaa. Skenaarioiden

alla ovat skenaarion vaiheiden määritykset, jotka ovat korkein varsinaisella ohjelmointikielellä kirjoitettu taso. Tämän kerroksen tehtävä on kuvata, mitä ohjelmiston tulisi tehdä, jotta skenaarion kuvaama toiminta tai tila saavutettaisiin. Jotta vaihemäärityket olisivat helposti ymmärrettäviä ja ylläpidettäviä on ne pidettävä yksinkertaisina. Tämän vuoksi yleensä tarvitaan vielä yksi tai useampi alemman tason testiautomaatiokerros. Vasta alimmat kerrokset vuorovaikuttavat varsinaisen testattavan ohjelmiston kanssa. (Smart 2015, s. 145-146)



Kuva 4: Testiautomaation arkkitehtuuri koostuu useita kerroksista. Mukaillen (Smart 2015, s. 146,195)

Kuvassa 4 esitetään myös, kuinka luettavuus vaikeutuu ja epävakaus lisääntyy siirryttäessä alemmalle tasolle arkkitehtuurissa. Skenaariokuvaukset ovat luonnollisella kielellä kirjoitettuja määrityksiä, joten niiden tulkitsemiseen ei yleensä tarvita teknistä osaamista. Alimmalla tasolla lähdekoodin tulkitseminen sen sijaan vaatii erityistä osaamista. Vakaudella tarkoitetaan tässä yhteydessä sitä, kuinka usein arkkitehtuurikerrokseen tulee muutoksia. Skenaariokuvaukset pyritään pitämään abstraktilla tasolla ja korkean tason vaatimukset harvemmin muuttuvat. Sen sijaan määritysten toteutus saattaa muuttua useinkin. Erityisesti käyttöliittymään tehdään herkästi muutoksia, vaikka perimmäinen toiminnallisuus pysyykin samana.

Testiautomaatiokerroksen yksi tärkeä tehtävä on mahdollistaa ohjelmakoodin uudel-

leenkäyttö, kun samoja testiautomaatiokerroksen tarjoamia toimintoja voidaan hyödyntää eri ominaisuuksien vaihemäärityksissä. Tämä on ylläpidettävyyden kannalta hyvin tärkeää. Jos varsinainen vuorovaikutus testattavan ohjelman kanssa tehtäisiin jokaisessa vaihemäärityksessä erikseen, saattaisi pieni muutos testattavassa ohjelmistossa rikkoa useita testejä. Koska testattava ohjelmisto on jatkuvan muutoksen tilassa, voivat testit hajota usein ja jokaisen muutoksen jälkeen pitäisi testit korjata yksi kerrallaan. Liian suuri ylläpitotaakka tekisi testauksen automatisoinnin hyvin nopeasti hyödyttömäksi (Fewster ja Graham 1999, s. 11). Sen sijaan, jos käytössä on alempi testiautomaatiokerros, joka on vuorovaikutuksessa testattavan ohjelmiston kanssa ja jota kaikki vaihemääritykset käyttävät, riittää korjauksen tekeminen testiautomaatiokerrokseen. (Smart 2015, s. 194)

Ei ole olemassa suoraa sääntöä, kuinka moneen alikerrokseen automaatiokerros tulisi jakaa. Tämä riippuu hyvin paljon testattavasta ohjelmistosta ja skenaariokuvauksista. Monimutkaiset skenaariot vaativat useamman tason, jotta ylläpidettävyys ja selkeys saadaan taattua. Teknisen kerroksen toteutus tehdään vasta kun rajapinnat ovat riittävän vakaat. Sen sijaan vaihemääritykset on hyvä toteuttaa jo ennen rajapintojen tekoa. Ne määritetään yhdessä kehittäjien, testaajien ja liiketoiminta-analyytikon kanssa. Varsinainen käyttöliittymä tehdään näiden määritysten pohjalta. (Smart 2015, s. 199)

Kaikkien testien ei tule kattaa ohjelman kaikkia arkkitehtuuritasoja päästä päähän. Ongelma kattavissa testeissä on, että ne särkyvät hyvin helposti. Pieni virhe missä tahansa ohjelman arkkitehtuuritasossa hajottaa testin ja testin ollessa hyvin laaja voi vian löytyminen olla vaikeaa. Sen vuoksi testien tulisi testata hyvin rajallisella alueella. Näiden rajattujen testien lisäksi voidaan käyttää muutamaa päästä päähän testaavaa testiä. Tällainen testi voi olla esimerkiksi yksinkertaisempi skenaario, joka käyttää ohjelmiston eri osia eri arkkitehtuuritasoilla. (Adzic 2011, s. 172)

Automaatiokerroksen alin kerros vuorovaikuttaa testattavan ohjelmiston kanssa jonkin rajapinnan kautta. Jos ohjelmistolla on käyttöliittymä, jonka kautta ihminen sitä käyttää, on tämä sama rajapinta yksi mahdollisuus myös testejä varten. Kuitenkin erityisesti graafisia rajapintoja kannattaa välttää, mikäli mahdollista. Käyttöliittymiä muutetaan varsin usein ohjelmistoprojektien aikana, jolloin myös testit hajoavat herkästi muutosten myötä. Käyttöliittymän kautta toimivat testit ovat usein myös hitaita. Käyttöliittymää parempi tapa on vuorovaikuttaa ohjelmiston kanssa suoraan käyttöliittymän alla olevan kerroksen kautta. (Adzic 2011, s. 149-151)

Skenaarioiden automatisointi toiminnot nauhoittamalla saattaa olla houkutteleva tapa toteuttaa automatisointi. Nauhoittamisessa testaaja suorittaa skenaarion kuvaamat vaiheet testattavalla ohjelmistolla samaan aikaan kun erillisellä nauhoitusohjelmalla tallennetaan kaikki testaajan tekemät toiminnot. Myöhemmin tämä nauhoite voidaan toistaa ilman ihmisen vuorovaikutusta. Nauhoittaminen on todella nopea tapa tehdä automatisointi, mutta se ei yleensä ole suositeltavaa. Jotta nauhoittaminen olisi mahdollista, on käyttöliittymän oltava olemassa, mikä on vastoin käyttäytymisohjatun kehityksen periaatteita, joiden mukaan ohjelmakoodi syntyy vasta skenaarioiden automatisoinnin jälkeen. Nauhoitukset ovat usein myös herkästi hajoavia, koska ne kohdistuvat usein muuttuvaan käyttöliittymään. Myös ylläpito on vaativaa, koska erillistä automaatiokerrosta ei voida tehdä testin ja testattavan ohjelmiston väliin, mikä lisää muutostarpeita itse testeihin. (Adzic 2011, s. 155)

3.2 Skenaarioiden rakenne

Kuten jo luvussa 2.2 nähtiin, voidaan skenaariokuvauksissa käyttää Oletetaan-Kun-Niin -rakennetta. Cucumber-testikirjastojen kohdalla tätä rakennetta kutsutaan myös Gherkin-kieleksi. Toinen yleisesti käytetty kirjasto on JBehave, jonka syntaksi vastaa hyvin pitkälti Gherkiniä (JBehave 2017). Cucumberissa skenaariot kirjoitetaan ominaisuustiedostoihin. Ne ovat yksinkertaisesti tekstitiedostoja, joiden pääte on `.feature`. Kussakin ominaisuustiedostossa kuvataan yksi ominaisuus kattavasti skenaariokuvausten avulla. (Cucumber limited 2017)

Alla kuvatut esimerkit esittelevät, kuinka Gherkin-kielellä voidaan kuvata skenaarioita, jotka kuvaavat yksinkertaisen laskimen toimintaa. Esimerkkien englanninkieliset versiot ovat peräisin Cucumber-JVM -testikirjaston lähdekoodin esimerkeistä (Cucumber limited 2018). Esimerkit on käännetty suomen kielelle hyödyntäen Cucumber-JVM-kirjaston suomen kielen tukea. Esimerkit on myös testattu ja ne toimivat yhä käännöksen jälkeen.

Tarkastellaan alla esitettyä skenaariota, joka kuvaa kahden luvun yhteenlaskua laskinohjelmalla. Ominaisuustiedoston alussa ilmaistaan ominaisuuden nimi käyttämällä `Ominaisuus`-avainsanaa. Esimerkissä ominaisuuden nimi on `Peruslaskutoimitukset`. Tämän jälkeen tiedostossa listataan ominaisuuden skenaariokuvaukset. Kukin skenaariokuvaus alkaa skenaarion nimellä, joka aloite-

taan avainsanalla `Tapaus`, kuten esimerkissä `Tapaus: Yhteenlasku`. Nimen jälkeen alkaa varsinainen skenaarion kuvaus Oletetaan-Kun-Niin -rakenteella. Aluksi `Oletetaan`-lauseilla kuvataan skenaarion alkutila, josta se lähtee liikkeelle. Esimerkissä `Oletetaan`-lauseella määritetään, että laskin on käynnistetty. Alustuksen jälkeen `Kun`-lauseilla määritetään suoritettava toiminta. Esimerkissä laskimella lasketaan yhteen luvut neljä ja viisi. Lopuksi `Niin`-lauseilla määritetään odotettu lopputila. Esimerkissä odotetaan, että laskin näyttää yhteenlaskun tulokseksi lukua yhdeksän.

```
1: Ominaisuus: Peruslaskutoimitukset
2:   Tapaus: Yhteenlasku
3:     Oletetaan laskin, jonka juuri käynnistin
4:     Kun lasken yhteen 4 ja 5
5:     Niin tulos on 9
```

Seuraava esimerkki esittää, kuinka skenaarioiden alkumääritys voidaan tehdä globaaliksi siten, että se suoritetaan ennen kutakin skenaariota. Tässä tapauksessa ennen kutakin skenaariota suoritetaan `Tausta`-vaihe, jossa laskin käynnistetään. Tämän jälkeen suoritetaan varsinainen skenaario.

```
1: Ominaisuus: Peruslaskutoimitukset
2:   Tausta: Laskin
3:     Oletetaan laskin, jonka juuri käynnistin
4:   Tapaus: Yhteenlasku
5:     Kun lasken yhteen 4 ja 5
6:     Niin tulos on 9
7:   Tapaus: Toinen yhteenlasku
8:     Kun lasken yhteen 4 ja 7
9:     Niin tulos on 11
```

Gherkin-kieli tukee myös taulukkomuotoista dataa. Tapoja tähän on kaksi: skenaarion vaihekohtainen taulukko tai koko skenaariota koskeva taulukko. Alla esitetyssä skenaariossa käytetään molempia tapoja. Ensimmäisessä `Oletetaan`-lauseessa on vaihekohtainen taulukko, jolla määritetään aiemmin suoritetut laskutoimitukset. Kun skenaarion ensimmäistä vaihetta suoritetaan, käytetään koko taulukon dataa kyseisen vaiheen parametrina. Sen sijaan toinen taulukkojen hyödyntämistapa ilmenee kahdella taulukolla skenaarion lopussa. Nämä taulukot koskevat koko skenaariokuvausta ja mahdollistavat esimerkkiperustaisen testauksen. Taulukoiden jokainen rivi kuvaa yhtä mahdollista tapausta ja koko skenaario suoritetaan kerran kutakin taulukon riviä kohden. Kussakin suorituksessa syötetään taulukon rivillä olevat arvot parametreina vaihekuvauksiin. Kun vaihetta `Ja lasken yhteen <a> ja <b>` suoritetaan, syö-

tetään taulukosta sarakkeiden a ja b -arvot parametreina skenaarion vaiheen <a> ja <b> parametrimääritysten paikalle.

```
1: Ominaisuus: Peruslaskutoimitukset
2:   Tapausaihio: Monta yhteenlaskua
3:     Oletetaan seuraavat aiemmat syötteet:
4:       | ensimmainen | toinen | operaatio |
5:       | 1           | 1      | +         |
6:       | 2           | 1      | +         |
7:     Kun painan +
8:     Ja lasken yhteen <a> ja <b>
9:     Ja painan +
10:    Niin tulos on <c>
11:    Tapaukset: yksinumeroiset luvut
12:      | a | b | c  |
13:      | 1 | 2 | 8  |
14:      | 2 | 3 | 10 |
15:    Tapaukset: kaksinumeroiset luvut
16:      | a  | b  | c  |
17:      | 10 | 20 | 35 |
18:      | 20 | 30 | 55 |
```

3.3 Vaihemääritykset

Kun skenaariokuvaukset on kirjoitettu rakenteellisella kielellä, täytyy ne jotenkin saada yhdistettyä testattavaan järjestelmään. Tätä osaa hoitavat vaihemääritykset (step definition). Vaihemääritys on jollakin ohjelmointikielellä kirjoitettu lyhyt ohjelmakoodi, joka määrittää, mitä ohjelmiston tulee tehdä siihen liittyvän skenaarion vaiheen kohdalla. Automatisointikirjasto tulkitsee skenaariokuvausta ja jokaisen Oletetaan-, Kun- tai Niin-vaiheen kohdalla etsii ja suorittaa oikean vaihemäärityksen. Automatisointikirjasto myös hoitaa mahdollisten parametrien ja parametritaulukoiden välittämisen skenaariosta vaihemääritykselle. (Smart 2015, s. 144,145)

Alla oleva ohjelmakoodi on vaihemääritys aikaisemmin esitetyille laskimen toimintoja kuvanneille skenaarioille. Ohjelmakoodi hyödyntää Cucumber-JVM -kirjaston Java 8 -tukea, jossa skenaariot kuvataan lambdanotaatiolla. Vaihemääritykset on sijoitettu metodikutsuihin luokan konstruktoriin. Kukin Oletetaan-, Kun- ja Niin-lause vastaa yhtä skenaarion vaihetta. Se, mihin vaiheeseen vaihemääritys linkittyy, määräytyy vaiheen tekstistä, joka on metodin ensimmäisenä parametrina. Vaiheen teksti kuvataan säännöllisellä lausekkeella (regular expression). Vaihemetodin toinen parametri määrittää, mitä vaiheessa suoritetaan. Esimerkissä ensimmäinen `Oletetaan`-lause luo uuden laskinobjektin. Seuraava `Kun`-lause lukee parametrina annetut luvut ja syöttää ne las-

kimeen. Lopuksi esimerkin `Niin`-lauseessa luetaan parametrina annettu odotettu tulos ja tarkistetaan, että laskimen antama tulos on sama.

```
 1: public class LaskinTesti implements Fi {
 2:     private Laskin laskin;
 3:     public LaskinTesti() {
 4:         Oletetaan("^laskin, jonka juuri käynnistin$", () -> {
 5:             laskin = new Laskin();
 6:         });
 7:         Kun("^lasken yhteen (\\d+) ja (\\d+)$", (Integer luku1, Integer luku2)
        ↪ -> {
 8:             laskin.paina(luku1);
 9:             laskin.paina(luku2);
10:             laskin.paina("+");
11:         });
12:         Oletetaan("^painan (.+)$", (String operaatio) -> laskin.paina(operaatio)
        ↪ );
13:         Niin("^tulos on (\\d+)$", (Double odotettu) -> assertEquals(odotettu,
        ↪ laskin.value()));
14:         Oletetaan("^seuraavat aiemmat syötteet:$", (DataTable taulukko) -> {
15:             List<Syote> syotteet = taulukko.asList(Syote.class);
16:             for (Syote syote : syotteet) {
17:                 laskin.paina(syote.ensimmainen);
18:                 laskin.paina(syote.toinen);
19:                 laskin.paina(syote.operaatio);
20:             }
21:         });
22:     }
23:     public class Syote {
24:         Integer ensimmainen;
25:         Integer toinen;
26:         String operaatio;
27:     }
28: }
```

Vaihemäärityksessä on vain yksi metodi `Kun`-vaiheen toteutukselle. Tämä sama vaihemääritys kuitenkin toteuttaa kahdessa aiemmin esitetyssä skenaariossa olevat `kun`-vaiheet: `Kun lasken yhteen 4 ja 5` ja `Kun lasken yhteen 4 ja 7`. Tällöin skenaariosta luetaan yhteenlaskettavat luvut ja ne syötetään vaihemäärityksen toteuttavalle metodille. Samaa vaihemääritystä voidaan käyttää myös taulukkomuotoista dataa hyödyntävässä skenaariossa vaiheen `Ja lasken yhteen <a> ja <b>` toteuttamisessa. Tällöin luvut luetaan taulukosta ja syötetään vaihemäärityksen metodille.

`Oletetaan`-vaiheelle, jonka kuvausteksti on `^Seuraavat aiemmat syötteet:$` syötetään taulukkomuotoista dataa. Data annetaan vaihemääritykselle DataTable-objektina. Cucumber-JVM antaa hyvin vapaat mahdollisuudet

lukea data taulukosta, jotta monet erilaiset skenaariot olisivat mahdollisia. Tässä tapauksessa data luetaan riveittäin Syote-objekteiksi, joista se edelleen ohjataan laskinohjelmalle.

3.4 Automatisoitujen skenaarioiden suorittaminen

Automatisoitujen skenaarioiden suoritus riippuu paljon siitä, mitä automaatiokirjastoa hyödynnetään. Cucumber-JVM:n kohdalla skenaariot voidaan suorittaa esimerkiksi käynnistämällä Cucumber-JVM komentokehotteen kautta. Yleensä skenaariot voidaan suorittaa myös suoraan integroidun kehitysympäristön kautta tai käyttämällä testiprojektin hallinnassa käännöksenhallintajärjestelmää, kuten Mavenia tai Gradlea. Joka tapauksessa skenaarioiden suorituksen pitäisi olla mahdollisimman helppoa, jotta se tulisi tehtyä mahdollisimman usein muutosten jälkeen. Vielä edullisempaa on, jos skenaarioiden suoritus saadaan yhdistettyä jatkuvan integraation järjestelmään (Continuous integration) (Smart 2015, s. 325-326). Tällöin ne voidaan suorittaa automaattisesti aina kun uusia muutoksia tehdään. Koska suuren skenaariomäärän suoritus on yleensä hidasta, erityisesti käyttöliittymätestien kohdalla, on usein järkevää suorittaa skenaariot öisin.

Koska käyttäytymisohjatut testit vuorovaikuttavat yleensä käynnissä olevan ohjelmiston kanssa, on myös automaattisessa testien ajossa oltava käytössä käynnissä oleva ohjelma. Näin ollen automaattinen testien suoritus vaatii, että myös ohjelmiston ajoympäristö saadaan asetettua käyttökuntoon automaattisesti.

Kun automatisoitu skenaario on suoritettu, voi skenaario olla yhdessä neljästä eri tilasta: suoritus onnistui (success), suoritus epäonnistui (fail), suoritus päättyi virheeseen (error) tai skenaario odottaa suoritusta (pending). Tila riippuu skenaarion kunkin vaiheen suorituksen tilasta. Vaiheilla on kullakin samat neljä tilaa ja lisäksi suorituksen ohitus (skip). Skenaarion vaiheiden tila heijastuu myös itse skenaarion tilaan. Skenaario merkitään onnistuneeksi vain, jos kaikki vaiheet on suoritettu onnistuneesti. Jos yhdenkin vaiheen suoritus epäonnistuu, myös skenaario merkitään epäonnistuneeksi. Vaiheen epäonnistuminen tarkoittaa, että ohjelmisto ei täytä skenaarion vaiheen odotuksia, esimerkiksi Niin-vaiheen tarkistuksessa odotettu tulos ei vastaa havaittua tulosta. Virhe sen sijaan tarkoittaa, että testattava ohjelmisto päätyi virheelliseen tilaan ja tällöin joko ohjelma tai testi on viallinen. (Smart 2015, s. 152-153)

Vaiheen tila on odottava, kun suoritusta ei ole vielä aloitettu, automaatiokirjasto ei löydä sille vaihemääritystä tai vaihemääritys on keskeneräinen. Tämä tilanne on yleinen kehitystyön aikana, kun uusia skenaarioita on luotu, mutta vaihemäärityksetovat vielä tekemättä. Jos odottava tila johtuu puuttuvasta tai keskeneräisestä vaihemäärityksestä, merkitään myös skenaario odottavaan tilaan ja jatketaan testausta seuraavalla skenaariolla. Jos vaihe merkitään suorituksen jälkeen epäonnistuneeksi tai odottavaksi merkitään kaikki tulevat vaiheet ohitetuiksi. Jos esimerkiksi ainoastaan suoritettavan skenaarion ensimmäisellä vaiheella on vaihemääritys, suorittaa automaatiokirjasto ensimmäisen vaiheen. Jos vaiheen tulos on onnistunut, jatkuu suoritus seuraavaan vaiheeseen. Koska seuraavalla vaiheella ei ole vaihemääritystä, merkitään se odottavaan tilaan ja kaikki seuraavat vaiheet merkitään ohitetuiksi. Lopuksi itse skenaario merkitään odottavaan tilaan. (Smart 2015, s. 152-153)

Yksi automaattisten testien ongelma on ei-deterministiset testit. Tällaiset testit menevät läpi toisinaan ja toisinaan päättyvät virheeseen. Tämä vaihtelu on satunnaista eli ympäristö tai ohjelmakoodimuutokset eivät siihen vaikuta. Jos testin tilaan ei voi luottaa, menettää testi merkityksensä, eikä testin virheistä välttämättä enää välitetä. Tämän seurauksena todelliset virheet saattavat jäädä huomaamatta. Ongelma esiintyy herkemmin juuri korkean tason testeillä, kuten automatisoiduilla skenaarioilla, koska ne kattavat suuremman osan ohjelmasta. (Fowler 2011)

Testien epävakaus saattaa johtua useista eri syistä. Testit saattavat vuorovaikuttaa toistensa kanssa, esimerkiksi edellinen testi ei ole puhdistanut ajoympäristöä tarvittavalla tavalla ennen seuraavan testin suoritusta. Yksikkötesteissä vastaavaa ongelmaa ei yleensä esiinny, koska testit suoritetaan eristyksissä toisistaan (JUnit 2018). Automatisoidut skenaariot suoritetaan yleensä toimivaa, käynnissä olevaa ohjelmistoa vasten, jolloin sen täydellinen alustaminen ennen jokaista testiä kestäisi yksinkertaisesti liian kauan.

Toinen ongelma ovat asynkroniset toiminnot, joissa kutsuun liittyvä vastaus saattaa eri ajokerroilla tulla vaihtelevalla viiveellä. Ongelmia voivat aiheuttaa myös ulkoiset palvelut, joiden palautteet saattavat vaihdella suorituskerrasta toiseen. Martin Fowler ehdottaa, että tällaiset palvelut kannattaa korvata testejä varten tehdyillä sijaispalveluilla, jotka matkivat todellisten palveluiden ominaisuuksia (Fowler 2011).

Yleinen ratkaisu epävakaisiin testeihin on eristää ne toimivista testeistä. Mikäli tällaisia testejä kertyy useampia, kannattaa varata aikaa niiden korjaamiseen, jotta niiden määrä

ei pääse kasvamaan liian suureksi. (Fowler 2011)

4 Käyttäytymisohjattu kehitys yksikkötasolla

Testiohjattu kehitys (Test-Driven Development, TDD) on Kent Beckin esittelemä tapa toteuttaa ohjelmistoa. Beck esitteli sen osana XP Programming menetelmää, joka on yksi tapa toteuttaa ketterää kehitysmallia (Beck 2003). Tosin, kuten Beck itsekin on maininnut, ei kyseessä ole mikään uusi menetelmä (Beck 2012). Samankaltaista menetelmää hyödynsi esimerkiksi Nasa jo 1960-luvulla Mercury-avaruusohjelmassa (Williams, Maximilien ja Vouk 2003). Testiohjattu kehitys on otettu yleiseen käyttöön kuitenkin vasta viime aikoina (Koskela 2008, s. 4).

Yleensä testiohjattua kehitystä käytetään matalan tason yksikkötestiohjatussa kehityksessä. Yksikkötesteillä pyritään varmistamaan, että yksittäiset ohjelmiston osat toimivat oikein, kun taas korkeamman tason toiminnallisilla testeillä testataan, että osat yhdessä tai mahdollisesti sovellus kokonaisuudessaan toimii oikein. Toinen huomattava ero on siinä, että skenaarioihin pohjautuvat testit määritetään yhdessä eri osapuolten kesken. Sen sijaan yksikkötestit määrittää yleensä kehittäjä itsenäisesti kehitystyön aikana. (Soeken, Wille ja Drechsler 2012)

Testiohjatun kehityksen juuret näkyvät vahvasti siinä, että TDD-lyhenteellä viitataan usein juuri yksikkötestiohjattuun kehitykseen. Tässä tutkielmassa sillä kuitenkin viitataan testiohjattuun kehitykseen, jonka yksi sovellus on yksikkötestauksessa. Yksikköohjattu kehitys painottaa testien automatisointia ja on sitä kautta vahvasti työkaluriippuvainen. Lukuisille eri kielille on kehitetty yksikköohjattua kehitystä varten xUnit-testityökaluja. Ensimmäinen näistä oli Beckin SmallTalkille kehittämä SUnit. Javakielelle vastaava testauskirjasto on nimeltään JUnit (Koskela 2008, s. 36). Nämä työkalut mahdollistavat testikoodin suorittamisen.

Myöhemmin testiohjattua kehitystä on ryhdytty soveltamaan myös korkeamman tason testauksessa muun muassa käyttäytymisohjatun kehityksen muodossa. Myös käyttäytymisohjatun kehityksen juuret ovat vahvasti yksikköohjatussa kehityksessä: se kun lähti liikkeelle Dan Northin pyrkimyksestä tehdä yksikkötestiohjatusta kehityksestä helpommin lähestyttävä sitä opiskeleville (North 2006). Vaikka käyttäytymisohjatun kehityksen pääpaino on korkeamman tason määrityksissä ja hyväksymistason testauksessa, voidaan sitä edelleen hyödyntää myös yksikkötason testauksessa.

Tässä luvussa palataan takaisin käyttäytymisohjatun kehityksen juurille ja tutustutaan, kuinka käyttäytymisohjatun kehityksen periaatteita voidaan soveltaa yksikköohjatussa

kehityksessä. Aluksi luvussa 4.1 tutustutaan, kuinka käyttäytymisohjatun kehityksen mukainen yksikkötason testaus eroaa perinteisestä tavasta tehdä yksikköohjattua kehitystä. Luvussa 4.2 käydään läpi yksikkötason kehityssykli ja luvussa 4.3 perehdytään muutaman esimerkin kautta, kuinka yksikkötestejä käytännössä tehdään. Lopuksi luvussa 4.4 tarkastellaan, millaisia ongelmia yksikköohjatun kehityksen hyödyntämiseen liittyy.

4.1 Eroavaisuudet perinteiseen yksikköohjattuun kehitykseen

Yksikkötason testaus on perinteisesti hyvin riippuvainen työkaluista, joilla testaus automatisoidaan. Lähes kaikille nykyaikaisille ohjelmointikielille onkin kehitetty niin sanottu xUnit-testauskirjasto, jolla testaus onnistuu. Esimerkiksi Java-ohjelmoinnissa testejä voi kirjoittaa JUnit-kirjaston avulla. Usein testikoodi kirjoitetaan samalla ohjelmointikielellä kuin itse ohjelmistokin. Testikoodi on kuitenkin yleensä eristetty itse ohjelmakoodista, jolloin lopullisessa ohjelmistossa testikoodi ei ole enää rasittamassa suoritusta. Testauskirjaston lisäksi kehittäjä tarvitsee ympäristön, jossa testit voidaan vaivattomasti suorittaa. Yleensä käytössä on erillinen jatkuvan integraation mahdollistava ympäristö, joka suorittaa testejä aina kun ohjelmakoodiin tulee uusia muutoksia. (Koskela 2008, s. 37-38)

Käyttäytymisohjattu kehitys on yksikkötasolla hyvin samankaltaista kuin perinteinen yksikkötestiohjattu kehitys. Käytännössä ei ole mitään estettä käyttää testiohjattuun kehitykseen suunniteltuja kirjastoja. Esimerkiksi JUnit-kirjasto sopii hyvin käytettäväksi myös käyttäytymisohjatussa kehityksessä. Suurin ero perinteiseen yksikkötestiohjattuun kehitykseen on nimeämiskäytännöissä. Yksikkötestit nimetään usein hyvin lyhyesti ja ne on perinteisesti aloitettu “test”-sanalla (Smart 2015, s. 286). Käyttäytymisohjatussa kehityksessä testit pyritään nimeämään käyttäytymistä kuvaavalla lauseella. Kuvaava, lauseen mittainen nimi rajoittaa testiä luonnollisesti, jolloin testistä ei tule liian laaja: sen kun tulee testata ohjelman osan käyttäytymistä yhdessä tietyssä tilanteessa. Lisäksi, jos testi myöhemmin hajoaa, testin nimi kertoo suoraan, mikä ohjelmiston käyttäytymisessä ei vastaa määritystä. (North 2006)

Koska yksikkötesteissä kyse on pohjimmiltaan suoritettavista määrityksistä, pyritään käyttäytymisohjatussa kehityksessä eroon testi-sanan käytöstä (North 2006). Esimerkiksi JUnit kirjasto on teknisesti rakennettu siten, että siinä käytetään “@test”-

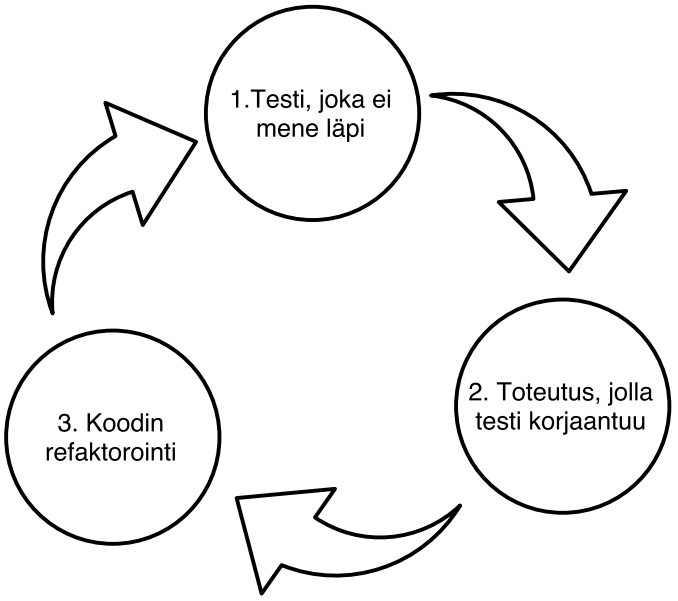
annotaatiota testimetodien merkitsemiseen (Bechtold et al. 2018). Lukuisia kirjastoja on kuitenkin luotu tukemaan paremmin käyttäytymisohjattua kehitystä yksikkötestauksessa. Tällaisia ovat esimerkiksi RSpec Rubylle, Jasmine JavaScriptille, Spock Groovylle ja ScalaTest Scalalle. Nämä kirjastot hylkäävät testi-sanan käytön kokonaan ja keskittyvät ennemmin käyttäytymisen kuvaamiseen. Esimerkiksi Spock-kirjastossa käytetään lähes samaa Given-When-Then -rakennetta, jota myös Gherkin-kielessä käytetään (Niederwieser 2018).

Yksikkötason käyttäytymisohjattu kehitys eroaa järjestelmätason kehityksestä lähinnä siinä, että määritykset ovat huomattavasti kevyemmät. Luonnollisella kielellä tehtyjä skenaariomäärityksiä ei tehdä lainkaan vaan määritykset tehdään suoraan ohjelmakoodilla. (Smart 2015, s. 265)

4.2 Yksikkötason kehityssykli

Yksikkötason käyttäytymisohjatun kehityksen kehityssykli koostuu kolmesta vaiheesta: testin määrityksestä, toiminnallisuuden toteutuksesta ja toteuttavan ohjelmakoodin refaktoroinnista eli muokkauksesta parempaan muotoon ilman, että toiminnallisuus muuttuu. Tämä sykli on esitetty kuvassa 5. Sykli alkaa halutun käyttäytymisen määrityksellä. Käytännössä määritys on suoritettavissa oleva yksinkertainen ohjelmakoodi, joka kutsuu määritettävän ohjelman yksittäistä osaa, kuten esimerkiksi metodia tai luokkaa. Se pyrkii varmistamaan, että osa todella tekee sen, mitä siltä odotetaan. Uutta toimintoa tehtäessä prosessi alkaa mahdollisimman yksinkertaisen määrityksen laatimisella. Tässä vaiheessa määrityksen suoritus epäonnistuu, koska testattava ohjelmisto ei vielä toteuta sen kuvaamaa käyttäytymistä. (Smart 2015, s. 271)

Seuraavassa vaiheessa ohjelmisto “korjataan”, eli varsinainen toteutus muunnetaan vastaamaan uutta määritystä. Käytännössä ohjelmoidaan mahdollisimman vähän, mutta riittävästi, jotta määrityksen suoritus saadaan menemään läpi. Uuden ohjelmakoodin ei tarvitse tässä vaiheessa olla kaunista tai tehokasta. Pääasia on, että se toteuttaa halutun toiminnallisuuden. Parannusten aika tulee vasta myöhemmin, kun tarvittava toiminnallisuus on valmis ja sen tarpeista on parempi ymmärrys ohjelmakoodin näkökulmasta. Nyt kaikkien suoritettavien määritysten pitäisi ajautua virheittä läpi: Mikäli näin ei ole, on joko uudessa toiminnallisuudessa tai määrityksissä vikaa. (Koskela 2008, s. 16-18; Smart 2015, s. 271)



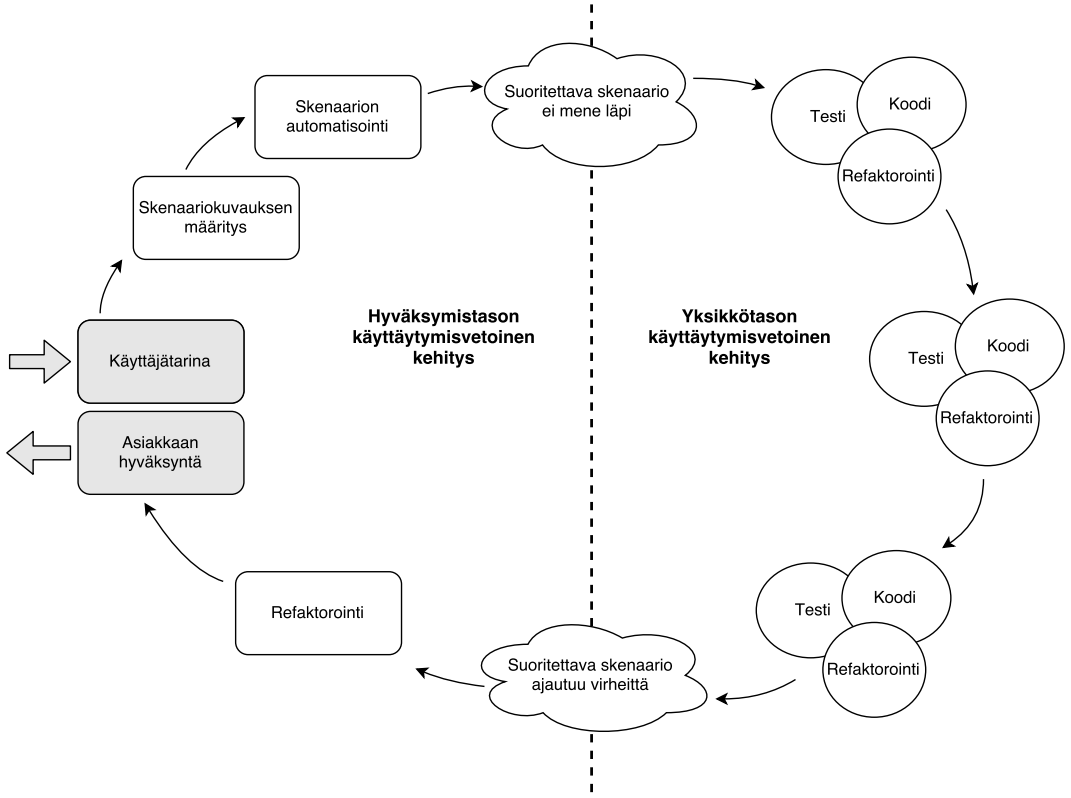
Kuva 5: Kehityssykli yksikköohjatussa kehityksessä. (Smart 2015)

Kun tarvittava toiminnallisuus on saavutettu ja kaikki testit suoriutuvat ilman virheitä, voidaan ryhtyä parantamaan ohjelmakoodin rakennetta eli refaktoroimaan. Tässä vaiheessa kehittäjä ryhtyy muokkaamaan ohjelmakoodia helpommin ymmärrettävään muotoon, parantamaan ohjelman rakennetta ja mahdollisesti myös tehostamaan sen toimintaa. Tarkoitus ei kuitenkaan ole enää muuttaa ohjelman toiminnallisuutta. Suoritettavat määritykset toimivat testeinä, joiden tehtävänä on juuri estää ohjelmiston käyttäytymisen muuttuminen vahingossa. Ne ovat kehittäjän turvana ja lisäävät luottamusta ohjelmakoodin toimivuuteen, kun kehittäjä etsii parasta mahdollista muotoa toteutukselle. (Koskela 2008, s. 19)

Nyt sykli on käyty loppuun ja pieni osa ohjelmistosta on toteutettu. Seuraavaksi sykli alkaa jälleen alusta seuraavan tehtävän tai skenaarion osalta. Sykliä toistetaan, kunnes koko ominaisuus on valmis ja kaikki testit suoriutuvat virheittä. (Smart 2015, s. 270)

Kun ohjelmisto on katettu suoritettavilla määrityksillä, antavat ne ohjelmoijalle tehokkaan työvälineen. Testien suoritus antaa lähes välittömän palautteen siitä, kuinka ohjelmoijan toiminta on muokannut koko ohjelmiston käyttäytymistä. Kun ohjelmiston kehitys pilkotaan hyvin pieniin kertamuutoksiin, ovat mahdolliset virheet nopesti paikannettavissa. Virheen etsintään ei tällöin kulu juuri aikaa ja korjaukset tapahtuvat hyvin nopeasti. Näin ollen aikaa jää enemmän ohjelmakoodin laadulliseen parantamiseen. (Koskela 2008, s. 9)

Pienet muutokset tarkoittavat myös sitä, että yksittäiset ratkaistavat ongelmat ovat pieniä. Kehittäjän tarvitsee keskittyä kerralla vain rajalliseen osaan ongelmasta ja kaikkia yksityiskohtia ei tarvitse pitää mielessä. (Koskela 2008, s. 21-22)



Kuva 6: Yksikköohjattu kehitys käyttäytymisohjatussa kehityksessä mukaillen (Koskela 2008, s. 342)

Yksikkötason käyttäytymisohjattu kehitys sulautuu luonnollisesti myös järjestelmätason käyttäytymisohjattuun kehitykseen, kuten nähdään kuvasta 6. Kuvassa hyväksymistestiohjattu ja yksikkötestiohjattu kehitys on korvattu hyväksymis- ja yksikkötason käyttäytymisohjatulla kehityksellä John Ferguson Smartin BDD in Action -kirjassa kuvaamalla tavalla (Smart 2015, s. 261). Kuvan esittämä sykli alkaa siitä, kun uusi käyttäytymistarina otetaan työstettäväksi. Aluksi käyttäytymistarinaan tehdään yksi tai useampi skenaariokuvaus tavanomaisesti käyttäytymisohjatun kehityksen mukaisesti. Tämän jälkeen skenaariokuvaus automatisoidaan ja sen toteutus aloitetaan. Toteutuksessa hyödynnetään yksikkötason käyttäytymisohjattua kehitystä edellä esitetyn syklin mukaisesti. Yksikkötason syklejä jatketaan, kunnes skenaariokuvauksen kuvaama toiminnallisuus on valmis. Tämän jälkeen siirrytään seuraavaan skenaariokuvaukseen, kunnes käyttäjätarinan kaikki skenaariot on käyty läpi eli käyttäjätarinan kuvaama toiminnallisuus on toteutettu. Sykli alkaa alusta, kun uusi käyttäjätarina otetaan

työstettäväksi.

4.3 Määritysten rakenne

Siinä missä skenaariokuvaukset ovat vakiintuneet Given-When-Then -rakenteeseen, ovat yksikkötestit huomattavasti riippuvaisempia ohjelmointikielestä ja käytetystä kirjastosta. Näin ollen skenaarioiden tapainen vapaa ilmaisu ei ole mahdollista. Ohjelmointikielten rajoituksista huolimatta on olemassa yksikkötestaukseen tarkoitettuja kirjastoja, jotka mahdollistavat käyttäytymisohjatun mallin mukaiset testit. Seuraavissa esimerkeissä tarkastellaan Groovy-kielellä toimivaa Spock-testauskirjastoa, joka mahdollistaa Given-When-Then -muotoiset yksikkötestit.

Seuraava ohjelmakoodi esittää yksinkertaisen Java-kielellä tehdyn laskinohjelman. Ohjelma koostuu yhdestä metodista, jolle syötetään kaksi lukua ja laskentaoperaatio. Tuloksena metodi palauttaa laskentatuloksen. Esimerkiksi, jos metodille syötetään luvut yksi ja kaksi sekä operaationa “+”, niin tuloksena metodi palauttaa luvun kolme.

```
 1: public class Laskin {
 2:     public Double laske(double luku1, double luku2, String operaatio) {
 3:         switch (operaatio) {
 4:             case "+": return luku1 + luku2;
 5:             case "-": return luku1 - luku2;
 6:             case "*": return luku1 * luku2;
 7:             case "/": return luku1 / luku2;
 8:         }
 9:         return null;
10:     }
11: }
```

Seuraava Groovy-kielellä kirjoitettu ohjelmakoodi on Spock-kirjastoa hyödyntäen kirjoitettu yksinkertainen yksikkötesti, jolla testataan edellä esitetyn laskinohjelman toimintaa. Aluksi `Given`-lohkossa määritetään kaksi lukua ja käytetty laskentaoperaatio, joka tässä tapauksessa on yhteenlasku. `When`-lohkossa kutsutaan laskimen laskentaoperaatiota, jonka tulos tallennetaan `tulos`-muuttujaan. Lopuksi `Then`-lohkossa tarkistetaan, että `tulos`-muuttujan arvo vastaa odotettua tulosta eli lukua viisi. Luvun perässä oleva `d`-kirjain merkitsee, että kyseessä on double-tyyppinen liukuluku.

```
    def "kaksi plus kolme on viisi"() {
        given:
        double luku1 = 2
        double luku2 = 3
        String operaatio = '+'

        when:
        Double tulos = new Laskin().laske(luku1, luku2, operaatio)

        then:
        tulos == 5d
    }
```

Alla oleva ohjelmakoodi kuvaa toisen tavan tehdä testejä Spock-kirjastolla. Tämän tavan avulla voidaan ohjelman osan toimintaa testata usealla eri syötearvolla. Aluksi `expect`-lohkossa määritetään metodikutsu, jota tullaan toistamaan eri syötteillä. Tässä esimerkissä määritetään laskimen `laske`-metodin kutsu kahdella luvulla ja operaatiolla. Metodikutsun tulos tallennetaan `tulos`-muuttujaan. Seuraavaksi `where`-lohkossa määritetään muuttujien arvot taulukkomuodossa. Ensimmäinen sarake määrittää muuttujan `luku1` arvot, toinen muuttujan `luku2` arvot, kolmas käytettävän operaation eli `operaatio`-muuttujan arvon ja lopuksi neljäs sarake määrittää odotetun tuloksen eli `tulos`-muuttujan arvon. Testiä suoritettaessa `expect`-lohkon metodikutsu tehdään kertaalleen kullekin taulukossa olevalle riville sen rivin muuttuja-arvoja käyttäen.

```
import spock.lang.Specification

class LaskinTesti extends Specification {
    def "LaskimenTulisiLaskeaOperaationMukaisestiKaksiLukua"() {
        expect:
        new Laskin().laske(luku1, luku2, operaatio) == tulos

        where:
        luku1 | luku2 | operaatio | tulos
        1d    | 2d    | '+'       | 3d
        5d    | 4d    | '-'       | 1d
        5d    | -4d   | '-'       | 9d
        3d    | 10d   | '*'       | 30d
        30d   | -10d  | '/'       | -3d
    }
}
```

Spock on yksi monista käyttäytymisohjatun kehityksen mukaisista kirjastoista ja RSpec on vastaava Ruby-kielelle tehty kirjasto. Myös sillä pyritään kuvaamaan ohjelmiston käyttäytymistä, mutta määritysten rakenne ei suoraan vastaa aikaisempia Given-

When-Then -rakenteita. Sen sijaan RSpec-määritykset jaetaan `describe` ja `it` -lohkoihin. Kukin näistä `describe`-lohkoista määrittää yhden pienen ominaisuuden. Kuvauslohkojen sisällä olevat `it`-lohkot vastaavat skenaarioita eli niillä määritetään, kuinka ominaisuus käyttäytyy tietyissä tilanteissa. (Chelimsky et al. 2010, s. 151-153)

Seuraava RSpecillä kirjoitettu testi esittää, kuinka pinon toimintaa voidaan testata. Esimerkin `describe`-lohkoilla määritetään pinotietorakenteen eri ominaisuuksia eli tässä tapauksessa `peek` (arvon tarkastus) ja `pop` (arvon poisto) -toiminnot. `Describe`-lohkojen sisällä on edelleen `it`-lohkot. Esimerkiksi `peek`-toiminnon kohdalla ensimmäisellä `it`-lohkolla tarkastetaan, että toiminto palauttaa viimeisimmän arvon ja toisella `it`-lohkolla tarkistetaan, että toiminto ei poista arvoa pinosta.

```
 1: describe Stack do
 2:     before(:each) do
 3:         @stack = Stack.new
 4:         @stack.push :item
 5:     end
 6:
 7:     describe "#peek" do
 8:         it "should return the top element" do
 9:             @stack.peek.should == :item
10:         end
11:
12:         it "should not remove the top element" do
13:             @stack.peek
14:             @stack.size.should ==1
15:         end
16:     end
17:
18:     describe "#pop" do
19:         it "should return the top element" do
20:             @stack.pop.should == :item
21:         end
22:
23:         it "should remove the top element" do
24:             @stack.pop
25:             @stack.size.should == 0
26:         end
27:     end
28: end
```

Jasmine-kirjastolla tehdyt testit ovat hyvin samankaltaisia RSpec-kirjastolla tehtyjen testien kanssa. Myös Jasmine käyttää describe-it -rakennetta testien määrityksessä. Jasminella testejä voidaan tehdä JavaScriptille, Nodelle, Rubylle ja Pythonille. Näkyvimpänä erona Jasminen ja RSpeckin välillä on se, kuinka odotetun tuloksen tarkistus on toteutettu. Siinä missä RSpec hyödyntää vertailuoperaattoreita, Jasminessa käytetään

`expect`-komennolla alkavia vertailufunktioita (Jasmine 2018). Seuraava ohjelmakoodi esittää yksinkertaisen Jasminella tehdyn esimerkin, jossa testataan laskimen toimintaa eri laskuoperaatioilla.

```
 1: describe("calculator addition", function() {
 2:     it("can add, subtract, multiply, and divide positive integers",
 3:       function() {
 4:         var calc = new Calculator;
 5:         expect(calc.add(2, 3)).toEqual(5);
 6:         expect(calc.sub(8, 5)).toEqual(3);
 7:         expect(calc.mult(4, 3)).toEqual(12);
 8:         expect(calc.div(12, 4)).toEqual(3);
 9:     });
10: });
```

4.4 Ongelmia

Käyttäytymisohjatun kehityksen hyödyntäminen yksikkötasolla ei ole täysin ongelmatonta. Tässä luvussa esitetyt ongelmat eivät kuitenkaan päde ainoastaan yksikkötason käyttäytymisohjattuun kehitykseen vaan yleisesti automatisoituihin yksikkötesteihin.

Yksikkötason testauksen automatisoinnissa on useita samoja ongelmia kuin järjestelmätason automatisoidussa testauksessa. Testien ylläpidettävyyden ja käytettävyyden säilyttäminen on usein vaikeaa. Kuten järjestelmätason testeihin, myös yksikkötesteihin ja niitä varten luotuun testauskehikkoon on suhtauduttava kuin mihin tahansa ohjelmistoprojektiin (Fewster ja Graham 1999, s. 65). Ohjelmakoodin on oltava laadukasta ja helposti ymmärrettävää. Tämä korostuu, kun testejä on hyvin paljon. Testit tulisi tehdä siten, että ne eivät hajoa liian herkästi. Lisäksi testien on oltava luotettavia eli ne eivät saa satunnaisesti antaa esimerkiksi virheellisiä tuloksia. Testien suorituksen on myös oltava nopeaa, jotta kehittäjät voivat suorittaa niitä omassa kehitysympäristössään ja voivat siten saada lähes välittömän palautteen. Tämä rajoittaa sitä, mitä ja miten voidaan testata. Esimerkiksi tietokantayhteydet hidastavat testejä yleensä aivan liikaa (Haikala ja Mikkonen 2011). Hitaat, herkästi hajoavat testit, joiden tarkoitusta on vaikea ymmärtää, vesittävät täysin testien automatisoinnin. Tällöin, kun testejä on paljon, on suurena riskinä, että vanhojen testien ylläpitoon ja uusien tekemiseen kuluu liikaa aikaa ja pahimmassa tapauksessa ei aika riitä varsinaiseen kehitystyöhön.

Kehitysmalli, jossa määritys tehdään ennen toteutusta, vaatii kehittäjiltä huomattavaa kurinalaisuutta. Täyden hyödyn saa todella vain tekemällä määrityksen ennen toteu-

tusta ja pitäytymällä pienissä muutoksissa kehityssyklin edetessä. Kiristyvä aikataulu saa kehittäjät herkästi lipsumaan testiohjatusta kehityksestä. Testien tekeminen ei kuitenkaan myöhäisemmässä vaiheessa ole enää lainkaan niin helppoa kuin asioiden ollessa vielä tuoreessa muistissa. Lisäksi myöhemmin tehdyillä testeillä on ainoastaan testausarvo eivätkä ne auta toiminnallisuuden määrittämisessä ja siten toteutuksessa. (Haikala ja Mikkonen 2011)

Lopulta yksikkötason testit ovat vain osa koko ohjelmiston testausta. Vaikka testejä on hyvin kattavasti, ne eivät testaa kaikkea. Ne eivät anna luotettavaa tietoa järjestelmätason toimivuudesta, kuten eri ohjelmiston osien toimimisesta keskenään tai vuorovaikutuksesta muiden järjestelmien kanssa. Yksikkötestien lisäksi tarvitaan myös järjestelmätason testejä. (Koskela 2008, s. 221-222,334)

5 Skenaariokuvausten automatisoinnin vaikutusten arviointi

Tässä luvussa tarkastellaan, kuinka skenaarioiden automatisointi käyttäytymisohjatun kehityksen mukaisesti vaikuttaa ohjelmistokehityksen eri osa-alueisiin. Vaikutus kohdistuu aina määrittelyvaiheesta toteutukseen, testaukseen ja jopa tekniseen dokumentaatioon.

Automatisoidut skenaariot vaativat tietyn muodon, jossa skenaariot kirjoitetaan määritysvaiheessa. Toteutusvaiheessa automatisointi tehdään ennen varsinaista toteutusta. Näitä ja muita automatisoitujen skenaarioiden vaikutuksia määrityksiin ja toteutukseen tarkastellaan luvussa 5.1.

Kun skenaariot automatisoidaan, samalla luodaan vahvat linkit aina määrityksistä ohjelmakoodiin. Tämän linkityksen hyödyntämistä teknisessä dokumentaatiossa tarkastellaan luvussa 5.2. Samaa linkitystä voidaan hyödyntää myös ohjelmistoprojektin edistymisen seurannassa, johon tutustutaan tarkemmin luvussa 5.3.

Automatisoidut skenaariot ovat tehokas tapa testata ohjelmistoa automaattisesti. Automatisoitujen skenaarioiden vaikutusta testaukseen tarkastellaan luvussa 5.4. Lopuksi luvussa 5.5 tutustutaan kuinka automatisoidut skenaariot vaikuttavat yksikkötason kehitykseen ja testaukseen.

5.1 Vaikutus määritysten ja ohjelmakoodin tekemiseen

Käyttäytymisohjattu kehitys painottaa määritysten tekoa yhdessä yhteisellä kielellä. Tätä tarkasteltiin lähemmin luvussa 2. Yhteisellä kielellä vähennetään väärinymmärryksen riskiä ja yhteisellä määrityksellä tavoitellaan kerralla kattavaa, kaikki tilanteet huomioivaa, määritystä. Määrityksen kirjaukseen vaikuttaa vahvasti se, millä menetelmillä mahdollinen automatisointi tullaan tekemään myöhemmissä ohjelmiston kehitysvaiheissa. Esimerkkinä luvussa 3 esiteltiin Cucumber-JVM, jossa käytetään Oletetaan-Kun-Niin -muotoa. Tällöin myös määritykset kannattaa kirjata suoraan tälle muodolle, jotta niitä ei tarvitse myöhemmin muuntaa toiseen muotoon: muunnoksissa kun on aina riskinä, että merkitys muuttuu.

Käytetyt työkalut vaikuttavat myös määritysten organisointiin. Cucumber-JVM:ssä määritykset jaotellaan ominaisuuksiksi siten, että yksi ominaisuus sisältää useita skenaarioita. Kehitystyötä sen sijaan tehdään yleensä käyttäjätarinoiden avulla, jolloin yksi ominaisuus katetaan yhdellä tai useammalla käyttäjätarinalla. On kuitenkin hyvä muistaa, että käyttäjätarinat ovat vain kehityksen aikaisia välineitä, jotka yleensä hävitetään kehityksen päätteeksi. Sen sijaan ominaisuudet ja skenaariot ovat pysyviä ja jäävät elämään ohjelmiston muutosten mukana.

Varsinainen toteutus tehdään testiohjatusti skenaario kerrallaan. Toteutusvaiheen alussa kehittäjä tekee yhdelle skenaariolle tarvittavan automatisoinnin. Vasta tämän jälkeen kehitetään varsinainen toteutus, jolla skenaario saadaan ajautumaan läpi. Toteutus itsessään tehdään yksikkötestiohjatusti ja se on skenaarion osalta valmis, kun kaikki automatisoidut skenaariot suoriutuvat läpi virheittä.

Ohjelmistokehityksessä on tavallista, että kesken projektin havaitaan ongelmia, joiden vuoksi ohjelmistoa joudutaan tekemään osittain uusiksi. Voidaan esimerkiksi havaita tekninen este, jonka ylittäminen vaatii uudenlaisen toteutuksen tai toteutuksessa ei ole huomioitu jotakin mahdollista skenaariota ja uuden skenaarion lisääminen aiheuttaa suuria muutoksia ohjelmakoodiin. Käyttäytymisohjattu kehitys taistelee osaltaan muutostarvetta vastaan yhdessä tuotetuilla skenaariokuvauksilla. Jos muutostarve kaikesta huolimatta tulee, on muutoksen hallinta käyttäytymisohjatussa kehityksessä vaikeampaa automatisoitujen skenaarioiden vuoksi. Suurempi työmäärä johtuu hajautetusta rakenteesta, kun skenaariot, vaihemääritykset ja ohjelmakoodi ovat eriytetty toisistaan (Borg ja Kropp 2011).

Vanhan ominaisuuden poistaminen tai muokkaaminen ja uuden ominaisuuden lisääminen vaikuttavat usein tavalla tai toisella sekä skenaarioihin, vaihemäärityksiin että ohjelmakoodiin. Ohjelmakoodin ja vaihemääritysten refaktorointi on työlästä, koska molemmat on tehtävä erikseen ja tähän tarkoituksen ei ole olemassa työkaluja kuten on yksikkötestien refaktoroinnille (Rahman ja Gao 2015). Jos skenaariota muutetaan, on testikoodi muutettava vastaamaan sitä ja edelleen ohjelmakoodi muutettava vastaamaan testikoodia. Myös ohjelmakoodin muuttaminen tarkoittaa usein testikoodin päivittämistä, vaikka toiminnallisuus pysyisi entisellään. Tämä ongelma korostuu entisestään, jos testit vuorovaikuttavat graafisen käyttöliittymän kautta testattavaan ohjelmaan: tällöin pienetkin muutokset käyttöliittymässä rikkovat testit.

Suoraa ratkaisua refaktorointityömäärän lisääntymiseen ei ole. Käyttäytymisohjattua

kehitystä tukevat refaktorointityökalut saattaisivat olla ratkaisu. Kuitenkin luultavasti tehokkain tapa on jo aiemmin luvussa 3.1 mainittu sopiva automaatiokerroksen arkkitehtuuri. Tällöin testit eivät vaikuta suoraan itse ohjelman kanssa muissa, kuin triviaaleissa tapauksissa. Näin ollen muutokset ohjelmakoodiin aiheuttavat muutospainetta pääasiassa automaatiokerroksen alimpiin osiin, joita lukuisat testit voivat hyödyntää. Muutostyö jää tällöin huomattavasti pienemmäksi kuin jos suuri määrä testejä pitäisi päivittää.

5.2 Vaikutus dokumentaatioon

Skenaariot ovat tärkeä osa dokumentaatiota. Skenaariot säilyvät toteutuksen jälkeenkin, vaikka kehitystyön apuna olleet käyttäjätarinat hylätään hankkeen päätteeksi. Automatisoinnin myötä skenaariokuvaukset linkittyvät testikoodiin, joka edelleen linkittyy varsinaiseen ohjelmiston lähdekoodiin. Näin skenaariokuvaukset pysyvät melko helposti ajan tasalla. Perinteisesti ohjelmistoa varten laaditaan teknisiä dokumentaatioita, jotka kehittäjien on muistettava päivittää aina, kun ohjelmistoon tehdään muutoksia. Tällaiset dokumentaatiot saattavat jäädä päivittämättä tai osa tärkeistä tiedoista unohdetaan, jos päivitys muistetaan tehdä vasta myöhemmin hankkeen aikana. Automatisoiduilla skenaarioilla tämä ongelma pitkälti väistyy, kun ohjelmistomuutokset tehdään pääasiassa skenaariomuutosten ja testikoodin muutosten pohjalta. (Smart 2015, s. 302-304)

Kun skenaarioita automatisoidaan kattavasti, ovat ne tehokas tekninen dokumentaatio. Niistä selviää, kuinka ohjelmisto käyttäytyy eri tilanteissa. Lisäksi testauskerroksen avulla voidaan tarkastella, kuinka ohjelmiston kanssa vuorovaikutetaan. Testauskoodi sisältää runsaasti käytännön esimerkkejä vuorovaikutuksesta ohjelmiston kanssa. Lopuksi linkit testauskerroksesta ohjelmakoodiin johdattavat kehittäjän niihin osiin koodia, joissa ominaisuus on käytännössä toteutettu. (Smart 2015, s. 302-304)

Elävä dokumentaatiokaan ei ole aukoton tapa tehdä dokumentaatiota. Mikään muu kuin kurinalaisuus ei estä kehittäjiä tekemästä ohjelmistomuutoksia ensin ja vasta sen pohjalta testikoodin muutoksia. Huonoimmassa tapauksessa skenaariot muunnetaan automaattisiksi testeiksi vasta jälkikäteen ohjelmakoodin toteutuksen jälkeen. Tällöin inhimillisen erehdyksen riski voi olla suurempi ja kaikkia muutoksia ei tule dokumentoitua skenaarioiksi.

Skenaarioiden on oltava selkeitä, jotta niiden merkitys ymmärretään myös myöhemmin. Epäselvät skenaariot eivät auta ketään ymmärtämään ohjelmiston toimintaa. Selkeyden takaamiseen auttaa, kun skenaarioista ei tehdä liian pitkiä. Turhat yksityiskohdat eivät auta skenaarion ymmärtämistä, joten ne kannattaa piilottaa ja keskittyä olennaiseen asiaan. Lisäksi yhdessä skenaariossa kannattaa keskittyä vain yhteen asiaan kerrallaan. Toisaalta myös liian suuri määrä skenaarioita voi olla ongelma. Jos samassa ominaisuustiedostossa on paljon skenaarioita, voi olla vaikea hahmottaa, kuinka ne liittyvät toisiinsa. Tällöin myös vaihemääritystiedostot paisuvat helposti liian pitkiksi. (Adzic 2011)

Elävä dokumentaatio voidaan koostaa tarpeen mukaan. Esimerkiksi kehitysvaiheessa, kun työstetään yksittäisiä käyttäjätarinoita, saattaa olla hyötyä organisoida myös tekninen dokumentaatio käyttäjätarinoiden mukaan. Myöhemmin, kun ominaisuudet ovat valmiita, on hyödyllisempää organisoida dokumentaatio toiminnallisten alueiden mukaan. Dokumentaation pitäisi myös olla helposti saatavilla, jotta sitä todella tulisi käytettyä. (Adzic 2011, s. 191-192)

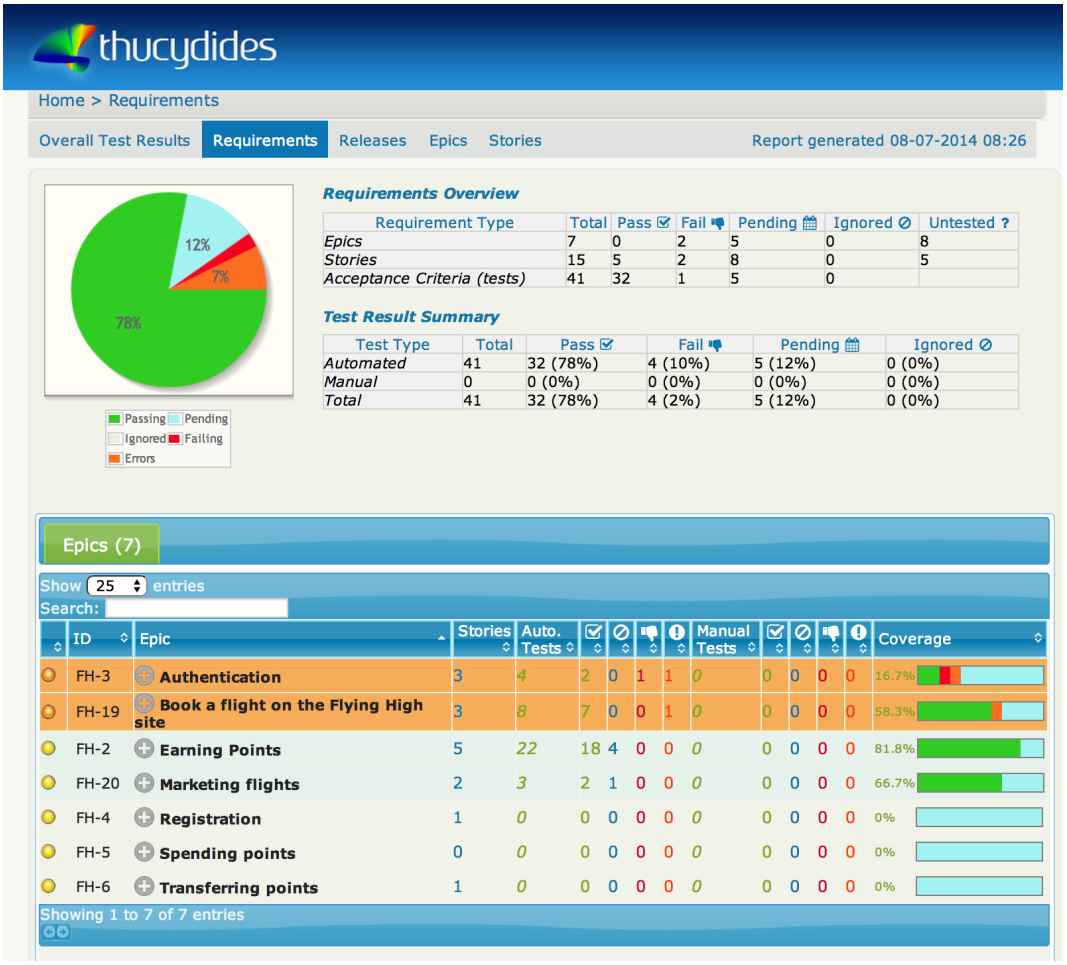
5.3 Vaikutus projektin edistymisen seurantaan

Elävän dokumentaation tapaan automatisoidut skenaariot mahdollistavat myös projektin edistymisen seurannan. Yksinkertaisimmillaan voidaan tarkkailla, minkä ominaisuuden testit ovat työn alla. Kuitenkin paremman kuvan edistymisestä saa, mikäli raportointityökalut kykenevät ilmoittamaan kokonaisten ominaisuuksien tilan. Edistymisen seurannassa kun yleensä ollaan kiinnostuneita kokonaisten ominaisuuksien valmiudesta (Smart 2015, s. 304).

Seuranta perustuu siihen, että automatisoimatta olevat skenaariot ovat keskeneräisiä. Näin ollen, jos ominaisuudella on keskeneräisiä skenaarioita, on myös ominaisuus keskeneräinen. Täysin tarkkaa käsitystä ohjelmistoprojektin edistymisestä ei tällä tavalla ole kuitenkaan mahdollista saada, koska voi olla niin, että raportin mukaan ominaisuuden kaikki skenaariot on automatisoitu, mutta todellisuudessa samaan ominaisuuteen ollaan tekemässä vielä lisää skenaarioita (Smart 2015, s. 307-308). Tosin kaikkia skenaarioita ei olekaan tarkoitus automatisoida, joten ne on jotenkin voitava merkitä työkalussa valmiiksi, jotta raportit vastaavat todellisuutta.

Kuva 7 kuvaa, kuinka Serenity-työkalu mahdollistaa edistymisen seurannan. Tässä ta-

pauksessa edistymistä seurataan laajempina epic-tason käyttäjätarinoina ja niiden sisältäminä tarkempina käyttäjätarinoina. Näkymä mahdollistaa pikaisen yleiskatsauksen projektin edistymisestä. Edistymisen seuranta on hyödyllistä myös testaajille, jotka näkevät suoraan raportista ominaisuuksien tilan ja voivat siitä päätellä, mitä ominaisuuksia kannattaa ryhtyä testaamaan tutkivan testauksen keinoin (Smart 2015, s. 303).



Kuva 7: Ominaisuuksien valmius raportoituna Serenity-työkalussa. (Smart 2018)

5.4 Vaikutus toiminnalliseen testaukseen ja regressiotestaukseen

Suoritettavat skenaariot ovat niin kutsuttuja hyväksymistestejä, joilla kehitysvaiheessa voidaan varmistaa, että toteutus vastaa odotuksia (Leffingwell 2010, s. 77). Kurinalainen käyttäytymisohjatun kehityksen noudattaminen mahdollistaa hyvin kattavan joukon automaattisia hyväksymistestejä. Koska kaikki määritys tehdään skenaarioiden avulla ja kaikki skenaariot automatisoidaan, on teoriassa mahdollista saavuttaa täysi

kattavuus ohjelmiston ohjelmakoodin suorituspoluista. Käytännössä tämä ei kuitenkaan ole mahdollista vaan vaikuttaa siltä, että projektin edetessä testien kattavuus vähitellen laskee. Yksi syy on ilmeisesti se, etteivät kehittäjät kykene huomioimaan testeissä kaikkia mahdollisia ohjelmakoodin suorituspolkuja, jolloin osa ohjelmakoodista jää väistämättä testien ulkopuolelle (Diepenbeck et al. 2013).

Kattavuuden lisäksi toinen testauksen kannalta oleellinen asia on suuri automaattisten testien määrä. Tämä vapauttaa testaajat tekemään enemmän laadullista testausta, johon ei ole olemassa hyviä automaattisen testauksen menetelmiä. Testejä voidaan myös suorittaa hyvin usein, mahdollisesti jopa jokaisen muutoksen jälkeen. Tällöin saadaan nopeasti tieto siitä, jos jokin ohjelmistoon tehty muutos on vaikuttanut odottamattomasti muihin ominaisuuksiin. Kun aiheuttaja on helposti selvitettävissä, tarvittavat korjaukset voidaan tehdä nopeasti.

Ketterissä menetelmissä nopea testaus korostuu, kun kehitystyötä tehdään lyhyissä jaksoissa. Kun tavoitteena on, että jakson päätteeksi ohjelmisto on potentiaalisesti julkaisukelpoinen, on myös oltava melko hyvä varmuus siitä, että se toimii oikein. Tämä varmuus pyritään saavuttamaan jatkuvan regressiotestauksen avulla. Regressiotestauksessa testataan tietyllä suppealla testijoukolla ne ohjelmiston osa-alueet, joihin on tehty muutoksia. Vaikka testijoukko pidetään suppeana, on manuaalinen testaus silti hyvin työlästä. Sen sijaan, jos automaattisia testejä on kattavasti, voidaan koko ohjelmisto testata useaan kertaan kehitysiteraation aikana ja saada hyvä ymmärrys ohjelmiston valmiudesta.

Automatisointi tuo myös ongelmia ja kaikkea testausta ei voi tai kannata automatisoida. Manuaalinen testaus kannattaa esimerkiksi laadullisessa testauksessa, joissa ihminen on tietokoneeseen nähden ylivoimainen. Lisäksi automatisointi on hidasta ja vaatii aina enemmän aikaa kuin manuaalisten testien määritys.

Myös suuren automaattisten testien joukon ylläpito käy nopeasti mahdottomaksi, jos automatisointirajapintaa ei ole huolella suunniteltu ylläpidon näkökulmasta. Esimerkiksi muutokset määrityksiin tai ohjelmakoodin uudelleenjärjestely ja refaktorointi saattaa olla työlästä, jos suuri joukko automatisoituja skenaarioita liittyy samaan ohjelmakoodin toteuttamaan toiminnallisuuteen (Rahman ja Gao 2015).

5.5 Vaikutus yksikkötason kehitykseen ja testaukseen

Kuten luvussa 4 esitettiin, voi käyttäytymisohjatun kehityksen menetelmiä hyödyntää rajoitetusti myös yksikkövetoisessa kehityksessä. Tällöin eroavaisuus on kuitenkin lähinnä terminologiassa. Ennen kuin North sai ensimmäiset oivallukset käyttäytymisohjatusta kehityksestä, yritti hän löytää keinoja, joilla yksikköohjatusta kehityksestä saataisiin tehtyä helpommin lähestyttävä ja omaksuttava menetelmä. Hänen ajatuksenaan oli vaihtaa testi-sana käyttäytyminen-sanaan. Tällöin testien asemasta tehtäisiinkin määrityksiä, joilla kuvataan käyttäytymistä. (North 2006)

Myöhemmin uudet yksikkötestikirjastot ovat vieneet tätä ajatusta pidemmälle ja esimerkiksi Spock-kirjasto mahdollistaa hyvin pitkälle Give-When-Then -tyylisen ilmaisumuodon käyttäytymisohjatun kehityksen hengessä. Ero perinteiseen yksikkötestien toteutustapaan on kuitenkin lopulta vain syntaksissa, joten lopulta hyödyt saattavat jäädä melko pieniksi. Verrattuna siihen, mitä käyttäytymisohjattu kehitys tuo toiminnalliseen testaukseen, jossa se painottaa automaattisten testien lisäksi vahvasti yhteistyötä ja kommunikaatiota, ovat yksikkötestiohjatun kehityksen parannukset varsin pieniä.

6 Yhteenveto ja pohdinta

Tässä opinnäytetyössä tarkasteltiin skenaarioiden automatisoinnin vaikutuksia käyttäytymisohjatussa kehityksessä. Tutkielmassa tutustuttiin käyttäytymisohjattuun kehitykseen saatavilla olevan kirjallisuuden pohjalta. Päälähteenä hyödynnettiin Abigail Egbreghtsin tekemää kirjallisuuskatsausta käyttäytymisohjatusta kehityksestä (Egbreghts 2017). Kirjallisuuskatsauksessa Egbreghts tutkii käyttäytymisohjatun kehityksen määritelmää useiden eri kirjallisuuslähteiden pohjalta. Lähteet on kerätty seuraavista tietokannoista: IEEE, ScienceDirect, ACM, Web of Science, Scopus ja Google Scholar. Kirjallisuuden pohjalta Egbreghts määrittää Käyttäytymisohjattuun kehitykseen liittyvät eri osa-alueet, jotka ovat määrityksissä käytetty kieli, määritykset, hyväksymistestit, työkalut, yhteistyö ja testien automatisointi.

Tutkielma jaettiin neljään osaan, jotka olivat johdanto käyttäytymisohjattuun kehitykseen, skenaarioiden kirjoitusmuoto ja automatisointi käytännössä, käyttäytymisohjattu kehitys yksikkötasolla ja skenaarioiden automatisoinnin vaikutukset ohjelmistotuotannon eri osa-alueisiin.

Johdannossa käyttäytymisohjattuun kehitykseen, eli luvussa 2, perehdyttiin ketterissä menetelmissä käytettyihin käyttäjätarinoihin ja niiden hyödyntämiseen laadittaessa uusia ominaisuuksia ohjelmistoon. Käyttäjätarinat ovat kehitystyössä käytetty väline, jolla suuret ominaisuudet saadaan pilkottua pienemmiksi, helpommin hallittaviksi, ymmärrettäviksi ja toteutettaviksi palasiksi. Ketterissä menetelmissä pyritään välttämään liiallista suunnittelua, joten käyttäjätarinoita tarkennetaan oikea-aikaisesti tarpeen niin vaatiessa. Tarkennus aloitetaan määrittelyvaiheessa, jolloin pyritään saamaan kokonaiskuva toteutuksesta ja riittävä ymmärrys ominaisuuksista, jotta ne osataan asettaa tärkeysjärjestykseen. Tarkennusta jatketaan edelleen kehitysvaiheessa, kun tarvitaan tarkka ymmärrys siitä, mitä käyttäjätarinan puitteissa käytännössä tulisi toteuttaa.

Käyttäytymisohjatussa kehityksessä käytetään skenaarioita käyttäjätarinoiden tarkentamisessa. Skenaariot ovat käytännönläheisiä esimerkkejä, jotka kuvaavat, kuinka ohjelmiston odotetaan käyttäytyvän tietyissä tarkasti määritetyissä tilanteissa. Näin määrittelyyn osallistuvat henkilöt pääsevät konkreettisten esimerkkien avulla keskustelemaan ohjelmiston toiminnallisuudesta.

Ketterän kehitysmallin käyttäminen ei ole edellytys käyttäytymisohjatun kehityksen hyödyntämiselle. Periaatteessa samoja käyttäytymisohjatun kehityksen periaatteita

voitaisiin hyödyntää suunnitelmaperustaisissa kehitysmalleissa, kuten vesiputousmallissa. Suurin osa käyttäytymisohjatusta kehityksestä tiivistyy kuitenkin skenaarioiden määritykseen ja niiden automatisointiin. Periaatteessa skenaariot voitaisiin määrittää suunnitelmaperustaisen mallin määritysvaiheessa. Kuitenkin kirjallisuudessa käyttäytymisohjattu kehitys esitetään yleensä osana ketterää kehitystä.

Skenaarioiden on oltava määrityksinä riittävän väljiä. Niiden ei tule ottaa kantaa varsinaiseen toteutukseen tai teknisiin ratkaisuihin. Sen sijaan ne kuvaavat tavoitteet ja halutun käyttäytymisen, joiden pohjalta tekniset ratkaisut valikoidaan myöhemmin varsinaisessa toteutuksessa.

Käyttäytymisohjattu kehitys ei varsinaisesti vaadi skenaariokuvauksilta mitään tiettyä rakennetta tai muotoa. Kuitenkin vakiintunut tapa on käyttää Given-When-Then, eli Oletetaan-Kun-Niin -rakennetta. Tässä muodossa skenaario muodostuu vaiheista, joista kukin alkaa joko oletetaan-, kun- tai niin-sanalla. Oletetaan-lauseella ilmaistaan skenaarion alkutilaa, kuten “Oletetaan, että pankkiautomaatista saa 20:n ja 50:n euron seteleitä”. Kun-lauseella ilmaistaan jotakin tapahtumaa tai toimintaa, esimerkiksi “Kun asiakas nostaa 70€”. Niin-lauseella sen sijaan ilmaistaan odotettua lopputulosta, esimerkiksi “Niin automaatti antaa 20:n ja 50:n euron setelit”. Lisäksi skenaarioissa voidaan käyttää ja-sanalla alkavia lauseita korvaamaan oletetaan, kun ja niin -sanoja silloin, kun niitä on useita peräkkäin. Näin skenaarion kieli on luonnollisempaa.

Käyttäytymisohjatussa kehityksessä kannustetaan määrittämään skenaariot yhteistyönä eri osapuolten kesken. Määrittelyssä mukana tulisi olla vähintään asiakkaan, kehittäjän ja testaajan edustajat. Tällä tavalla saadaan laajalti katettua eri näkökulmat. Asiakas tietää parhaiten, mitä ohjelmistolla halutaan saavuttaa, mutta tekninen näkemys on yleensä puutteellinen. Kehittäjällä sen sijaan on kattava tietämys teknisistä mahdollisuuksista ja rajoitteista. Testaaja sen sijaan osaa ajatella parhaiten mahdollisia rajatapauksia, jotka muuten saattaisivat jäädä huomaamatta.

Skenaariokuvaukset ovat tärkeä osa kommunikaatiota ja niitä hyödyntävät monet eri sidosryhmien jäsenet kuten asiakas, kehittäjät, testaajat, projektipäälliköt ja analyytikot. Kuten luvussa 2 todettiin, on suuri merkitys sillä, millaista sanastoa ja kieltä projektissa ja skenaariokuvauksissa käytetään. Tämän vuoksi projektin aikana on tärkeä kiinnittää huomiota luonnollisen, yhtenäisen ja yksiselitteisen sanaston keräämiseen. Käytetyn kielen on hyvä olla lähellä tavallisesti kommunikoinnissa käytettyä kieltä, jotta sen käyttö olisi luontevaa ja helppoa. Kuitenkin kielen on oltava yksiselitteistä eli

yhdestä asiasta käytetään vain yhtä ilmausta ja samoja ilmaisuja ei käytetä eri asioista. Näin vältytään väärinymmärryksiltä.

Tutkielman toisessa osassa käytiin taarkemmin läpi skenaariokuvausten rakennetta sekä sitä, kuinka skenaariot käytännössä automatisoidaan. Luvussa 3 tutustuttiin, kuinka skenaariot tulee määrittää, jotta automatisointi olisi mahdollista Cucumber-JVM testauskehikon kanssa. Cucumber-JVM:n kanssa skenaariot kirjoitetaan tekstitiedostoon Gherkin-kielellä, jossa käytetään Oletetaan-Kun-Niin -rakennetta. Jotta automatisointi olisi mahdollista, tarvitaan jokaista skenaarion Oletetaan, Kun ja Niin -vaihetta varten ohjelmointikielellä määritetty vaihekuvaus. Vasta vaihekuvauksen tehtävänä on varsinaisesti vuorovaikuttaa testattavan ohjelmiston kanssa. Skenaariota suoritettaessa Cucumber-JVM -testauskehikon tehtävänä on tulkita skenaariota vaiheittain ja kunkin vaiheen kohdalla suorittaa vaihetta vastaava vaihekuvaus, joka suorittaa skenaarion määrittämän toiminnon testattavassa ohjelmistossa.

Yleensä ei ole järkevää, että vaihekuvaukset ovat suorassa vuorovaikutuksessa testattavan ohjelmiston kanssa. Parempi ratkaisu on tehdä väliin erillinen automaatiorajapinta, jonka tehtävänä on tehdä testien ylläpidosta helpompaa. Tällöin, vaikka ohjelmistoon tehty muutos rikkoisi useita testejä, riittää pelkän automaatiorajapinnan korjaus ja jokaista testiä ei tarvitse korjata erikseen. Erityisesti automatisoinnin arkkitehtuuri korostuu, kun testit kohdistuvat suoraan käyttöliittymään. Käyttöliittymä on usein muutosten alla ja jo pienet muutokset saattavat rikkoa useita testejä.

Automatisoidut skenaariot mahdollistavat parametrien hyödyntämisen. Parametrisoinnilla osa skenaarion vaiheen sisältämästä tiedosta voi olla parametreina, jotka automaatiokehikko lukee kuvauksesta ja syöttää suoritettavaan vaihekuvaukseen. Tämä tekee skenaarioista joustavampia, sillä esimerkiksi useampi skenaario voi käyttää samoja vaihekuvauksia eri parametreilla. Dataa voidaan syöttää myös taulukkomuodossa, jolloin yksittäisille vaiheille voidaan syöttää useita arvoja taulukossa tai koko skenaario voidaan suorittaa lukuisia kertoja taulukon eri arvoilla. Taulukoiden avulla voidaan joissakin tilanteissa korvata useita yksittäisiä skenaarioita.

Tutkielman kolmannessa osassa, luvussa 4, tutustuttiin, kuinka käyttäytymisohjatun kehityksen periaatteet soveltuvat yksikkötasolla tapahtuvan testauksen automatisointiin. Yksikkötasolla käyttäytymisohjattu kehitys on hyvin lähellä perinteistä yksikköohjattua kehitystä. Automaatiossa voidaan jopa hyödyntää samoja xUnit-testauskehikoita. Toisaalta tarjolla on myös käyttäytymisohjatun kehityksen periaat-

teisiin paremmin sopivia työkaluja. Tällaisina mainittiin Spock, RSpeck ja Jasmine -kirjastot. Spock hyödyntää Gherkin-kielen kaltaista Given-When-Then -rakennetta, kun taas RSpec ja Jasmine käyttävät describe-it rakennetta. Molemmissa tavoissa on kuitenkin tarkoitus kannustaa käyttäytymisen kuvaamiseen sen sijaan että määritettäisiin suoraan testejä.

Tutkielman viimeisessä osassa, luvussa 5, tutustuttiin, kuinka skenaarioiden automatisointi vaikuttaa ohjelmistotuotannon eri vaiheisiin. Skenaariot kulkevat läpi koko tuotannon määrityksistä toteutukseen ja testaukseen sekä myös ylläpitoon asti. On siis luonnollista, että automatisoidut skenaariot vaikuttavat tavalla tai toisella kaikkiin näihin vaiheisiin. Määritysvaiheessa skenaariot ovat kommunikaatioväline, joka auttaa halutun ohjelmiston käyttäytymisen määrittämisessä. Kuitenkin, jotta skenaariot olisivat myöhemmin automatisoitavissa helposti, on niissä käytettävä tiettyä rakennetta. Rakenne määräytyy sen mukaan, mitä automatisointikirjastoa hyödynnetään. Esimerkiksi Cucumber-JVM ja JBehave vaativat tietyn hieman toisistaan poikkeava syntaksin. Näiden lisäksi on olemassa myös esimerkiksi Fitnesse-työkalu, jolla määritykset ja automatisointi tehdään wiki-tyyppisessä ympäristössä.

Toteutusvaiheessa käyttäytymisohjattu kehitys noudattaa testivetoista kehitysmenetelmää eli ensin tehdään automaattinen testi ja vasta sen pohjalta varsinainen toteutus. Automaattisen testin tekeminen on käytännössä skenaarioon liittyvän vaihemäärityksen ohjelmointia. Vaihekuvausten pohjalta lähdetään ohjelmoimaan sopivaa, käyttäytymisen toteuttavaa ohjelmakoodia. Toteuttava ohjelmakoodi voidaan tehdä yksikkötestiohjattuna kehityksenä, jossa voidaan myös hyödyntää käyttäytymisohjatun kehityksen menetelmiä.

Yksikkötestiohjattu kehitys toimii siis käyttäytymisohjatun kehityksen sisällä siten, että skenaarion kuvaama toiminnallisuus saadaan aikaan iteratiivisella kehityksellä usean yksikkötestin kautta, kunnes automaattinen skenaario suoriutuu virheittä läpi. Tämän jälkeen voidaan siirtyä toteuttamaan seuraavaa skenaariokuvausta. Tätä sykliä jatketaan, kunnes kokonainen skenaarioista koostuva ominaisuus on valmis.

Automatisoiduilla skenaarioilla on huomattava merkitys myös ohjelmiston testauksessa. Kun automatisointi on tehty huolella ja riittävällä kattavuudella, vapauttaa se testaajat tekemään enemmän tutkivaa ja laadullista testausta. Nämä osa-alueet ovat sellaisia, joita ei voida järkevästi automatisoida, mutta ihmiselle ne ovat varsin helppoja testattavia. Onkin huomattavan tärkeää, että testauksen automatisointi keskitetään sinne,

missä sillä saavutetaan suurin hyöty.

Ketterissä menetelmissä, joissa pyritään joka kehitysiteraation päätteeksi saamaan aikaiseksi potentiaalisesti julkaisukelpoinen tuote, on testauksella hyvin suuri merkitys. Jos testaus jouduttaisiin aina tekemään manuaalisesti, saattaisi se nopeasti käydä mahdottomaksi yksinkertaisesti testien suuresta määrästä johtuen. Sen vuoksi testauksen automatisointi korostuu ketterissä menetelmissä. Automaattiset testit voidaan suorittaa suhteellisen nopeasti ja hyvin tiuhaan, parhaimmillaan jokaisen pienen muutoksen jälkeen. Jos testien suoritus on hidasta, voidaan ne yleensä suorittaa yön aikana, jolloin aamulla saadaan tieto ohjelmiston tilasta.

Usein suoritettavia automatisoituja skenaarioita voidaan hyödyntää myös suhteellisen luotettavana tapana seurata ohjelmiston kehityksen edistymistä. Kun tiedetään, mihin käyttäjätarinoihin kukin skenaario kuuluu, voidaan tehdä raportteja, joista nähdään, kuinka suuri osa käyttäjätarinoista on valmiita. Esimerkiksi Serenity on raportointityökalu, joka mahdollistaa tällaisen seurannan.

Automatisoidut skenaariot voivat auttaa myös teknisen dokumentaation kasaamisessa. Skenaariokuvaukset kertovat tarkasti, kuinka ohjelmiston odotetaan käyttäytyvän eri tilanteissa. Skenaarion vaiheisiin linkitetyt vaihemääritykset sen sijaan kuvaavat, kuinka ohjelmiston kanssa vuorovaikutetaan. Vaihemäärityksestä on myös vahva linkki ohjelmiston osaan, jossa varsinainen toiminta tapahtuu. Näin ollen skenaariokuvauksesta on luotettava linkki aina ohjelmiston lähdekoodiin asti. Dokumentaationa automatisoidut skenaariot ovat myös varsin luotettavia. Ne päivittyvät luonnollisesti kehityksen aikana, koska muutokset skenaarioihin heijastuvat aina ohjelmiston lähdekoodiin asti.

Käsin kirjoitettu lisädokumentaatio voi olla huomattavasti kevyempää, kun kaikkia teknisiä yksityiskohtia ei tarvitse kattaa. Sen sijaan dokumentaatio voi keskittyä selventämään ohjelmiston vaikeasti ymmärrettäviä osia ja kuvaamaan niiden arkkitehtuuria.

Kokonaisuutena käyttäytymisohjattu kehitys, jossa hyödynnetään automatisoituja skenaarioita, vaikuttaa suureen osaan ohjelmistokehityksen vaiheita. Se lupaa parantaa ohjelmiston laatua, mutta tutkielmaa tehdessä ei löytynyt yhtään tutkimusta, joka olisi selvittänyt käyttäytymisohjatun kehityksen vaikutusta ohjelmiston laatuun. Tässä olisikin aihetta lisätutkimukselle. On verraten todennäköistä, että menetelmä vaikuttaa positiivisesti laatuun useissa eri ohjelmistokehityksen vaiheissa. Skenaariokuvaukset ovat selkeä menetelmä kuvata ohjelmiston käyttäytymistä eikä niitä tarvitse muuttaa

erillisiksi testeiksi testausta varten, kuten yleensä ohjelmistotuotannossa käytettyjen määritysten kanssa toimitaan. Toteutuksessa automaattinen skenaario määrittää varsinaisen toteutuksen, jolloin toteutus todella tuottaa halutun käyttäytymisen. Tämä toki vaatii kehittäjiltä kurinalaisuutta, jotta skenaario todella automatisoidaan ennen varsinaista toteutusta. Skenaarioiden automatisointi vapauttaa testaajat tekemään enemmän laadullista testausta, jolloin mahdolliset ongelmat havaitaan paremmin. Oletuksena on tietenkin se, ettei ihmistestaajia korvata täysin automaattisilla testeillä.

Käyttäytymisohjatulla kehityksellä on myös ongelmakohtansa. Skenaariokuvaukset eivät välttämättä sovellu aivan kaikkeen määritystyöhön, vaan tukena joudutaan toisinaan hyödyntämään myös muita menetelmiä, kuten käyttötapauskaaviota. Myös skenaarioiden automatisointi on huomattavan hidasta samalla tavoin kuin testauksen automatisointi yleensä. Se on tehtävä riittävällä huolellisuudella ja suunnitelmallisuudella, jotta se ei ajaudu tarkoitustaan vastaan. Huonosti suunniteltu testaus saattaa koitua mahdottomaksi ylläpitää. Pienet muutokset ohjelmistoon saattavat rikkoa suuren määrän testejä, joiden korjaaminen voi huonon suunnittelun tuloksen olla lähes mahdotonta kohtuullisessa ajassa. Jos käyttäytymisohjattu kehitys todella lisää ohjelmiston laatua, ei se tule ilmaiseksi. Menetelmä vaatii kurinalaisuutta ja suunnitelmallisuutta, jotta mahdolliset hyödyt voidaan saavuttaa.

Viitteet

Wiegers, K. ja J. Beatty (2013). *Software Requirements*. Developer Best Practices. Pearson Education. ISBN: 9780735679627.

Williams, Laurie, E. Michael Maximilien ja Mladen Vouk (2003). ”Test-driven development as a defect-reduction practice”. Teoksessa: *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. IEEE, s. 34–45.

Adzic, G. (2011). *Specification by Example: How Successful Teams Deliver the Right Software*. 2011499864. Manning. ISBN: 9781617290084.

Baillon, Christophe ja Shanti Bouchez-Mongardé (2010). ”Executable requirements in a safety-critical context with Ada”. *Ada User Journal* 31.2, s. 131–135.

Bechtold, Stefan et al. (2018). *JUnit5*. URL: `https://junit.org/junit5/docs/current/user-guide/` (viitattu 08. 05. 2018).

Beck, Kent (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Beck, Kent (2012). *Why does Kent Beck refer to the "rediscovery"of test-driven development?* URL: `https://www.quora.com/Why-does-Kent-Beck-refer-to-the-rediscovery-of-test-driven-development-Whats-the-history-of-test-driven-development-before-Kent-Becks-rediscovery` (viitattu 25. 05. 2018).

Borg, Rodrick ja Martin Kropp (2011). ”Automated acceptance test refactoring”. Teoksessa: *Proceedings of the 4th Workshop on Refactoring Tools*. ACM, s. 15–21.

Chelimsky, David et al. (2010). *The RSpec book: Behaviour driven development with Rspec, Cucumber, and friends*. Pragmatic Bookshelf.

Cucumber limited (2017). *Cucumber dokumentaatio*. URL: `https://cucumber.io/docs/reference` (viitattu 09. 09. 2017).

Cucumber limited (2018). *Cucumber lähdekoodi*. URL: `https://github.com/cucumber/cucumber-jvm` (viitattu 09. 05. 2018).

Diepenbeck, Melanie et al. (2013). ”Towards automatic scenario generation from coverage information”. Teoksessa: *Automation of Software Test (AST), 2013 8th International Workshop on*. IEEE, s. 82–88.

Egbreghts, Abigail (2017). ”A Literature Review of Behavior Driven Development using Grounded Theory”.

Fewster, Mark ja Dorothy Graham (1999). *Software test automation: effective use of test execution tools*. Addison-Wesley Reading.

Fowler, Martin (2011). *Eradicating Non-Determinism in Tests*. URL: `https://martinfowler.com/articles/nonDeterminism.html` (viitattu 20. 05. 2018).

Haikala, Ilkka ja Tommi Mikkonen (2011). ”Ohjelmistotuotannon käytännöt”. *Helsinki: Talentum*.

Jasmine (2018). *Jasmine introduction.js (versio 2.0.0)*. URL: `https://jasmine.github.io/2.0/introduction.html` (viitattu 21. 05. 2018).

JBehave (2017). *JBehave story syntax*. URL: `http://jbehave.org/reference/stable/story-syntax.html` (viitattu 09. 09. 2017).

JUnit (2018). *JUnit4 Frequently Asked Questions versio 4.12*. URL: `https://junit.org/junit4/faq.html` (viitattu 20. 05. 2018).

Koskela, Lasse (2008). *Test driven practical TDD and acceptance TDD for Java developers*. Greenwich, CT: Manning. ISBN: 978-1932394-85-6.

Leffingwell, Dean (2010). *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. Addison-Wesley Professional.

Niederwieser, Peter (2018). *Spock dokumentaatio*. URL: `http://spockframework.org/spock/docs/1.1/all_in_one.html` (viitattu 08. 05. 2018).

North, Dan (2006). *Introducing BDD*. URL: `http://dannorth.net/introducing-bdd/` (viitattu 09. 09. 2017).

Rahman, Mazedur ja Jerry Gao (2015). ”A reusable automated acceptance testing architecture for microservices in behavior-driven development”. Teoksessa: *Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium on*. IEEE, s. 321–325.

Sathawornwichit, Chaiwat ja Shigeru Hosono (2012). ”Consistency reflection for automatic update of testing environment”. Teoksessa: *Services Computing Conference (APSCC), 2012 IEEE Asia-Pacific*. IEEE, s. 335–340.

Smart, John Ferguson (2015). *BDD in Action: Behavior-driven development for the whole software lifecycle*. Manning.

Smart, John Ferguson (2018). *The Serenity Reference Manual (versio 1.0.1.1)*. URL: `http://thucydides.info/docs/serenity-staging/#_integration_with_issue_project_tracking_systems` (viitattu 25. 05. 2018).

Soeken, Mathias, Robert Wille ja Rolf Drechsler (2012). ”Assisted behavior driven development using natural language processing”. Teoksessa: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, s. 269–287.

Solis, Carlos ja Xiaofeng Wang (2011). ”A study of the characteristics of behaviour driven development”. Teoksessa: *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*. IEEE, s. 383–387.