Samppa Hynninen

Behaviour-driven development mobiiliohjelmistojen kehityksen tukena

Tietotekniikan (Ohjelmisto- ja
tietoliikennetekniikka)
pro gradu -tutkielma
9. huhtikuuta 2014

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Samppa Hynninen
Yhteystiedot: samppa.hynninen@jyu.fi
Työn nimi: Behaviour-driven development mobiiliohjelmistojen kehityksen tukena
Title in English: Using behaviour-driven development to support software development on mobile platforms
Työ: Tietotekniikan (Ohjelmisto- ja tietoliikennetekniikka) pro gradu -tutkielma
Sivumäärä: 39
Tiivistelmä: Tähän tulee tutkielman tiivistelmä.
Abstract: Here comes the abstract of the thesis.
Avainsanat: bdd, behariour-driven development, mobiilialustat
Keywords: bdd, behariour-driven development, mobile platforms

Sisältö

1 Johdanto **1**

2 Tutkimuksesta **3**
2.1 Termejä 3
2.2 Aikaisemmat tutkimukset 4
2.3 Tutkimusmenetelmistä 5

3 Ketterä ohjelmistokehitys **6**
3.1 Taustaa ja ketterien menetelmien kehitys 6
3.1.1 Vesiputousmalli 6
3.1.2 Spiraalimalli 8
3.1.3 Scrum 11
3.2 Laatu 15
3.2.1 Laatu vaiheellisissa ohjelmistotuotantoprosesseissa 15
3.2.2 Laatu iteratiivisissa ohjelmistotuotantoprosesseissa 16
3.3 TDD 17
3.4 ATDD 20

4 Behaviour-driven development **22**
4.1 BDD:n lähtökohdat ohjelmistokehitystä tukevana menetelmänä . . . 22
4.2 Teknologioista 23
4.2.1 Yleistä teknologioista 23
4.2.2 Parsereista 26

5 BDD:n tarjoamat liiketoiminnalliset edut **28**
5.1 Asiakkaan käyttäjätarinoista hyväksymistesteiksi 28
5.2 BDD:n hyödyntäminen mobiilikehityksessä offshore-tiimeillä 28

6 BDD mobiilialustoilla **29**
6.1 Natiivisovellukset 29
6.2 HTML5-sovellukset 29

7 Crossplatform-testaaminen eri mobiilialustoilla **30**
7.1 iOS BDD-frameworkit 30
7.2 Android BDD-frameworkit 30
7.3 Windows phone BDD-frameworkit 30
7.4 Mahdollisuudet testata kaikki alustat yhdellä testisetillä 30

8 Pohdintaa **31**

9 Yhteenveto **32**

Lähteet **33**

1 Johdanto

Lähivuosina erilaisten mobiilipäätteiden osuus kuluttajien käyttämistä laitteista on kasvanut räjähdysmäisesti. Perinteisten pöytätietokoneiden, kannettavien tietokoneiden ja matkapuhelinten väliin on syntynyt täysin uusia laiteryhmiä, kuten esimerkiksi tablet-tietokoneet. Ohjelmistoteollisuus on joutunut muuntautumaan uusiin vaatimuksiin, jotka uudenlaiset sovellusalustat ovat tuoneet mukanaan. Palvelujen tulee nykyään monesti olla käytettävissä useilla eri alustoilla ja erilaisia alustoja voivat koskea erilaiset vaatimukset. Ohjelmistokehitysmenetelmiä on jouduttu kehittämään jatkuvasti muuttuvan ympäristön ja uusien tarpeiden myötä. Behaviour-driven development, eli käyttäytymislähtöinen kehitys on yksi viime vuosina kehittyneistä ohjelmistokehitystä tukevista tekniikoista, jossa kehityksen lähtökohtana ovat määrittelyt siitä, kuinka ohjelmiston tulisi toimia ja käyttäytyä.

Ohjelmistotestauksen ollessa kriittinen osa ohjelmistotuotantoprosessia ja erityisesti laadunvarmistusta, sen merkitys korostuu entisestään, kun ohjelmistoja toimitetaan useille eri alustoille. Tässä tutkielmassa käsitellään behaviour-driven developmentia ja sen taustoja niin menetelmällisestä kuin puhtaasti teknologisesta näkökulmasta. Aluksi selvitetään mitä BDD:llä tarkoitetaan ja mistä se on kehittynyt. Taustojen ja historian ohella katselmoidaan ohjelmistokehityksen haasteita ja ongelmakohtia, joita BDD:n avulla pyritään välttämään. Teknologioiden osalta käsitellään erityisesti BDD-testaamista mobiiliohjelmistojen kehityksessä ja sen poikkeavuuksia sekä erityispiirteitä verrattuna BDD-menetelmiin muissa ympäristöissä. Tässä yhteydessä tarkastellaan erikseen eri mobiilialustojen natiivisovelluksia sekä uusiin HTML5-teknologioihin pohjautuvia alustariippumattomia sovelluksia, sillä niiden testaaminen eroaa merkittävästi toisistaan.

Tämän pro gradu -tutkielman tavoitteena on kartoittaa behaviour-driven developmentin tarjoamia mahdollisuuksia helpottaa mobiiliohjelmistojen kehitystä ja erityisesti testausta. Tarkoituksena on selvittää millaisia BDD-testikehyksiä merkittävimmille mobiilialustoille on tällä hetkellä olemassa ja miten eri mobiilialustojen testikehykset poikkeavat toisistaan. Tämä lisäksi tarkoituksena on selvittää, millaisia liiketoiminnallisia etuja BDD:llä voidaan mobiilisovelluskehityksessä saavuttaa. Tutkielma toteutetaan käyttäen konstruktiivista tutkimusotetta ja siinä pyritään lopputuloksena toteuttamaan ympäristö, jossa samaa sovellusta voitaisiin testataan eri mobiilialustoilla yhdellä testikokoelmalla. Testiympäristön toteutuksen yhteydessä

tutkitaan millaisia puutteita nykyratkaisuissa on ja miten niitä voitaisiin kehittää, jotta yhden testikokoelman mallia voitaisiin käyttää.

2 Tutkimuksesta

2.1 Termejä

- Apache Cordova: Avoimen lähdekoodin ohjelmisto, johon PhoneGap perustuu.
- ATDD: Acceptance test-driven development. Hyväksymistestilähtöinen ohjelmistokehitys.
- BDD: Behaviour-driven development. Käyttäytymislähtöinen kehitys. Ohjelmistokehitysprosessi, jossa vaatimukset pohjautuvat ohjelmiston käyttäytymiseen.
- CI: Continuous Integration, jatkuva integraatio.
- Cucumber: Rubylla kirjoitettu BDD-testikehys.
- HTML5: HTML-merkintäkielen uusin versio ja samalla käsite nykyaikaisille web-teknologioille, kuten JavaScriptille ja CSS3:lle.
- Jasmine: BDD-testikehys JavaScript-ohjelmointikielelle.
- JBehave: Yksi ensimmäisistä BDD-testikehyksistä Java-ohjelmointikielelle.
- Natiivisovellus: Jonkin sovellusalustan omalla kehitysympäristöllä ja ohjelmointikielellä toteutettu sovellus.
- PhoneGap: Adoben omistama mobiiliohjelmistokehys, jolla voidaan toteuttaa monialustaisia sovelluksia moderneilla web-teknologioilla.
- RSpec: BDD-testikehys Ruby-ohjelmointikielelle.
- Selenium: Työkalu verkkoselaimen automatisointiin.
- TDD: Test-driven development. Testivetoinen kehitys, ohjelmistokehityksen prosessi, jossa testit kirjoitetaan ennen ohjelmakoodia

2.2 Aikaisemmat tutkimukset

Aikaisempaa tieteellistä tutkimusta Behaviour-driven developmentista on tehty verrattain vähän, sillä aihepiiri on niin tuore. Terminä BDD on tullut käyttöön ensimmäisiä kertoja vasta 2000-luvun puolivälissä erirtyisesti Dan Northin blogitekstien myötä [19]. Täten voidaan laskea BDD:n olevan käsitteenäkin vasta noin kymmenen vuoden ikäinen.

Hakiessani lähteitä ja aikaisempia tutkimuksia BDD:n aihepiireistä käytin pääasiallisesti suurimpia tietokantoja etsimiseen. Näitä olivat ACM Digital Library, IEEE Xplore, Google Scholar. Aloitin haut tekemällä muutamia avainsanahakuja [1] eri tietokantoihin. Termillä "bdd" tehdyt haut eivät tuottaneet juuri mistään kannasta toivottuja tuloksia, sillä lyhennettä ilmeisesti käytetään monilla muillakin tieteenaloilla tarkoittamaan eri asiaa. Avainsanahaussa hakusanat "behaviour driven development" tuottivat jo selvästi paremman tuloksen. Google Scholar palautti kyselyllä noin 250 tulosta, joista suuri osa oli relevantteja. IEEE Xploresta ei samoilla hakutermeillä ja sen variaatioilla palautunut kuin kaksi eri julkaisua. ACM:n hausta termeillä ja niiden muutamalla variaatiolla (esim. "behavior driven development") palautui noin kaksikymmentä tulosta.

Hauilla löytyneet tutkimukset pystyi karkeasti jakamaan muutamaan eri luokkaan. Ensimmäiseen ryhmään kuuluvissa tutkimuksissa keskityttiin lähinnä määrittelemään Behaviour-driven developmentia ja ylipäänsä siihen liittyvää termistöä. Näissä tutkimuksissa ja lähteissä aihepiiri oli ikäänkuin uusi ja vielä vakiintumaton. Toisen pääteeman löytyneissä tutkimuksissa muodostavat lähteet, joissa BDD:n tehokkuutta ja sopivuutta on tutkittu eri tilanteissa. Tähän teemaan lukeutuvat myös tutkimukset, joissa BDD:n menetelmiä sovelletaan jollekin tietylle toimialalle.

Sopivimmille artikkeleille tein lisähakuja käyttäen "Backward search"-menetelmää [1], jossa kävin läpi löytyneiden tutkimusten lähteitä. Tällä tavoin löytyi muutamia lähteitä, jotka esiintyivät hyvin monissa BDD:hen liittyvissä tutkimuksissa. Näitä olivat esimerkiksi Dan Northin tekstit [19], Gojko Adzicin kirjat [24],[20] sekä esimerkiksi Cucumber-kirja [21]. Samanlaisia tuloksia löytyi, kun kävin lähteille läpi vielä "Backward search"-menetelmää, mutta tällä kertaa keskittyen kirjoittajiin, eli tutkin kirjoittajien muita teoksia.

Näiden hakujen myötä päädyin käyttämään lähteinä erityisesti usein esilletulleita teoksia, kuten Adzicin kirjat, Cucumber-kirja sekä Dan Northin kirjoitukset. Tämän lisäksi mukaan valikoitui kirjoittaessa jatkuvasti uusia lähteitä, joita löytyi ennen kaikkea tekemällä "Backward search"-hakuja aineistolle.

2.3 Tutkimusmenetelmistä

Tutkimus on toteutettu käyttäen konstruktiivista tutkimusotetta. Konstruktiivisessa tutkimusotteessa tavoitteena on rakentaa jonkinlainen artefakti, joka ratkaisee jonkin tietyn alan kysymyksen ja tätä kautta lisää tietoa ongelmasta ja sen ratkaisusta alalla [2]. Tutkielmassani yritän löytää ratkaisua kysymykseen, voidaanko tämänhetkisten modernien mobiilialustojen ohjelmistoja testata tehokkaasti käyttäen Behaviour-driven developmentin toimintatapoja.

Tutkimuskysymykseen vastatakseni tutkimuksessa kartoitetaan ensiksi millaisia BDD-työkaluja mobiilialustoille on nykyään saatavilla. Tutkimuksessani käytettävät alustat olen rajannut kattamaan kolme suosituinta alustaa Gartnerin tutkimuksen pohjalta (Q4/2013) [3]. Kolme mukaan otettua alustaa olivat Googlen Android, Applen iOS ja Microsoftin Windows Phone. Tämän jälkeen tehtävänä oli kartoittaa valituille alustoille tällä hetkellä olemassa olevia BDD:n mahdollistavia ratkaisuja. Löydettyjä ratkaisuja käydään läpi luvussa 7.

Olemassa olevien ratkaisuiden kartoittamisen myötä tutkielmassa tutkittiin onko mahdollisesti jo olemassa BDD-testikehyksiä, joilla voidaan testata kerralla useita eri alustoja. Samalla tutkittiin onko eri mobiilialustoille löydettävissä sellaisia BDD-testikehyksiä, joiden käyttämä luonnollinen kieli toimisi niin, että samoilla luonnollisella kielellä kirjoitetuilla testeillä voisi ainoastaan eri testikehysten kieltä jäsentävää ohjelmakoodia muokkaamalla saada samat testit toimimaan eri testikehyksillä.

Tämän jälkeen tutkimuksessa käydään vielä läpi erikseen ratkaisu ongelmaan, kun kyseessä ovat natiivisovellusten sijaan HTML5-sovellukset, sillä näiden kohdalla testaus voidaan suorittaa eri tavalla. Tutkimuksessa kartoitetaan HTML5-sovellusten testaamista sekä käydään läpi ratkaisuja, joilla HTML5-sovelluksia voidaan paketoida helposti eri alustoja varten ja näin ollen saavuttaa samat edut BDD:llä kuin natiivisovellusten kohdalla. Tästä muodostetaan oma ratkaisunsa täydentämään natiivisovellusten testaukseen löytyneitä ratkaisuja.

3 Ketterä ohjelmistokehitys

3.1 Taustaa ja ketterien menetelmien kehitys

Niin kauan kuin tietokoneohjelmistoja on kehitetty, on niiden yhteydessä kehitetty myös erilaisia malleja, joiden tarkoituksena on helpottaa ja sujuvoittaa ohjelmistokehitystä. Menetelmät voidaan karkeasti jakaa inkrementaalisiin ja iteratiivisiin menetelmiin [4]. Inkrementaalisissa malleissa ideana on se, että ohjelmisto kasvaa jatkuvasti kehittyen kohti lopullista valmista tuotetta, kun taas iteratiivisissa malleissa ohjelmistoa kehitetään pienissä osissa lyhyemillä aikaväleillä ja tätä työtä toistetaan useaan otteeseen. Seuraavassa esitelläänkin muutamia tunnettuja ohjelmistotuotannon malleja ja niiden kehitystä kohti ketteriä menetelmiä.

3.1.1 Vesiputousmalli

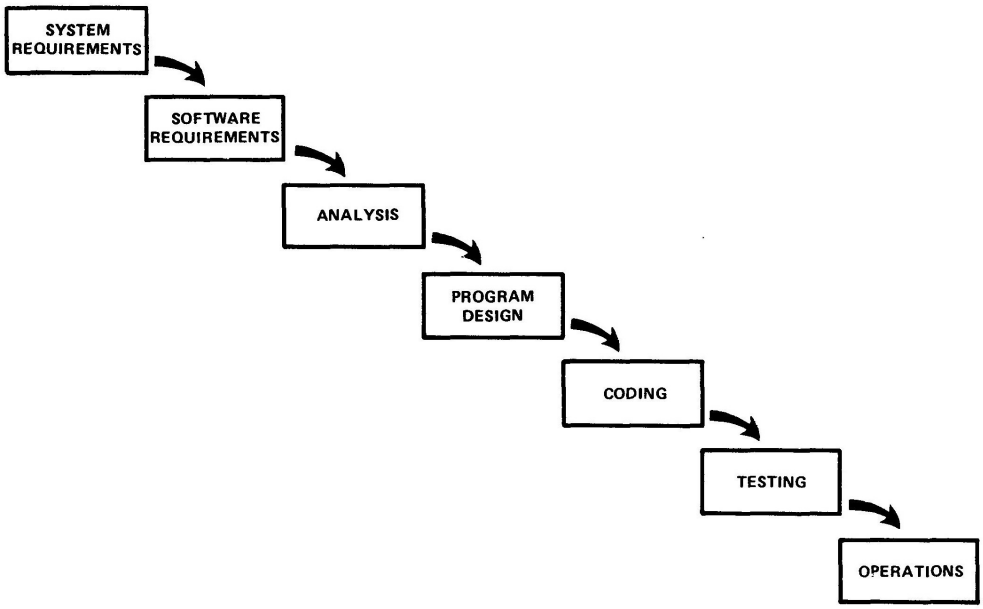
Yksi varhaisista ohjelmistokehityksen malleista on inkrementaalinen vesiputousmalli, jonka perustana toimii hyvin pitkälti Winston Roycen kuvaama malli artikkelissaan Managing the development of large software systems [6] vuodelta 1970. Huolimatta siitä, että vesiputousmalli on erittäin vanha ja sen käyttäminen sisältää tunnettuja riskejä, on malli edelleen hyvin suosittu. Roycen malli perustuu ajatukselle, että kaikissa ohjelmistokehitystehtävissä on aina ainakin kaksi vaihetta, analyysivaihe ja itse ohjelmiston koodausvaihe. Vesiputousmallissa luodaan tämän ajatuksen ympärille laajempi malli, jossa edellämainittuja vaiheita on täydennetty uusilla vaiheilla, jotka ovat kuitenkin täysin erillisiä analyysi- ja koodausvaiheista. Näitä ovat Roycen mallissa [6] kaksi vaatimusmäärittelyvaihetta, ohjelmiston suunnittelu ja koodauksen jälkeinen testausvaihe. Näillä lisäyksillä malli on soveltuvampi suurten ohjelmistoprojektien läpivientiin. Roycen esittämä malli ei kuitenkaan ole täysin se vesiputousmalli, joka on vielä nykyäänkin käytössä, sillä Royce itsekin ehdottaa parhaimpana versiona mallistaan sellaista, jossa eri vaiheiden välillä voidaan liikkua kumpaankin suuntaan, eli myös takaisinpäin edellseen vaiheeseen.

Royce kuitenkin tiedostaa vesiputousmallin sisältämiä riskejä jo omassa artikkelissaan. Yksi näistä on esimerkiksi vaara, että vasta testivaiheessa huomataan puutteita, jotka johtavat perinpohjaisiin muutoksiin ohjelmiston rakenteessa tai toiminnassa, jolloin käytännössä koko kehitysprosessi joudutaan aloittamaan alusta.

Näitä puutteita voi suurellakin todennäköisyydellä ilmetä, sillä testausvaihe on ensimmäinen kerta, kun järjestelmä toimii kokonaisuutena integroituna muihin järjestelmiin [6]. Toinen merkittävä tekijä, joka monesti jätetään huomioimatta vesiputousmallia käytettäessä, on se, että Royce itsekin suosittelee käymään prosessin läpi kahteen kertaan. Tämä pätee etenkin tilanteisiin, jossa toimitetaan asiakkaalle jokin täysin uusi tuote. Tällöin ensimmäisellä kerralla luodaan pilotti tuotteesta lyhyessä ajassa. Tämän pilotin tarkoituksena on tuoda esiin kehitettävän tuotteen erityiset haasteet, löytää suunnittelun mahdolliset puutteet sekä luoda eri vaihtoehtoja näiden ratkaisemiseen [6]. Tällöin lyhyessä ajassa kehitetty pilottituote toimii ikään kuin koko projektin simulaationa, jolloin vaikeisiin tilanteisiin osataan varautua paremmin.

Royce ei itsekään pitänyt malliaan parhaana mahdollisena suurten ohjelmistojen kehitykseen, vaan Larmanin ja Basilin artikkelissa [4] kerrotaankin, kuinka hän itsekin toteaa vesiputousmallin olevan kaikista yksinkertaisin malli, joka ei kuitenkaan todennäköisesti tule toimimaan kuin kaikkein yksinkertaisimpien ja suoraviivaisimpien projektien yhteydessä. Ensimmäiset maininnat ja kehotukset iteratiiviseen ohjelmistokehitykseen ovatkin jo ajalta ennen Roycen mallin julkaisua. Jo vuonna 1969 IBM:n tutkimuskeskuksessa M.M Lehman kuvasi iteratiivista mallia, jossa erotettiin suunnittelu, arviointi ja dokumentointi [4]. Tässä mallissa suunnittelu tuotti jo toimivan mallin, jota sitten iteratiivisesti kehitettiin eteenpäin.

Kuitenkin vielä nykypäivänäkin suuriakin ohjelmistoprojekteja toteutetaan käyttäen vesiputousmallia, sillä sen koetaan sisältävän joitain etuja tietyissä ohjelmistoprojektin vaiheissa verrattuna ketteriin menetelmiin. Monesti nämä ratkaisevat edut liittyvät erityisesti projektien myyntiin tarjousvaiheessa tehtyjen tarjousten muotoiluun. Vesiputousmallilla myytävissä ratkaisuissa on helpompaa myydä asiakkaalle kiinteällä hinnalla yksi tarkasti määritelty kokonaisuus, jonka myötä asiakas kokee tietävänsä tarkalleen, millaisen tuotteen hän on saamassa ja mihin hintaan. Ketteriä projekteja myytäessä asiakas käytännössä ostaa vain työtä, eikä kiinteästi määritettyä lopputulosta [5]. Tällöin asiakkaalle jää jatkuvasti mahdollisuus vaikuttaa siihen, että tehty työ on juuri sitä, mistä hänelle on eniten hyötyä, eikä ohjelmistoon toteuteta esimerkiksi täysin turhia tai vanhentuneita vaatimuksia. Kuitenkin tästä huolimatta monesti ainoastaan lopullinen kiinteä hinta ja myyntivaiheessa määritelty lopputulos ratkaisevat ostopäätöstä tehdessä, jolloin edelleen nykyään saatetaan päätyä toteuttamaan projekteja vesiputousmallia käyttäen.



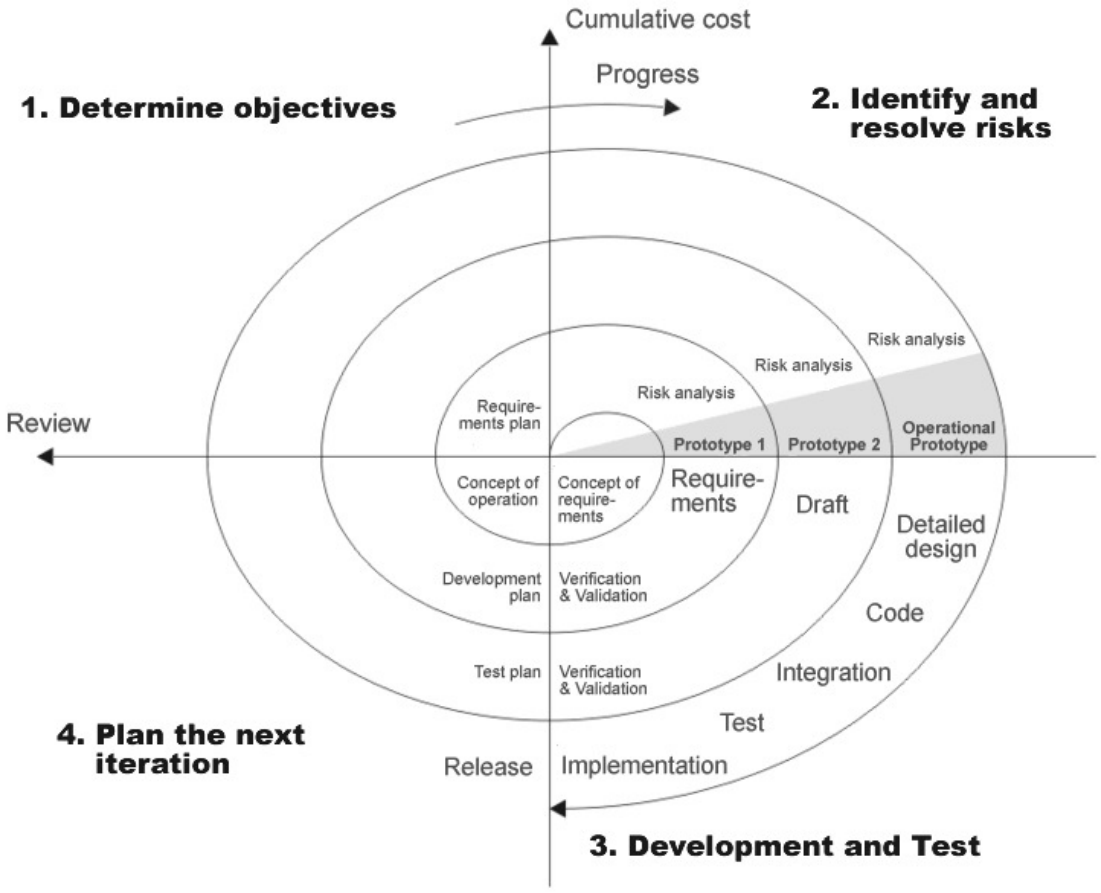
Kuva 3.1: Vesiputousmalli [6]

3.1.2 Spiraalimalli

Spiraalimalli on Barry Boehmin kehittämä [7] malli vuodelta 1986, joka on ensimmäisiä tunnettuja iteratiivisia malleja ohjelmistokehityksessä. Spiraalimallin tärkeimpänä ajatuksena on se, että siinä otetaan riskit entistä paremmin huomioon verrattuna esimerkiksi perinteiseen vesiputousmalliin. Spiraalimallissa jokainen iteraatio sisältää neljä merkittävää osaa, jotka ovat [7]:

1. Iteraation tavoitteiden, vaihtoehtojen ja rajoitteiden määrittely
2. Eri vaihtoehtojen arviointi sekä riskien tunnistaminen ja arviointi
3. Kehitystyö ja seuraavan tason tuotteen määrittely
4. Seuraavan iteraation suunnittelu

Mallissa jokaisessa iteraatiossa toteutetaan riskianalyysin jälkeen prototyyppimalli, josta sitten lähdetään kehittämään valmista tuotetta, aivan kuten Roycen [6] ehdottamassa pilottituotemallissa. Lisäksi spiraalimallissa merkittävää on se, että jokainen iteraatiokierros sillä hetkellä valmiina olevan tuotteen arviointiin, johon osallistuvat kaikki tärkeimmät henkilöt, jotka tulevat käyttämään tuotetta [7]. Boehmin malli toimiikin pohjana hyvin monille nykyisille ketterille menetelmille, joissa hyödynnetään samankaltaisia iteraatioita. Esimerkiksi seuraavana käsiteltävässä



Kuva 3.2: Spiraalimalli [6]

Scrumissa sprintit vastaavat hyvin pitkälle spiraalimallin iteraatioita, eli Scrum käytännössä sisältää spiraalimallin prosessit, joiden päälle on vielä vain lisätty muuta.

Spiraalimallissa käytetään erittäin paljon riskiä määrittämään tehtäviä asioita. Boehmin mukaan [7] esimerkiksi eri työvaiheisiin käytettävä aika määräytyy sen perusteella, miten ne vaikuttavat olemassaolevaan riskiin. Jos esimerkiksi testaukseen käytetty aika pienentää projektin kokonaisriskiä, siihen kannattaa käyttää aikaa mieluummin kuin esimerkiksi uuden sovelluskehyksen käyttöön vanhan tutun sovelluskehyksen sijaan, sillä tällainen valinta vain lisäisi projektin riskiä ja epäonnistumisen mahdollisuutta. Tämän lisäksi Boehmin mukaan projektiryhmä käyttää riskin arviointia siihen, kuinka yksityiskohtaisella tasolla asioita kannattaa projektissa tehdä [7]. Esimerkiksi projektin dokumentointi voisi olla yksi osa-alue, johon tätä voidaan soveltaa. Projektiryhmän tulee tällöin tehdä päätös, milloin projektin dokumentaatio on riittävällä tasolla siten, että sen tarkentamisesta ja edelleen kehittämisestä ei saada enää mitään hyötyä, vaan se voi johtaa vain kasvavaan riskiin, kuten esimerkiksi julkaisun myöhästymiseen.

Uudemmassa julkaisussaan vuodelta 2000 Boehm [8] vielä erikseen listaa merkkejä, jotka erottavat projektin spiraalimallista ja ovat ennenkaikkea haitallisia projektin onnistumisen edellytyksille. Käytännössä nämä merkit ovat sellaisia, joista voidaan huomata käytettävän oikeasti vesiputousmallia, jota vain nimitetään spiraalimalliksi. Tällaisia merkkejä ovat Boehmin mukaan seuraavat kuusi olettamusta [8]:

1. Vaatimukset tunnetaan ennen ohjelmiston toteutusta
2. Vaatimuksissa ei ole epäselviä olettamuksia
3. Vaatimusten luonne ei muutu kehitystyön ja projektin etenemisen myötä
4. Vaatimukset täyttävät kaikkien sidosryhmien edustajien oletukset
5. Oikea arkkitehtuuri vaatimusten täyttämiseen on hyvin ymmärretty ja omaksuttu
6. Tarpeeksi kalenteriaikaa on varattu, jotta voidaan edetä peräkkäisessä järjestyksessä vaihe kerrallaan

Boehm määrittelee myös erilaisia invariantteja, jotka tulee olla olemassa spiraalimallin toimimisen edellytyksenä [8]. Nämä invariantit käytännössä määrittelevät vain tarkemmin spiraalimallin eri osa-alueet ja niiden sisällöt. Näihin Boehmin invariantteihin kuuluu esimerkiksi edellämainitut käytettävän panoksen määrittely

riskin perusteella sekä työvaiheen toteutuksen yksityiskohtaisuuden tason arviointi riskin perusteella. Näiden lisäksi Boehmin määrittelemiin invariantteihin lasketaan myös keskittyminen järjestelmään ja sen elinkaareen sen sijaan, että ainoastaan mietittäisiin ohjelmiston kehittämistä teknisestä näkökulmasta [8].

3.1.3 Scrum

Scrum on moderni iteratiivinen ohjelmistokehityksen viitekehys, jota ovat kehittäneet pääasiallisesti yhdysvaltalaiset Ken Schwaber ja Jeff Sutherland aina 1990-luvun alkupuolelta lähtien. Kuten termi viitekehys antaa ymmärtää, ei Scrum tarjoa mitään yksityiskohtaista prosessia ohjelmistoprojektin toteuttamiseen, vaan ennemminkin juuri kehyksen, jonka puitteissa voidaan hallita monimutkaisten tuotteiden kehitystä [9]. Scrum tarjoaa käytännön työkalut ohjelmistojen kehittämiseen iteratiivisesti siten, että riski on jatkuvasti mahdollisimman pieni ja työn tulokset optimoituja, eli käytännössä tehdään aina sitä, mikä on juuri sillä hetkellä asiakkaalle kaikkein hyödyllisintä. Tämä saavutetaan, kun työskennellään nojautuen Scrumin kolmeen lähtökohtaan, jotka ovat läpinäkyvyys, tarkastelu ja sopeutuminen [9].

Läpinäkyvyys tulee Scrumissa ilmi erityisesti esimerkiksi siinä, että kaikki eri sidosryhmät ovat jatkuvasti tietoisia toteutettavan projektin tilanteesta. Käytännön työssä tämä on toteutettu monin eri tavoin [9]:

- Työtä suoritetaan kiinteän pituisissa jaksoissa, joita kutsutaan sprinteiksi. Usein sprinttien kesto on yhdestä kolmeen viikkoa. Näihin sprintteihin projektitiimi ottaa mukaan työtä niin paljon kuin he arvioivat kykenevänsä toteuttamaan. Tässä kohtaa ainoastaan projektitiimi voi määritellä mitä he tekevät, sillä heillä on ainoina henkilöinä tieto siitä, mitä he tiiminä pystyvät toteuttamaan. Tämä sprintiin otettu työ muodostaa sprintin backlogin, johon voidaan sprintin suunnittelun jälkeen ainoastaan projektitiimin luvalla ottaa lisää työtä. Tällöin asiakas ja projektin muut sidosryhmät tietävät aina mitä seuraavan sprintin aikana on tarkoituksena saada toteutetuksi [9].

- Koko projektin seurantaan ja läpinäkyvyyteen Scrumissa on olemassa tuotteen backlog, joka sisältää kaikki tulevaisuudessa toteutettavaksi toivotut ominaisuudet ja tehtävät asiakkaan puolelta. Tätä tuotebacklogia asiakas priorisoi yhdessä Scrum-tiimiin kuuluvan tuoteomistajan kanssa. Tämä tarkoittaa sitä, että backlogilla nostetaan prioriteetiltaan tärkeimmät tehtävät ensimmäisiksi. Tämä on tärkeää, sillä projektitiimi ottaa sitten tuotebacklogilta jokaiseen sprintiin niin paljon työtä kuin he kokevat pystyvänsä tekemään. Työtä ote-

taan tuotebacklogilta mukaan sprintiin siinä järjestyksessä, mihin asiakas on tehtävät yhdessä tuoteomistajan kanssa priorisoinut.

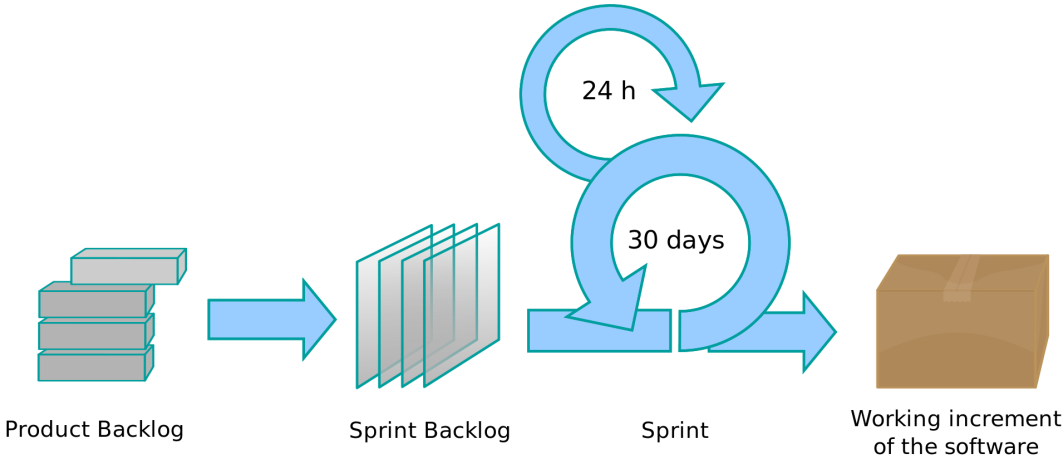
- Projektissa toteutettavan tuotteen tila on koko ajan selvillä kaikille sidosryhmille, sillä Scrumissa on tavoitteena, että jokaisen valmiin sprintin jälkeen projektiryhmä voisi toimittaa asiakkaalle käytännössä toimivan inkrementin toteutettavaan ohjelmistoon, jolloin jokaisen sprintin työn tulos olisi heti nähtävillä. Käytännössä toimivan inkrementin toimitus ei kuitenkaan aina ole esimerkiksi projektin luonteen vuoksi mahdollista, jos toteutettavana tuotteena on jokin laaja tietojärjestelmä, jossa käytetään useita sprinttejä esimerkiksi pelkästään back endin tai tietokantojen toteuttamiseen. Tällöin inkrementti voi olla kuitenkin toimiva, vaikkei asiakas siitä heti suoraa hyötyä saisikaan.

Tehtyä työtä, työtapoja ja projektia kokonaisuudessa arvioidaan ja tarkastellaan Scrumin puitteissa jatkuvasti. Erityistä tässä on se, että näitä asioita ei ole arvioimassa vain yksi taho, vaan työn ja tulosten tarkastelua tehdään jatkuvasti kaikkien toimesta, jotka liittyvät projektiin millään tapaa.

Päivittäin työtä arvioidaan ns. daily-palavereissa, joissa jokainen projektitiimin jäsen pääsee kertomaan mitä on tehnyt, mitä aikoo tehdä ja samalla hän voi kertoa, jos työssä on ilmennyt jotain ongelmia. Tällaisen lyhyen kiinteästi 15-minuuttiseksi ajoitetun palaverin myötä koko tiimi on joka päivä selvillä siitä, mitä muut tekevät ja mahdollisesti pystyvät näin ollen esimerkiksi auttamaan muita ja tehostamaan omaa työtään sillä, etteivät he tee päällekkäisiä tai toisistaan riippuvia tehtäviä.

Jokaisen sprintin päättyessä järjestetään niin kutsuttu sprintin arviointi, jossa projektitiimi yhdessä asiakkaan kanssa käy product ownerin johdolla läpi sprintin aikana toteutettuja ominaisuuksia ja tehtäviä. Samassa tilaisuudessa käydään läpi ne sprintiin mukaan otetut työt, joita projektitiimi ei ole ehtinyt saada valmiiksi. Sprintin arviointi onkin hyvä tilaisuus sekä projektitiimille, että asiakkaalle, sillä tällöin kummatkin pääsevät yhdessä keskustelemaan siitä, mitä tiimi on tehnyt. Tällöin asiakkaalla on mahdollisuus esittää kysymyksiä ja toteuttava tiimi pääsee myös kertomaan mahdollisista ongelmista ja niiden ratkaisuista, eli tilaisuudessa pystytään viimeistään oikomaan mahdollisia väärinkäsityksiä tai näkemyseroja.

Daily-palaverien ja sprintin arvioinnin lisäksi projektitiimi käy jokaisen sprintin jälkeen läpi retrospektiivin, jossa jokaisella tiimin jäsenellä on mahdollisuus arvioida, mikä tiimin työssä meni hyvin ja millä alueilla on vielä kehitettävää. Nämä onnistumiset liittyvät yleensä ennemminkin tiimin sisäisiin asioihin kuin mihinkään muuhun, eli ennenkaikkea asioihin, joihin tiimi itse voi vaikuttaa. Tällaisia asioita ovat esimerkiksi:



Kuva 3.3: Scrum [10]

- Kuinka hyvin Scrumin periaatteita on noudatettu?
- Onko tiimin sisäisessä dynamiikassa jotain ongelmia?
- Voisiko työtapoja parantaa jotenkin?
- Onko jotain projektiin liittymättömiä ongelmia, jotka haittaavat työtä?

Kolmas Scrumin perusperiaate, eli sopeutuminen, tulee myös esille monissa eri yhteyksissä ja toimintatavoissa. Edellämainituissa daily-palavereissa yksi tärkeä asia on se, että kaikki projektitiimin jäsenet pääsevät kertomaan mahdollisista ongelmistaan, joita ovat työssään kohdanneet. Tällöin tiimillä on mahdollisuus puuttua näihin ongelmiin heti ja ratkaista ne yhdessä sen sijaan, että tiimin jäsen jäisi yksin ongelmiensa kanssa ja töiden eteneminen mahdollisesti hidastuisi, jolloin myöskään sprintin tavoite ei välttämättä täyttyisi.

Jokaisen sprintin päätteeksi läpikäytävä retrospektiivi on työn ja työtapojen tarkastelun ohella erittäin hyvä tilaisuus tiimille sopeutumiseen, sillä tilaisuudessa ei ole tarkoitus tuoda esiin vain huonoja toimintatapoja tai ongelmia, vaan ennenkaikkea näihin esille tuleviin kehityskohteisiin yritetään koko tiimin voimin löytää ratkaisuja heti [11]. Tällöin ongelmat eivät jää kenenellekään yksin hoidettaviksi, vaan niihin voidaan heti puuttua useamman henkilön voimin. Myös toisessa sprintin päätteeksi läpikäytävässä tapahtumassa, eli sprintin arvioinnissa, on eri sidosryhmillä hyvät mahdollisuudet sopeutumiseen. Tässä tilaisuudessa tilaisuudessa asiakas voi antaa palautetta kehitystiimin suuntaan, kuten myös toisinpäin. Tällöin esi-

merkiksi kehitystiimi voi ottaa oppia, jos vaikkapa joitain asiakkaan pyyntöjä on ymmärretty väärin ja vastaavasti asiakas voi saada hyvää kokemusta siitä, millä tavalla asiat kannattaa kehitystiimille esittää.

Ylipäänsä Scrumin sprinttiajattelu tarjoaa jatkuvaan sopeutumiseen erittäin hyvät mahdollisuudet. Kun käytännön työssä työn alle otettavat tehtävät lukitaan vain yhdeksi sprintiksi kerrallaan, tiimin on helppo sopeutua muuttuviin vaatimuksiin ja asiakkaan toiveisiin. Esimerkiksi asiakkaalle tärkeä uusi ominaisuus voidaan ottaa heti seuraavassa alkavassa sprintissä työn alle, jos sen prioriteetti on niin tärkeä, että se menee muiden tuotteen backlogin tehtävien edelle. Muutenkin sprintit tarjoavat tiimille mahdollisuuden muuttua tarpeen mukaan nopeasti. Tiimin kokoa voidaan helposti tarpeen mukaan muokata, uusia toimintatapoja käydä retrospektiiveissä läpi sekä suuriakin suunnanmuutoksia projektissa voidaan tehdä helposti verrattuna esimerkiksi vanhaan vesiputousmalliin.

Edellämainittujen tilaisuuksien ohella Scrum-tiimissä on määritelty henkilö, joka mahdollistaa tiimin sopeutumista. Tälle henkilöllä on Scrum-tiimissä oma erityinen roolinsa, aivan kuten tuoteomistajalla ja kehitystiimin jäsenellä. Tätä kutsutaan scrummasteriksi, jonka tehtävänä on toimia palvelevana johtajana. Käytännön tasolla tämä tarkoittaa sitä, yksi kehitystiimin jäsenistä toimii mahdollisten omien tehtäviensä ohella scrummasterin roolissa tai mahdollisesti hänellä ei ole mitään muita tehtäviä kuin scrummasterin tehtävät. Scrummasterin tehtävänä on toimia kehitystiimille mahdollistajana, eli hän ratkaisee kehitystiimin puolesta sellaisia ongelmia, joita esimerkiksi daily-palavereissa tulee esille, jotka haittaavat heidän työntekoaan. Scrummasterin tehtävänä on myös valvoa, että Scrumin käytänteitä toteutetaan ja niistä pidetään kiinni. Tämä tarkoittaa sitä, että scrummaster toimii fasilitaattorina, eli pitää huolen siitä, että esimerkiksi daily-palaverit pidetään, sprintien suunnittelut pidetään ja kaikista määrätyistä aikamääreistä pidetään kiinni. Scrummasterin tehtävänä on huolehtia näille tapahtumille paikat, joissa ne voidaan järjestää. Hän myös toimii yhteydenpitäjänä kehitystiimin ja muiden sidosryhmien välillä siten, että itse kehitystiimin jäsenten työ ei häiriinny turhaan. Täten scrummaster voi esimerkiksi selittää Scrumin periaatteita muille sidosryhmille, jotta päästäisiin mahdollisimman hyvään ymmärrykseen siitä, miksi asioita tulee tehdä Scrumin toimintaperiaatteiden mukaan, jolloin kehitystiimin ei tarvitse huolehtia tämänkaltaisten asioiden selittämisestä muille tahoille.

3.2 Laatu

Tietokoneohjelmistojen laadun merkitys on jatkuvasti yhä suuremmassa roolissa, kun monet yhteiskunnan tärkeistä toiminnoista digitalisoituvat. Tämä tarkoittaa myös sitä, että yhä suuremmassa määrin yhteiskunnan toiminnan kannalta kriittiset palvelut nojaavat ohjelmistojen toimintavarmuuteen. Tämä sama ilmiö on tapahtunut myös teollisuudessa, eikä nykypäivänä oikeastaan ole mitään alaa, jossa tietotekniikka ei tarvittaisi. Laadunvarmistus ja suhtautuminen laadun käsitteeseen poikkeaa hyvin paljon vaiheellisissa ohjelmistotuotantoprosesseissa verrattuna iteratiivisesti toimiviin prosesseihin. Seuraavassa käsitelläänkin näiden erityispiirteitä ja eroavaisuuksia.

3.2.1 Laatu vaiheellisissa ohjelmistotuotantoprosesseissa

Vaiheellisissa ohjelmistotuotantoprosesseissa, kuten esimerkiksi vesiputousmallissa, testaus ja laadunvarmistus on erotettu omaksi erilliseksi vaiheekseen, joka tapahtuu vasta toteutusvaiheen jälkeen [6]. Tällöin tilanne on se, että koko projekti on toteutettu valmiiksi asti kiinteillä muuttumattomilla vaatimuksilla, jotka on lukittu ennen toteutustyön aloittamista. Testausvaiheessa toteutettua ohjelmisto testataan näitä vaatimuksia vasten, eli pidetään huolta, että nämä vaatimukset täyttyvät.

Monesti ongelmallista onkin, että asiakas saa toimivan tuotteen vasta tämän testausvaiheen jälkeen ensimmäistä kertaa itselleen testattavaksi, jolloin kyseessä on mahdollisesti alkuperäisten vaatimusten mukaisesti toimiva tuote, mutta käytännössä asiakkaan näkemys ja tarve on voinut muuttua jo monilta osin sinä aikana, kun ohjelmisto on toteutettu. Näin ollen valmiin tuotteen laatu voikin olla asiakkaan mielestä huonompi kuin projektin toteuttajien näkemys on. Kuitenkin teknisellä tasolla vaiheellisissa prosesseissa voidaan panostaa laatuun aivan samalla tapaa kuin iteratiivisissakin prosesseissa. Implementaatiota tehdessä yksikkötestauksen merkitys laadunvarmistuksessa on aivan yhtä suuri kuin iteratiivisissakin prosesseissa.

Kaikenkaikkiaan vaiheellisissa ohjelmistotuotantoprosesseissa testaukseen ja laadunvarmistukseen on hyvät ja kehittyneet keinot, sillä tämä osa-alue on aina ollut mukana omana vaiheenaan prosesseissa [12]. Ongelmaksi näissä prosesseissa laadun kohdalla muodostuukin lähinnä eri osapuolten käsitys laadusta ja mahdolliset asiakkaan muuttuneet tarpeet.

3.2.2 Laatu iteratiivisissa ohjelmistotuotantoprosesseissa

Iteratiivisista ohjelmistotuotantoprosesseista tässä keskitytään erityisesti Scrumiin ja sen suhtautumiseen laatuun. Iteratiivisissa prosesseissa jokaisen sprintin voidaan ajatella toimivan ikäänkuin samalla tavoin kuin kokonaisen vesiputousmallin projektin kaikki vaiheet, sprintin sisältä ei kuitenkaan välttämättä voida selvästi näitä vaiheita erottaa [12]. Kuitenkin toteutustyössä tärkeitä laadunvarmistuksen välineitä ovat esimerkiksi yksikkötestit, joita käytetään aivan samalla tavoin kuin vaiheellisissa prosesseissa. Erona vaiheellisiin prosesseihin, iteratiivisissa prosesseissa testaus tehdään pohjautuen sprintiin valittuihin tehtäviin, joka tarkoittaa sitä, että sprintiin valituille töille on määritetty hyväksymiskriteerit, joissa määritellään milloin haluttu ominaisuus on valmis. Sprintin aikana valmistuvia tehtäviä testataan sitten asetettuja hyväksymiskriteerejä vasten, jotka määritellään yhdessä asiakkaan kanssa.

Sprinteissä tapahtuvan testauksen ja laadunvarmistuksen ohella Software Quality and Methods -julkaisu [12] listaa muutamia muita iteratiivisissa ohjelmistotuotantoprosesseissa yleisesti käytössä olevia toimintatapoja, joiden voidaan katsoa parantavan ohjelmiston laatua:

- Jatkuvalla integraatiolla tarkoitetaan käytäntöä, jossa ohjelmistoon tehtyjä muutoksia integroidaan jatkuvasti ja muutosten myötä automaattisesti testataan ohjelmiston toiminta ja integraatiot [13]. Tällaisella käytännöllä kehitystiimillä on jatkuvasti toimiva ohjelmisto, ja kehitystiimi pääsee eroon suuresta riskistä, joka syntyy, kun suuri määrä muutoksia ohjelmistoon otetaan käyttöön samaan aikaan. Kun tehdyt muutokset integroidaan heti valmistumisen jälkeen muuhun järjestelmään, voidaan mahdollisiin virheisiin tai suurempiin ongelmiin puuttua heti.

- Jatkuvan integraation palvelimet, kuten esimerkiksi yksi suosituimmista avoimeen lähdekoodiin perustuviasta ratkaisuista, Jenkins [14], toimivat jatkuvan integraation mahdollistajina. Käytännössä nämä palvelimet aina versiohallinnassa tapahtuvien muutosten yhteydessä buildaavat sovelluksen automaattisesti, ajavat sille kirjoitetut testit ja ottavat sen mahdollisesti käyttöön sovelluspalvelimella. Jatkuvan integraation palvelimet antavat siten heti viestin, mikäli ohjelmiston buildaaminen ei onnistu tai testeissä on virheitä, jolloin kehittäjät tietävät virheistä lähes heti commitin jälkeen [13].

- Usein iteratiivisissa prosesseissa, kuten Scrumissa, on kommunikaatiolla ja yhdessä asiakkaan kanssa työskentelyllä tärkeä rooli, jonka voidaan nähdä pa-

rantavan ohjelmiston laatua. Tällä tavoin asiakkaaseen saadaan helposti yhteyttä ja hänen kanssaan voidaan jatkuvasti kommunikoida, mikä edistää esimerkiksi epäselvien vaatimusten selvityksessä ja niiden jatkuvaa parantamista [12]. Myös esimerkiksi Scrum-tiimin tuoteomistaja on usein asiakkaan edustaja, jolloin asiakkaalla on jatkuvasti todella hyvä näkyvyys siihen, mitä ollaan tekemässä, kun tuoteomistaja vastaa tuotebacklogin priorisoinnista. Tällöin asiakas voi varmistaa sen, että kehitystiimi työskentelee jatkuvasti sellaisten töiden parissa, jotka ovat asiakkaalle kaikkein hyödyllisimpiä. Tilanne, jossa tuoteomistaja on nimetty asiakkaan puolesta, on kommunikaatiomahdollisuuksiensa puolesta lähes ihanteellinen, sillä Scrumissa tuoteomistaja on henkilö, jolla tulisi olla jatkuvasti kaikkein paras visio siitä, mitä ollaan tekemässä ja mitä ohjelmistoon seuraavaksi halutaan tehdä [9].

Ketterissä menetelmissä kehittäjillä on enemmän vastuuta ohjelmiston laadusta kuin perinteisissa vaiheellisissa ohjelmistotuotantoprosesseissa. Tämä ei tule varsinaisesti esille suoraan iteratiivisissa ohjelmistotuotantoprosesseissa, mutta tämä johtuu hyvin pitkälti siitä, että näissä prosesseissa pelkän suunnittelun määrä on usein minimoitu ja jonkinlaista toteutusta lähdetään tekemään lähes heti. Tällöin laadunvarmistusta joudutaan tekemään paljon jo kehittäjien toimesta ohjelmistoa implementoidessa [12]. Nopeisiin sykleihin, jatkuvaan integraatioon ja tuotteen toimituksiin on ohjelmistoalalla iteratiivisissa prosesseissa sopeuduttu ohjelmistokehityksen toimintatapoja on muokkaamalla siten, että ohjelmoinnissa käytetään menetelmiä, joiden voidaan katsoa edistävän laadunvarmistusta. Yksi tärkeistä laatua edistävistä ohjelmistokehityksen malleista on TDD, jota käsitellään seuraavassa.

3.3 TDD

TDD, eli Test-driven development, tarkoittaa testivetoista ohjelmistokehitystä. TDD poikkeaa edellä esitetyistä malleista siinä, että siinä missä edellä esitellyt mallit kattavat koko ohjelmistokehitysprosessin, ottaa TDD kantaa ainoastaan edeltävissä malleissa esiintyneisiin käytännön toteutus- ja testausvaiheisiin. Käytännössä siis nykyään ohjelmistoprojekteissa valitaan jokin koko ohjelmistokehitysprosessin kattava malli, kuten Scrum, jossa sitten hyödynnetään TDD:tä ja sen ohjelmointikäytänteitä. Seuraavassa esitelläänkin TDD:n keskeisimmät ideat ja toimintaperiaatteet, joita käytännön ohjelmistokehityksessä usein noudatetaan.

Yksinkertaistettuna testivetoisessa ohjelmistokehityksessa ideana on se, että testit kirjoitetaan ennen kuin kirjoitetaan riviäkään koodia. Käytännössä tämä tarkoit-

taa sitä, että ennen ohjelmointia testeillä määritellään se, mitä seuraavaksi kirjoitettavan metodin olisi tarkoitus tehdä. Työvaiheet tässä prosessissa ovat seuraavanlaiset [25]:

- Kirjoitetaan testi, joka ei mene läpi.
- Kirjoitetaan koodi, joka läpäisee testin. Tässä vaiheessa ei ole vaatimuksia koodin optimoinnille tai laadulle.
- Ajetaan testit ja todetaan niiden menevän läpi.
- Refaktorointi. Refaktoroidaan kohdassa kaksi kirjoitettua koodia, eli esimerkiksi poistetaan siitä kaikenlaiset epäloogisuudet, siirretään metodi loogisesti oikeaan paikkaan ja nimetään se järkevästi.
- Aloitetaan prosessi alusta.

TDD:n voidaan katsoa edistävän työskentelyä ainakin neljästä eri näkökulmasta [15]. Ensiksikin kehittäjä saa jatkuvaa palautetta työstään viiveettä. Hän näkee heti toimiiko tekemänsä koodi oikein ja rikkooko se mahdollisesti muita toiminnallisuuksia ja niiden testejä. Tällöin kehittäjä pystyy heti korjaamaan virheensä, eikä niitä ehdi kasaantua kerralla paljoa.

Samalla tämä yhteen metodiin keskittyminen kerrallaan ohjaa ja jaksottaa kehittäjän työskentelyä. Tämä tarkoittaa sitä, että kehittäjä keskittyy yhteen tehtävään kerrallaan, kunnes saa valmiiksi toimivan testatun kokonaisuuden. Tämän myötä kehittäjien on välttämätöntä myös pilkkoa työtään pienempiin osiin siten, että tällainen tehtävä kerrallaan läpiviety kehitys on edes mahdollista [15]. Kun tehtävät pilkotaan riittävän pieniksi, kehittäjien on helpompi arvioida niihin kuluvaa aikaa, heillä on parempi käsitys siitä, mitä jonkin asian valmistuminen vielä vaatii ja he kykenevät hallitsemaan omia työn alla olevia töitään helpommin.

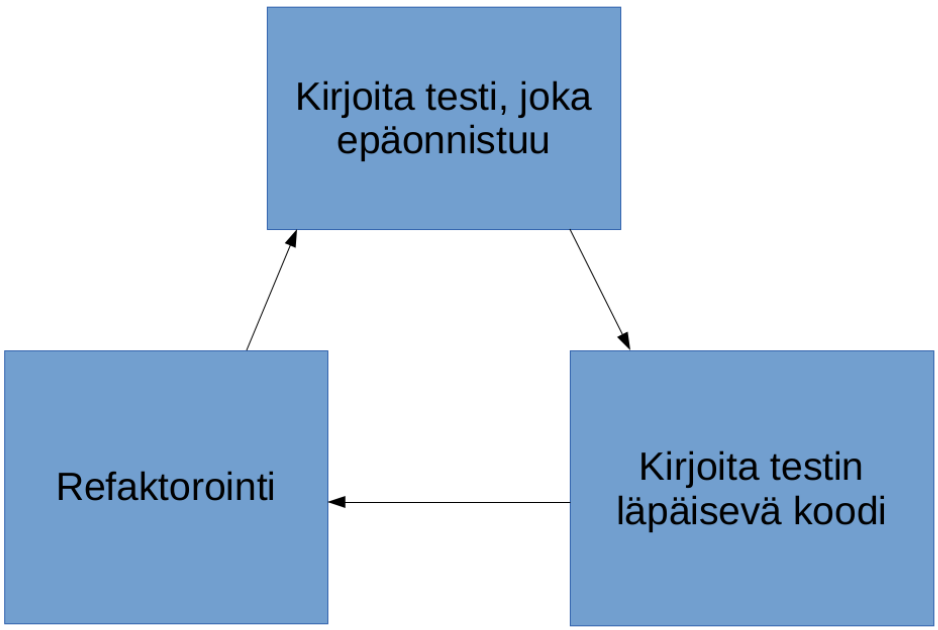
Kolmantena TDD tarjoaa omalta osaltaan yhden keinon laadunvarmistukseen. Kun ohjelmistoa kehitetään TDD:n keinoin, on jatkuvasti olemassa setti testejä, jotka parhaassa tapauksessa kattavat koko ohjelmiston, jos se on alusta alkaen kehitetty TDD:llä [15]. Sen lisäksi, että TDD:n myötä syntyvä testisetti on kattava, sitä myös ajetaan usein, sillä kaikkien uusien ominaisuuksien yhteydessä tulisi tarkistaa aina kaikkien testien toimivuus, jotta vältytään mahdollisilta uusien ominaisuuksien luomilta sivuvaikutuksilta.

Viimeiseksi TDD:ssä keskitytään ennen kaikkea matalalla tasolla koodin testaamiseen. Kun liikutaan yksittäisten metodien tasolla, on ohjelmoijalla usein hyvä näkemys siitä, mitä kirjoitettavan metodin olisi tarkoitus tehdä. Tällöin koodin laatu

paranee ainakin tällä matalalla tasolla, mutta TDD ei ota kantaa, eikä auta ohjelmoijaa ymmärtämään korkeamman tason toiminnallisia vaatimuksia tai suurempia kokonaisuuksia [25]. Tähän ongelmaan ratkaisuksi on lähivuosina kehitetty toimintatapoja, joilla pystytään hallitsemaan paremmin edellämainittujen korkeamman tason vaatimusten täyttymistä. Näitä menetelmiä ovat muun muassa Acceptance Test-Driven Development (ATDD) sekä Behaviour-Driven Development (BDD), joita käsitellään tarkemmin jäljempänä.

Vuonna 2005 julkaistussa tutkimuksessa [15] suurimpana kysymyksenä oli selvittää TDD:n vaikutusta tuottavuuteen. Tutkimuksessa todettiin testilähtöisen kehityksen todella lisäävän kehittäjien tuottavuutta. Tähän löydettiin tutkimuksessa [15] useita syitä. Yksi näistä oli jo edelläkin mainittu tehtävien pilkkominen ja sen myötä niihin keskittyminen. Näillä toimenpiteillä tehtävät ovat helpommin ymmärrettäviä kokonaisuuksia, jotka voidaan helposti toteuttaa kerralla. Tutkimus myös toteaa [15], että testivetoisessa kehityksessä huonot ja tehottomat toimintatavat hylätään verrattain nopeasti ja ne korvataan paremmilla. Useimmissa prosessimalleissa tällaiset päätökset tekevät kehittäjät itse, eli näitä parannuksia ei osoiteta mistään ylemmältä taholta. Esimerkiksi Scrumissa kehitystiimi itse käy jokaisen sprintin jälkeen retrospektiivissa läpi omaa toimintaansa ja yrittää löytää siitä kehityskohteita [9]. Tämä jatkuva toimintatapojen tehostaminen johtaa myös nopeaan oppimiseen. TDD laskee myös kynnystä asioiden uudelleen tekemiselle ja refaktoroinnille [15]. Kun ohjelmisto on toteutettu pienistä erikseen testatuista osista, on helppoa lähteä korjaamaan yhtä hajonnutta testiä, sillä hajoamisen syy on usein helppo löytää, kun testi kattaa vain yhden pienen osan ohjelmistosta, esimerkiksi yhden metodin, jonka tehtävän pitäisi olla helposti selitettävä.

Kaikenkaikkiaan vuoden 2005 tutkimuksessa [15] päädyttiin tulokseen, jonka mukaan testivetoinen kehitys todellakin parantaa työn tuottavuutta edellämainittujen seikkojen myötä. Samalla tutkimuksessa kuitenkin todetaan, että testivetoinen ohjelmistokehitys ei suoranaisesti paranna ohjelmistojen laatua. Tämä johtui siitä, että vaikka ohjelmistokehittäjät kirjoivat TDD:tä käyttäessään määrällisesti enemmän testejä, testien laatu riippui hyvin paljon kehittäjän kokemuksesta tehdä töitä testivetoisesti [15]. Testivetoisessa kehityksessä testeillä osaltaan ohjataan kehittäjän työtä, eli kehittäjä kertoo testeillä itselleen mitä hänen pitäisi seuraavaksi tehdä. Taito kirjoittaa hyviä testejä, jotka testaavat oikeita asioita syntyy vain testejä kirjoittamalla, eli tässä tilanteessa kokemus ja työskentelytapaan tottuminen tuovat lisää tehokkuutta työskentelyyn. Toisaalta myös matalan tason yksikkötesteistä saatavan hyödyn voitiin katsoa häviävän siinä, että kehittäjät pystyivät monissa tilanteissa ohjelmaa tarkastelemalla helposti toteamaan saman asian kuin mitä yksikkö-



Kuva 3.4: TDD-työvaiheet

testi testaa [15].

3.4 ATDD

ATDD eli Acceptance Test-driven development tarkoittaa nimensä mukaisesti hyväksymistestilähtöistä ohjelmistokehitystä. ATDD:ssä kehittäjät, käyttäjät ja testaajat yhdessä määrittelevät ohjelmistolle hyväksymiskriteerit, joiden pohjalta ohjelmistoa lähdetään kehittämään [23].

ATDD:llä ja BDD:llä tarkoitetaan useissa yhteyksissä samaa asiaa, sillä termit ovat alalla vielä niin uusia, eikä vakiintuneita käytäntöjä ole. Tämän myötä termistö ja nimet muuttuvat jatkuvasti kehityksen myötä. Hyvin pitkälti ATDD:n ja BDD:n ero voidaan nähdä löytyvän ennemminkin siinä, minkä tasoiseen testaamiseen niissä keskitytään [23]. Toinen tasoista on ATDD:n hyväksymistestilähtöinen malli, jossa keskitytään automaattisiin hyväksymistesteihin, joilla määritellään selviä vaatimuksia kehittäjätiimille, jotka toteutetun järjestelmän tulee täyttää [20].

BDD sen sijaan toimii tasolla, jossa määritellään erilaisia skenaarioita, joiden ta-

valla järjestelmän tulisi toimia. BDD:n mallissa keskeistä on ennenkaikkea luoda ymmärrystä kehittäjien ja muiden sidosryhmien, kuten esimerkiksi asiakkaan edustajien, välille [20]. Tämän ohella kuitenkin BDD auttaa myös välttämään toiminnallisia regressiobugeja aivan kuten ATDD:kin. Kummatkin tasoista toteuttavat kuitenkin myös osaltaan esimerkein määrittelyn periaatteita ja estävät toiminnallisia regressiobugeja syntymästä. ATDD:n tason voidaan nähdä myös soveltuvan paremmin tietynlaisiin toimintaympäristöihin, kun taas BDD:n taso toimii toisaalla paremmin. Gojko Adzic kirjoittaa Specification by Example -kirjassaan [20] ATDD:n mallin olevan hyödyllisempi, jos ollaan toteuttamassa ohjelmistoa, jossa on useita laatuun liittyviä haasteita. BDD:n tason taas voidaan nähdä sopivan paremmin tilanteeseen, jossa ei ole suurempia ongelmia ja BDD:tä voidaankin käyttää lähinnä selittämään asioita.

Kumpikin malli kuitenkin tuottaa erittäin hyvää dokumentaatiota järjestelmästä ilman erillistä panostusta sen tekemiseen, ja tämän voidaankin nähdä olevan yksi suurimmista eduista, joita saavutetaan käyttämällä ATDD:tä tai BDD:tä [20]. Haastatteluissaan Adzic sai kirjassaan selville jopa sen, että hyvin tehtynä selkeät, ylläpidettävät testit monesti jopa riittivät ainoaksi dokumentaatioksi järjestelmästä, sillä niiden käyttö oli helpompaa kuin erillisten dokumenttien tuottaminen [20]. Useat BDD- ja ATDD-työkalut, kuten esimerkiksi Cucumber [21] myös tukevat automaattisesti esimerkiksi nettisivujen luontia testeistä, jolloin niistä saadaan dokumentaation kaltainen artefakti hyvin pienellä vaivalla.

4 Behaviour-driven development

4.1 BDD:n lähtökohdat ohjelmistokehitystä tukevana menetelmänä

BDD, eli Behaviour-driven development, on nimensä mukaisesti ohjelmistokehityksen malli, jossa ohjelmistojen kehityksen lähtökohtana on niiden käyttäytyminen [19]. Toisin kuin edellä käsitellyssä ATDD:ssä, voidaan BDD:n painottavan enemmän kommunikaation merkitystä sidosryhmien välillä kuin vain automaattisten hyväksymistestien luontia. Yksi keinoista, joilla monissa BDD-testikehyksissä parannetaan kommunikaatiota ja yhteisymmärrystä eri sidosryhmien, kuten ohjelmistokehittäjien ja toimialan asiantuntijoiden välillä, on lähellä luonnollista kieltä oleva kieli, jolla testejä määritellään [17]. Esimerkiksi seuraavanlaisella muotoilulla voidaan määritellä testiskenaario Cucumber-testikehyksessä [21]:

```
Feature: Generation
  In order to finish the thesis
  As a student
  I want to generate good BDD-examples

  Scenario: Generate example of behaviour
    Given I have program running
    When I press generate
    Then the result should be an example behaviour on the screen
```

Kuten skenaarion muotoilusta voidaan nähdä, BDD:ssä testataan ennen kaikkea sitä, toimiiko ohjelmisto oikein ja halutulla tavalla, eli sen käyttäytymistä. Verrattuna TDD:n matalan tason testeihin, joissa keskitytään yhden metodin toimintaan kerrallaan, ero on merkittävä. Yksi merkittävistä asioista, joiden avulla BDD:llä voidaan ohjata ohjelmistokehitysprosessia on tapa, jossa toimintojen tuottama lisäarvo otetaan huomioon. Tällöin eniten lisäarvoa tuottavia ominaisuuksia voidaan ensimmäisenä lähtea kehittämään ja niille voidaan luoda BDD-testit, jotka nimetään sen mukaan, mitä järjestelmän halutaan tekevän [19].

Monilta osin BDD perustuu Test-driven developmentin parhaisiin käytänteisiin ja samalla BDD:n tarkoituksena on formalisoida niitä [17]. TDD:n voidaankin ajatel-

la auttavan kehittäjiä toteuttamaan ratkaisun oikein, kun taas ATDD ja BDD auttavat kehittäjiä toteuttamaan oikean ratkaisun [17]. Sekä TDD:stä, että BDD:stä kummastakin löytyy komponentit, joilla on helppo selventää eroja näiden kahden menetelmän välillä. Siinä missä TDD:ssä keskitytään testeihin, joilla testataan yhtä luokkaa ja sen metodien toimintaa, BDD:ssä vastaava komponentti on esimerkki, jolla kuvaillaan yhden luokan toivottua käyttäytymistä [26]. Siinä missä TDD:ssä testien odotetaan menevän läpi, BDD:ssä halutaan ohjelmiston käyttäytyvän oikein. BDD:ssä testeillä varmistetaan, että toiminnallisuus on oikea, eli tuottaa mahdollisimman paljon arvoa asiakkaalle, kun TDD:ssä testit testaavat toiminnallisuutta vain matalalla tasolla [26], eivätkä ne ota asiakkaan tarpeita niinkään huomioon.

4.2 Teknologioista

4.2.1 Yleistä teknologioista

Behaviour-driven developmentia varten on olemassa useita eri testikehyksiä, jotka soveltuvat eri tarkoituksiin. Eräitä esimerkkejä näistä ovat Cucumber [21], Jasmine [27], Specflow [28] ja RSpec [29]. Mainittujen testikehysten suurin ero on alustat, joita ne tukevat. Jasmine on JavaScript-ohjelmointikielelle tehty testikehys, kun taas SpecFlow on ratkaisu Microsoftin .NET-alustan BDD-testaamiseen. Cucumber ja RSpec pohjautuvat molemmat Ruby-ohjelmointikieleen, mutta ne poikkeavat toisistaan kuitenkin muuten varsin paljon. Siinä missä RSpec on ainoastaan Rubya tukeva kehys, jolla voidaan toteuttaa niin TDD- kuin BDD-testaustakin, niin Cucumber kehyksenä korostaa erityisesti määrittelyjen luettavuutta ja ymmärrettävyyttä. RSpec ei tarjoa mahdollisuutta määritellä erikseen lähellä selkokieltä olevia kuvauksia käyttäytymisestä, kuten Cucumberissa, vaan siinä koko testi on yhdessä yksittäisessä tiedostossa. Tällöin luettavuusero on varsin merkittävä, kun otetaan huomioon BDD:n lähtökohta kommunikaation parantajana:

RSpec:

```
# example_spec.rb

require 'spec_helper'
describe "exampletest" do
  it "should have a page titled Example at '/example'" do
   get '/example'
   response.should have_xpath("//title", :text => "Example")
 end
```

```
end
```

Vastaavasti Cucumberissä on erillisessä feature-tiedostossa määritelty luonnollisen kaltaisella kielellä eri skenaariot, joiden mukaan ohjelman halutaan toimivan. Tämän lisäksi on sitten erillinen Rubylla kirjoitettu tiedosto, jossa määritellään, mitä feature-tiedostossa käytetyt stepit tekevät. Tässä siis selvästi erotellaan liiketoiminnan kannalta tärkeät määrittelyt erikseen lähes selkokieliseen muotoon, jolloin niiden tulkinta on verrattain helpompaa kuin esimerkiksi RSpecin tapauksessa suoraan Ruby-koodin lukeminen. Edellä esitellyn RSpec-testin kaltainen testi Cucumberilla voitaisiin toteuttaa esimerkiksi seuraavalla tavalla:

Cucumber:

```
# example.feature

Scenario: As a user I want to see correct page title
  Given I am on the home page
  When I go to example page
  Then I should be on the page with the title: "Example"

# example_steps.rb

Given /^I am on the home page$/ do
  visit '/'
end

When /^I go to the example page$/ do
  visit '/example'
end

Then /^I should be on the page with the title: "([^"]*)"$/
do |page_title|
  response.should have_xpath("//title", :text => "#{page_title}")
end
```

Esimerkeistä voidaan huomata, kuinka Cucumberilla esimerkkiskenaariosta tulee paljon helpommin luettava henkilöille, jotka eivät ole teknisesti orientoituneita. Siinä missä RSpec-testissä samaan tiedostoon on määritetty testin kuvaus kuin sen toiminnalisuuskin, niin Cucumberissa testi on selvästi erikseen kuvattu feature-tiedostossa, jolloin eri sidosryhmien edustajat voivat keskittyä vain sen sisältöön,

eikä tekninen aineisto, eli eri steppien määritykset, ole sekoittamassa asiaan perehtymätöntä henkilöä.

Yhteistä lähes kaikille BDD-testikehyksille on niiden käyttämä testitapausten muoto yleisellä tasolla. Dan North esittelee tämän muodon Behaviour-driven development konseptia esittelevässä blogitekstissään [19]. Perinteinen pohja käyttäjätarinoille koostuu usein jotakuinkin seuraavanlaisista osista:

- Roolissa A
- Haluan tehdä asian B
- Jotta tapahtuu asia C

Tämä kyseisenlainen pohja toimii myös Northin määrittelemän testitapausten muodon pohjana. Tästä mallista North kehitti uuden mallin, jonka avulla voidaan käyttäjätarinat kirjoittaa suoraan hyväksyntätestien muotoon.

- Given (what is known/done beforehand)
- When (something is done)
- Then (something happens)

Tällaista Given-When-Then -muotoa käyttävää suuri osa BDD-testikehyksistä testiensä määrittämiseen. Poikkeuksiakin toki on, esimerkiksi aiemmin esimerkkinä ollut RSpec-testikehys Rubylle [29], jossa ei ole erikseen määrittelyjä testitapauksille, vaan kaikki sisältö löytyy koodin seasta. Esimerkiksi seuraavasta RSpec-testistä voidaan löytää vastaavat osat:

```
require 'spec_helper'

describe Example do
  it "is not valid without a title" do
    example = Example.new(title: nil)
    example.should_not be_valid
  end
end
```

Tässä voidaan ajatella, että pohjatietona, eli Given-osiona olisi tilanne, jossa on mahdollista luoda uusi Example. When-kohtana olisi Example.new(title: nil) eli luodaan uusi Example. Tällöin odotetaan tuloksena, että uusi Example ei olisi validi, mikä vastaisi siis Then-kohtaa Dan Northin muodossa. Voidaan kuitenkin miettiä,

onko välttämättä hyödyllistä yrittää puristaa jokaista testikehystä ennaltamääritettyyn muotoon, sillä RSpec poikkeaa muista kehyksistä muutenkin verrattain paljon sen tarjotessa mahdollisuudet laajemminkin testivetoiseen kehitykseen Rubylla kuin vain BDD:n toiminnallisuuksia.

Muista testikehyksistä esimerkiksi Cucumber [17] käyttää täysin Dan Northin määrittelemää muotoa [19] testeilleen, kun taas JavaScript-testikirjasto Jasminen [27] ratkaisu on hyvin samanlainen kuin RSpecin ratkaisu, eli Jasminessakaan ei ole erikseen määritelty luonnollisella kielellä testejä erikseen. Specflow [28], joka on testikehys .NET, Windows Phone ja Mono-alustoille, on sen sijaan toteutettu vastaamaan Dan Northin määrittelemää mallia, eli Given-When-Then -malliset määritykset testiesimerkeille toteutetaan siinä erikseen ja näille määrityksille ohjelmoidaan tulkit, joilla määritellään mitä testit tekevät.

4.2.2 Parsereista

Yksi tärkeä osa monia BDD-testikehyksiä on niiden käyttämät parserit, joilla tulkitaan määriteltyjä testitapauksia ajettavaksi ohjelmakoodiksi. Näillä parsereilla on kaikilla jokin hyvin määritelty syntaksinsa, jota ne ymmärtävät. Yksi parsereista on esimerkiksi avoimen lähdekoodin Gherkin [30], jota käyttävät Cucumber [21] ja sen sukuiset BDD-testikehykset, kuten esimerkiksi SpecFlow [28]. Gherkinin syntaksi on hyvin yksinkertainen, joka edesauttaa sitä, että eri sidosryhmien edustajat ymmärtävät ja parhaassa tapauksessa voivat jopa itse luoda uusia testitapauksia. Tällöin tarvitaan erityisesti kehitystiimin ja muiden sidosryhmien yhteistyötä, sillä tässäkin tapauksessa kehitystiimin tehtäväksi jää kirjoittaa testitapauksia vastaava ohjelmakoodi, jotta testitapauksista saadaan toimivia. Martin Fowler kirjoittaakin blogissaan [32], että ennen kaikkea tärkeää ja hyödyllistä on se, että sidosryhmien edustajat kykenet lukemaan ja ymmärtämään koodia, tai tässä tapauksessa testitapausten kuvauksia, sillä se toimii äärimmäisen hyvänä kommunikaatiovälineenä eri osapuolten välillä. Seuraavassa on esimerkki Gherkin-dokumentista, jossa määritellään esimerkkiskenaario:

```
Feature: Software installation
  Scenario: Uninstall software
    Given a software named "example" exists
    When I press uninstall software
    Then the software named "example" should be removed
```

Tässä esimerkissä ainoat kohdat, jotka määritellään Gherkinin syntaksissa, ovat kaikilla riveillä niiden alut, eli Feature, Scenario, Given, When ja Then [31]. Kaikki muu

sisältö, mitä testitapaukset sisältävät, on testien kirjoittajan itse luomaa sisältöä, eli kaikki muut kohdat tulee määritellä steppien kuvauksissa testikehyksen käyttämällä ohjelmointikielellä. Cucumberin tapauksessa tämä voitaisiin tehdä esimerkiksi Rubylla tai jollakin muulla ohjelmointikielellä, jota Cucumber tukee, kuten esimerkiksi Javalla. Edeltävän esimerkin Given-kohta voitaisiin määritellä toimivaksi testiksi esimerkiksi Java-ohjelmointikieltä käyttäen seuraavalla tavalla:

```
@Given("a software named (.*) exists")
public void softwareExists(string software) {
    // Tässä voitaisiin tehdä ohjelmistolla mitä tahansa,
    // esimerkissä halutaan hakea ohjelmistoa
    // sen nimellä SoftwareServiceltä
    getSoftwareService.findSoftware(software);
}
```

Gherkinin sijasta monet BDD-testikehykset käyttävät omia parsereitaan. Esimerkiksi RSpec [29] ja Jasmine [27] eivät käytä Gherkiniä, vaan niissä on toteutettu omat parserit testien tulkkaamiseen. Yksi eduista, joita saavutetaan yhden parserin yleisyydellä eri testikehyksissä, on se, että tämä edesauttaa testien uudelleenkäytettävyyttä. Kun eri testikehykset käyttävät samaa parseria, niin samat testimääritykset toimivat niille kaikille. Tämä onkin yksi merkittävä havainto etsittäessä ratkaisua tutkielman tutkimuskysymykseen mobiilialustojen BDD-testaamisesta yhdellä testisetillä. Tätä käsitellään myöhemmin lisää kappaleessa 7.

5 BDD:n tarjoamat liiketoiminnalliset edut

5.1 Asiakkaan käyttäjätarinoista hyväksymistesteiksi

5.2 BDD:n hyödyntäminen mobiilikehityksessä offshore-tiimeillä

6 BDD mobiilialustoilla

6.1 Natiivisovellukset

6.2 HTML5-sovellukset

7 Crossplatform-testaaminen eri mobiilialustoilla

7.1 iOS BDD-frameworkit

7.2 Android BDD-frameworkit

7.3 Windows phone BDD-frameworkit

7.4 Mahdollisuudet testata kaikki alustat yhdellä testisetillä

8 Pohdintaa

9 Yhteenveto

Yhteenveto tähän

Lähteet

[1] Levy, Y. & Ellis, T. J. *A Systems Approach to Conduct an Effective Literature Review in Support of Information Systems Research*, Informing Science Journal, 9, 2006.

[2] Gordana Dodig Crnkovic *Constructive Research and Info-Computational Knowledge Generation*, School of Innovation, Design and Engineering, Computer Science Laboratory, Mälardalen University, Sweden

[3] *Gartner Smart Phone Marketshare 2013 Q4*, Gartner, Inc., 2013, saatavilla WWW-muodossa <URL: `http://www.gartner.com/newsroom/id/2665715`, viitattu 02.04.2014.

[4] Craig Larman ja Victor Basili, *Iterative and incremental developments. a brief history*, IEEE Computer Society, Volume: 36, Issue: 6, 2003.

[5] Valtiokonttori, Valtion IT-palvelukeskus, *Ketterän palvelukehityksen ostaminen*, 2013.

[6] Winston Royce, *Managing the development of large software systems*, Proceedings of IEEE WESCON, 1970.

[7] Barry Boehm, *A spiral model of software development and enhancement*, ACM SIGSOFT Software engineering notes vol 11 no 4, 1986.

[8] Barry Boehm, *Spiral Development: Experience, Principles and Refinements*, Spiral Development Workshop, February 9, 2000.

[9] Ken Schwaber ja Jeff Sutherland, *The Scrum Guide*, Scrum.org, 2013.

[10] *Scrum process*, saatavilla WWW-muodossa <URL: `http://en.wikipedia.org/wiki/File:Scrum_process.svg`>, viitattu 24.03.2014.

[11] David Starr, *Effective Sprint Retrospectives*, 2012, saatavilla WWW-muodossa <URL: `http://msdn.microsoft.com/en-us/library/jj620912.aspx`, viitattu 26.03.2014.

[12] Ming Huo, June Verner, Liming Zhu ja Muhammad Ali Babar, *Software Quality and Agile Methods*, Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International

[13] Martin Fowler, *Continuous Integration*, 2006, saatavilla WWW-muodossa <URL: http://martinfowler.com/articles/continuousIntegration.html, viitattu 26.03.2014.

[14] *Jenkins - An extendable open source continuous integration server*, saatavilla WWW-muodossa <URL: http://jenkins-ci.org/, viitattu 27.03.2014.

[15] Hakan Erdogmus, Maurizio Morisio ja Marco Torchiano, *On the Effectiveness of the Test-First Approach to Programming*, IEEE Transactions On Software Engineering, Vol. 31, No. 3, 2005

[16] *Agile Manifesto*, 2001, saatavilla WWW-muodossa <URL: http://agilemanifesto.org/>, viitattu 12.02.2013.

[17] Matt Wayne ja Aslak Hellesøy, *The Cucumber Book, Behaviour-Driven Development for Testers and Developers*, Pragmatic Programmers LLC, 2012.

[18] Carlos Solís ja Xiaofeng Wang, *A Study of the Characteristics of Behaviour Driven Development*, 37th EUROMICRO Conference on Software Engineering and Advanced Applications, 2011.

[19] Dan North, *Introducing BDD*, 2006, saatavilla WWW-muodossa <URL: http://dannorth.net/introducing-bdd/>, viitattu 27.01.2013.

[20] Gojko Adzic, *Specification by Example*, Manning Publications Co, 2011.

[21] *Cucumber - Making BDD fun*, saatavilla WWW-muodossa <URL: http://cukes.info/>, viitattu 31.03.2014.

[22] Børge Haugset ja Tor Stålhane, *Automated Acceptance Testing as an Agile Requirements Engineering Practice*, 45th Hawaii International Conference on System Sciences, 2012.

[23] *What is TDD, BDD & ATDD?*, 2012, saatavilla WWW-muodossa <URL: http://assertselenium.com/2012/11/05/difference-between-tdd-bdd-atd viitattu 31.03.2014.

[24] Gojko Adzic, *Bridging the Communication Gap, Specification by Example and Agile Acceptance Testing*, Neuri Limited, 2009.

[25] Susan Hammond ja David Umphress, *Test driven development: the state of the practice*, ACM-SE '12 Proceedings of the 50th Annual Southeast Regional Conference, s. 158-163, 2012.

[26] Liz Keogh, *Translating TDD to BDD*, 2009, saatavilla WWW-muodossa <URL: http://lizkeogh.com/2009/11/06/translating-tdd-to-bdd/>, viitattu 03.04.2014.

[27] *Jasmine: Behavior-Driven JavaScript*, saatavilla WWW-muodossa <URL: http://jasmine.github.io/>, viitattu 07.04.2014.

[28] *SpecFlow - Pragmatic BDD for .NET*, saatavilla WWW-muodossa <URL: http://www.specflow.org/>, viitattu 07.04.2014.

[29] *RSpec.info*, saatavilla WWW-muodossa <URL: http://rspec.info/>, viitattu 07.04.2014.

[30] *Gherkin*, saatavilla WWW-muodossa <URL: https://github.com/cucumber/gherki viitattu 09.04.2014.

[31] *Cukes.info - Gherkin*, saatavilla WWW-muodossa <URL: http://cukes.info/gherkin.html>, viitattu 10.04.2014.

[32] Martin Fowler, *BusinessReadableDSL*, saatavilla WWW-muodossa <URL: http://www.martinfowler.com/bliki/BusinessReadableDSL.html>, viitattu 10.04.2014.

[33] IBM developerWorks: http://www.ibm.com/developerworks/java/library/j-cq09187/index.html