Pinja Partinen

Testivetoisen ohjelmistokehityksen kehittyminen

Tietotekniikan kandidaatintutkielma

30. huhtikuuta 2022

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Pinja Partinen

Yhteystiedot: pinja.m.partinen@student.jyu.fi

Ohjaaja: Jonne Itkonen

Työn nimi: Testivetoisen ohjelmistokehityksen kehittyminen

Title in English: Development of Test-Driven Development

Työ: Kandidaatintutkielma

Opintosuunta: Kaikki opintosuunnat

Sivumäärä: 18+0

Tiivistelmä: Tässä kirjoitelmakatsauksessa käsitellään testivetoisen ohjelmistokehityksen kehittymistä. Perehtymällä aiempiin tutkimuksiin testivetoisesta ohjelmistokehityksestä selvitetään millainen testivetoinen ohjelmistokehitys on ja sen taustat. Lisäksi tutkitaan millaisten ohjelmistojen kehitykseen tätä lähestymistapaa yleensä hyödynnetään ja miten.

Avainsanat: Testivetoinen ohjelmistokehitys, TDD, Ketterä kehitys, eXtreme Programming

Abstract: In this literature review development of test-driven development is being processed. What kind of test-driven development is and its background are being researched by taking a look to earlier researches of test-driven development. In addition is researched for what kind of projects test-driven development is generally used and how.

Keywords: Test-Driven Development, TDD, Agile Development, eXtreme Programming

Termiluettelo

Testivetoinen ohjelmistokehitys	Testivetoinen ohjelmistokehitys (Test-Driven Development, TDD) on suunnittelu- ja analyysimenetelmä, jossa testi kirjoitetaan ennen sitä vastaavaa koodi pätkää (Beck 2001).
Ketterä kehitys	Ketterä kehitys (engl. Agile Development) on ohjelmistokehityksen suuntaus jossa arvostus painottuu toimivaan ohjelmistoon, muutoksiin vastaamiseen, asiakasyhteistyöhön sekä yksilöihin ja kanssakäymiseen.
eXtreme Programming	eXtreme Programming (XP) on ketterä menetelmä, jonka käytänteisiin kuuluu testivetoinen ohjelmistokehityksen hyödyntäminen (Beck 2002).
Testaus-ensin	Testaus-ensin (engl. Test-First, lyhennettynä TF) tavassa testi kirjoitetaan ennen sitä vastaavaa koodi pätkää.
Automaatiotestaus	Automaatiotestaus (Test automation) on testauksen käytäntötapa, jossa testaustyökalut hallitsevat testien suoritusta ja vertaavat testien tuloksia odotettuihin tuloksiin (Nass, Alégroth ja Feldt 2021).
Refaktorointi	Koodia refaktoroidessa sen rakennetta muutetaan kuitenkin toiminnallisuuden pysyessä samana. TDD:ssä hyödynnetään refaktorointia (Fowler 2000).
Yksikkötestaus	Yksikkötestaus (Unit testing) on testausmenetelmä, jossa testataan ohjelmiston pieniä paloja.

Sisällys

1 Johdanto

Vaikka testivetoisuus on yleisessä tiedossa oleva lähestymistapa ohjelmistokehityksessä, on ohjelmoijien keskuudessa kuitenkin väärinkäsityksiä siitä mitä testivetoinen ohjelmistokehitys tarkoittaa (Janzen ja Saiedian 2008). Tutkielmassa kartoitetaan testivetoisen ohjelmistokehityksen piirteitä ja tarkoitusperää perehtymällä testivetoisen ohjelmistokehityksen kehittymiseen. Tämän tarkoituksena on luoda selkeä kuva mikä testivetoinen ohjelmistokehitys on ja miksi se on kehittynyt sellaiseksi kuin se nykypäivänä tunnetaan. Mikäli ohjelmoijilla olisi parempi käsitys testivetoisesta ohjelmistokehityksestä, saatettaisiin sitä hyödyntää useammin ja tehokkaammin.

Tutkielmassa käydään läpi millaista testivetoinen ohjelmistokehittäminen on nykyään ja mistä se on alunperin lähtenyt liikkeelle. Selvitetään kuinka se mukautuu erilaisten ohjelmistojen kehittämiseen, miten ohjelmoijat suhtautuvat testivetoiseen ohjelmistokehitykseen ja kuinka yleistä sen käyttö on. Perehtymällä testivetoisen ohjelmistokehityksen taustoihin ja nykytilanteeseen voidaan saada kuva millainen tulevaisuus testivetoisella ohjelmistokehityksellä mahdollisesti tulee olemaan.

Tutkimuksen kysymyksiin voidaan saada vastaukset selville kartoittamalla tutkimusartikkeleita laajalta aikaväliltä testivetoiseen ohjelmistokehitykseen, testaukseen ja ketterään kehitykseen liittyen. Näin ollen tämän tutkimuksen metodina ja strategiana käytetään kirjallisuuskartoitusta, sillä se soveltuu testivetoisen ohjelmistokehityksen kehittymisen tutkimiseen. Tutkimuksen aineistona on käytetty testivetoiseen ohjelmistokehitykseen liittyviä kirjoja, sekä JYKDOK:in kansainvälisten e-aineistojen haun (https://jyu.finna.fi/Primo/Advanced) kautta löytyneitä ja saatavilla olleita artikkeleita.

Seuraavaksi aloitetaan käymällä läpi testivetoisen kehittymisen kannalta keskeisiä ohjelmistokehitysken piirteitä, menetelmiä ja käsitteitä, jotta saadaan selkeämpi kuva testivetoisen ohjelmistokehityksen taustoista. Sen jälkeen perehdytään syvemmin itse testivetoiseen ohjelmistokehitykseen, sekä sen käyttökohteisiin ja miten testivetoisesta ohjelmistokehityksestä voitaisiin saada paras mahdollinen hyöty irti. Lopuksi käydään läpi mitä yleisiä väärinkäsityksiä aiheesta on ja millainen tulevaisuus testivetoisella ohjelmistokehityksellä voisi olla.

2 Teoreettinen tausta

Tässä luvussa esitellään testivetoisen ohjelmistokehityksen kehittymisen kannalta keskeisiä aiheita. Selvittämällä mitkä asiat ovat vaikuttaneet testivetoisen ohjelmistokehityksen kehitykseen, voidaan ymmärtää paremmin miksi testivetoinen ohjelmistokehitys on sellainen kuin se on. Muodostamalla parempi ymmärrys testivetoisen ohjelmistokehityksen taustoista, voidaan seuraavassa luvussa syventyä tarkemmin siihen mitä testivetoinen ohjelmistokehitys on.

2.1 Ketterä kehitys

Ketterä kehitys on ohjelmistokehityksen lähestymistapa, jossa ilmenevät Ketterän ohjelmistokehityksen -julistuksessa (Beck ym. 2001) esille tulevat arvot ja piirteet, jotka ovat seuraavanlaiset. Ketterässä kehityksessä ensisijaisesti arvostus painottuu toimivaan ohjelmistoon, muutoksiin vastaamiseen, asiakasyhteistyöhön, sekä yksilöihin ja kanssakäymiseen. Toissijaisena, mutta kuitenkaan unohtamatta, tulevat kattava dokumentaatio, suunnitelmissa pitäytyminen, sopimusneuvottelut sekä menetelmät ja työkalut. Käytännössä nämä piirteet voivat ilmetä muun muassa iteratiivisena projektin etenemisrytminä, sekä säännöllisesti asiakkaiden kanssa pidettävinä kokouksina läpi projektin elinkaaren.

Ketterän kehityksen alle kuuluvat ketterät menetelmät, jotka omaavat ketterälle kehitykselle ominaiset piirteet. Ketterä menetelmä on kokonaisuus harjoitteita, jotka ohjaavat ohjelmistokehityksen kulkua. Ketteriä menetelmiä ovat esimerkiksi Scrum ja eXtreme Programming. Huomattakoon, että eräs Ketterän kehityksen -julistuksen kirjoittajista on Kent Beck, joka on yksi eXtreme Programmingin sekä Testivetoisen ohjelmistokehityksen kehittäjistä.

2.2 Smalltalk

Smalltalk on puhdas olio-ohjelmointikieli, jossa kaikki asiat ovat olioita. SUnit on Smalltalkissa käytössä oleva yksikkötestaustyökalu. Se on ensimmäinen xUnit yksikkötestaustyökaluperheen jäsen (Beck 2002, s.158).

Ward Cunningham ja Kent Beck aloittivat työskentelyn Smalltalkin parissa 1980-luvulla. He kehittivät Smalltalkin kaltaisille ympäristöille suunnitellun ohjelmistokehitys tavan nimeltä eXtreme Programming. (Fowler 2000, s. 71)

2.3 eXtreme Programming (XP)

Extreme Programming (lyhennettynä XP) on ketterä menetelmä, jonka käytänteisiin kuuluu testivetoinen ohjelmistokehityksen hyödyntäminen (Beck 2002). XP on alunperin kehitetty vastaamaan pienien tiimien erityistarpeisiin, jotka kohtaavat epäselvästi määriteltyjä ja muuttuvia vaatimuksia ohjelmistokehityksessä (Beck 2000). Kent Beck ja Cynthia Andres uudelleen kirjoittivat kirjan *Extreme programming explained: embrace change* (Beck 2000). *Extreme programming explained: embrace change, Second edition* -kirjassa XP:n yhteensopivuus laajentui kaiken kokoisille tiimeille (Beck ja Andres 2004). Extreme Programmingiin kuuluu viisi arvoa. Kommunikaatio (engl. communication), yksinkertaisuus (engl. simplicity), palaute (engl. feedback) ja rohkeus (engl. courage) ovat neljä alkuperäistä arvoa (Beck 2000). Beck ja Andres (2004) lisäsivät arvoihin kunnioituksen (engl. respect).

Yksikkötestaus ja toiminnallinen testaus ovat keskeisimpiä testaustapoja XP:ssä. Hyödynnettävästä testivetoisesta ohjelmistokehityksestä tulee sivutuotteena yksikkötestit, ja 100 % niistä ajetaan jokaisella yksikkötestien ajokerralla. Asiakkaat osallistuvat toiminnallisen testauksen kehitykseen. XP-tiimissä tulisi olla ainakin yksi testaaja, joka auttaa toteuttamaan asiakkaiden ajatusten pohjalta suunniteltuja toiminnallisia testejä. Yksikkötestien ja toiminnallisten testien tulisi täyttää seuraavat kaksi vaatimusta. Testien tulee olla muista testeistä riippumattomia. Jos testit riippuvat toisistaan, virhe joka voisi tulla ilmi yhdessä testissä, voi aiheuttaa virheen näkymisen usean testin epäonnistumisena. Tämän takia virheen aiheuttajan löytämiseen saattaa kulua enemmän aikaa, kuin jos virhe näkyisi vain testissä johon se vaikuttaa suoraan. Toinen testauksen vaatimus on, että testit ovat automatisoituja. Näin testatessa voidaan säästää aikaa sekä välttyä inhimillisiltä virheiltä, joita stressi ja väsymys voivat aiheuttaa. (Beck 2000)

2.4 Testaus-ensin (Test-First)

Testaus-ensin-menetelmää käytetään osana muun muassa testivetoista (Karac ja Turhan 2018) sekä käyttäytymisvetoista ohjelmistokehitystä (Behavior-Driven Development, BDD) (Aghayi ym. 2021). Testaus-ensin-menetelmässä testi kirjoitetaan ennen sisältöä, jonka tulisi läpäistä testi. Kun koodi läpäisee testin, siirrytään tekemään seuraavaa sisältöpalaa. Vaikka testivetoinen ohjelmistokehitys on saanut alkunsa testaus-ensin nimen alla (Beck 2001), ne eivät ole enää täysin sama asia. Testaus-ensin-menetelmässä ohjelmiston suunnittelu ei ole yhtä keskeisessä osiossa testejä tehdessä kuin testivetoisessa ohjelmistokehityksessä. Testaus-ensin-menetelmässä tiedetään mitä koodilta halutaan, ja testien tarkoituksena on avustaa pitämään huolta koodin eheydestä ja ohjata tekemään sitä siisteissä paloissa.

2.5 Automaatiotestaus

Automaatiotestaus on testauksen käytäntötapa, jossa hyödynnetään testaustyökaluja. Nämä testaustyökalut hallitsevat testien suoritusta ja vertaavat testien tuloksia odotettuihin tuloksiin (Nass, Alégroth ja Feldt 2021). Automatisoidut testit vievät vähemmän aikaa verrattuna manuaaliseen testien hallintaan sekä vähentävät inhimillisten virheiden riskiä (Beck 2000). Automaatiotestausta hyödynnetään osana testaus-ensin ja testivetoista ohjelmistokehitystä yksikkötestien hallitsemiseksi edellä mainittujen hyötyjen vuoksi.

3 Testivetoinen ohjelmistokehitys

”The goal is clean code that works.” (Beck 2002, Ron Jeffries)

Testivetoinen ohjelmistokehitys on ohjelmiston suunnittelu- ja analyysimenetelmä (Beck 2001) sekä eräs ketterän kehityksen lähestymistavoista. Tässä suunnittelutavassa tehdään lähes poikkeuksetta testi ennen kuin kirjoitetaan riviäkään koodia uutta toiminnallisuutta varten. Kun testi tehdään ennen koodia on tilanteen analysointi välttämätöntä. Tehdystä testistä käy ilmi mitä koodin lopputulokselta odotetaan ja mitä koodia tehdessä on otettava huomioon. Testin suunnittelu ohjaa ohjelmoijaa arvioimaan koodiin vaikuttavia ympäristötekijöitä. (Beck 2001)

Vuonna 2002 Kent Beck julkaisi kirjan ”Test-Driven Development: By Example”, jolloin testivetoinen ohjelmistokehitys sai nykyisen nimensä ”Test-Driven Development”, lyhennettynä TDD. Sitä ennen hän oli kehittänyt testivetoista ohjelmistokehitystä nimellä ”Test-First” (Beck 2001).

”Test-Driven Development: By Example” sisältää useita eri toimintamalleja testivetoiseen ohjelmistokehitykseen ja sen eri vaiheisiin liittyen. Testivetoista ohjelmistokehitystä harjoittaessa ei tarvitse käyttää kaikkia siihen liittyviä malleja, mutta niiden olemassaolon tietäminen voi auttaa saamaan paremmin hyödyt irti testivetoisesta ohjelmistokehityksestä. Tässä tutkielmassa tuodaan esille malleista vain muutama. Esimerkiksi eräs malli on ”Test List”, joka kehottaa pitämään yllä muistilistaa. Muistilistaan kirjataan mitä kehitettäviltä ominaisuuksilta odotetaan sekä mitä tarvitsee tehdä, ja kun jokin listan kohta toteutuu, se voidaan yliviivata listasta. (Beck 2002)

Kuten tutkielman alaluvussa 2.3 ”eXtreme Programming” tuli esille, TDD:ssä syntyy sivutuotteena yksikkötestejä, jotka kaikki käydään läpi jokaisella testien ajokerralla. Tässä huomautettakoon, että testivetoinen ohjelmistokehitys ei ota kantaa muiden testaustapojen, kuten suorituskykytestaus, rasitustestaus ja käytettävyystestaus, hyödyntämiseen. Testivetoisessa ohjelmistokehityksessä jatkuvasti läpikäytävät testit pitävät huolen, että ohjelmisto pysyy ehjänä auttamalla ohjelmoijaa huomaamaan virheet heti niiden tapahduttua. (Beck 2002)

3.1 Toimintasykli

Testivetoisessa ohjelmistokehityksessä edetään punainen-vihreä-refaktorointi (engl. Red/Green/Refactoring) syklimallin mukaisesti (Beck 2002). Punainen-vihreä-refaktorointi sykliä toistetaan kunnes ohjelmiston osa toimii halutulla tavalla. Aina tarvittaessa voidaan palata syklin edelliseen vaiheeseen. Koodin tekeminen pienissä sykleissä auttaa vastaamaan tarvittaviin muutoksiin nopeasti. Kuitenkin TDD:ssä tehtävien askelten koko (esimerkiksi punainen-vihreä-refaktorointi syklin koko) ei ole kiveen hakattu. Beck ohjastaa, että jos pienet askeleet tuntuvat rajoittavilta, ota isompia askelia ja mikäli olet epävarma jostakin ota pienempiä askelia. Hän myös sanoo, että ”TDD:ssä ei ole kyse pienen pienien askelten ottamisesta, vaan kyse on kyvystä voida ottaa pienen pieniä askelia”. (Beck 2002)

- **Punainen:** TDD:n toimintasykli aloitetaan punaisesta vaiheesta. Siinä tehdään ensin testi, joka ei vielä mene läpi vaan epäonnistuu eikä välttämättä edes käänny. Kun testi on tehty siihen kuntoon että se kääntyy, mutta ajaessa epäonnistuu, voidaan siirtyä seuraavaan vaiheeseen. (Beck 2002)
- **Vihreä:** Vihreässä vaiheessa on tarkoitus saada testeistä vihreät palkit näkyviin testaustyökaluun eli onnistunut läpiajo. Vaihe aloitetaan tekemällä koodi, jota muokataan kunnes se läpäisee testit. Tässä vaiheessa ei vielä välitetä koodin siisteydestä vaan on vain tarkoitus saada vihreä palkki mahdollisimman nopeasti. Voidaan kopioida vaikka aiemmin tehty vastaavanlainen koodi pätkä. Teeskentele (engl. Fake it), ilmeinen toteutus (engl. Obvious Implementation) ja kolmiomittaus (engl. Triangulation) ovat lähestymistapoja testin siistiksi ja toimivaksi saamiseksi. Tarvittaessa voidaan ottaa askel taaksepäin, jotta testiä voidaan tehdä tarkemmaksi, jonka jälkeen koodia muokataan uudelleen, jotta se läpäisisi testin. (Beck 2002)
- **Refaktorointi:** Aiemmassa vaiheessa tehtiin koodi joka toimii. Tässä vaiheessa on tarkoitus tehdä siitä koodista siistiä. Muokataan koodia niin ettei siinä ole mitään ylimääräistä tai tarpeetonta. Jos refaktorointivaiheessa tulee virhe, suositeltavin tapa on poistaa muutokset jotka aiheuttivat virheen, jotta taas on vihreä palkki ja sitten korjata virhe ja virheen korjauksen jälkeen lisätä muutokset uudelleen (Beck 2002, s. 47). Virheen korjaamista varten voidaan vaihtoehtoisesti aloittaa uusi kierros eli kirjoitetaan uusi testi virheen eristämiseksi (Beck 2002, s. 71). Keskeytys tapaa tulisi käyttää vain

jos uusi korjattava asia on pieni ja aiheutuva keskeytys lyhyt. Lisäksi keskeytystä ei tulisi koskaan keskeyttää (Beck 2002, s. 71, Jim Coplien). Jos refaktoroitavassa koodissa ei ole tarpeeksi kattavaa testausta, kirjoitetaan ennen koodin muuttamista testit asioille joilta ne puuttuvat (Beck 2002, s. 29). Ilman kattavia testejä refaktoroidessa saatetaan hajottaa jotakin, mikä ei käy ilmi testeissä, ja viallinen koodi voi aiheuttaa ongelmia myöhemmin. Testejä jotka olivat hyödyllisiä aiemman koodi rakenteen kanssa mutta tarpeettomia uudessa toteutuksessa, voidaan poistaa (Beck 2002, s. 53). (Beck 2002)

3.1.1 Toimintasykli viidessä osassa

Punainen-vihreä-refaktorointi mallin sijaan voidaan TDD:n sykli jakaa myös seuraavaan viiteen vaiheeseen:

1. Kirjoitetaan testi. (Punainen)
2. Tehdään testistä riittävän ehjä että se kääntyy. (Punainen)
3. Ajetaan testi, jotta nähdään sen epäonnistuvan. (Punainen)
4. Korjataan koodi sellaiseksi, että testi onnistuu. (Vihreä)
5. Poistetaan päällekkäisyydet. (Refaktorointi)

(Beck 2002, s. 24)

3.1.2 Vihreän vaiheen toimintamalleja

Vihreässä vaiheessa voidaan hyödyntää malleja Teeskentele (engl. Fake it), Ilmeinen Toteutus (engl. Obvious Implementation) ja Kolmiomittaus (engl. Triangulation):

- Teeskentele: "Palauta vakio ja vähitellen korvaa vakiot muuttujilla kunnes sinulla on oikea koodi"
- Ilmeinen toteutus: Ilmeisessä toteutuksessa kirjoitetaan suoraan oikea koodi toteutus. Jos haluttu lopputulos on selkeä ja ohjelmoija tietää miten se voidaan toteuttaa, voidaan käyttää Ilmeistä toteutusta.
- Kolmiomittaus: Olemassa olevaa koodia yleistetään niin että uusi ominaisuus voidaan toteuttaa sen avulla.

(Beck 2002, s. 13)

3.2 Käyttökohteet ja käyttäminen

Testivetoista ohjelmistokehitystä voi hyödyntää osana lähes mitä tahansa ohjelmointiprojektia, jossa automaatiotestausta voi käyttää. Vaikkakin se on alunperin kehitetty osana eXtreme Programmingia, sitä voi käyttää itsenäisestikin. Extreme Programming soveltuu kaikenkokoisten tiimien käyttöön (Beck ja Andres 2004). Näin ollen TDD:tä, joka on osa sitä, voidaan myös hyödyntää kaikenkokoisten tiimien käytössä.

Testivetoinen ohjelmistokehitys ei rajaa minkä ohjelmointikielien yhteydessä sitä voi käyttää. Kuitenkin Beck (2002, s. 198) tuo esille, että sellaisten ohjelmointikielen ja testaustyökalun yhteydessä, joissa TDD:n toimintasykli on hankalampi suorittaa, saattaa houkutella ohjelmoijaa isompien askelten ottamiseen kerrallaan. Bissi, Serra Seca Neto ja Emer (2016) tekemässä systemaattisessa kirjallisuuskatsauksessa käsiteltiin empiirisiä tutkimuksia jotka vertailivat TDD:tä ja Testaus-jälkeen (Test-Last Development, TLD). Vaatimukset täyttäviä tutkimuksia löytyi 27 kappaletta joista 21 käytti hyödykseen Javaa. Kent Beck käytti kirjansa *Test-Driven Development: By Example* (2002) esimerkeissä hyödyksi Javaa sekä Pythonia. Java vaikuttaa olevan suosituin ohjelmointikieli TDD:n yhteydessä.

Testivetoista ohjelmistokehitystä voi käyttää lähes kaikenlaiseen ohjelmistokehitykseen, mutta sekään ei sovellu ihan kaikkeen. Testivetoista ohjelmistokehitystä voi olla hankala hyödyntää käyttöliittymien kehittämisessä, ja näin ollen niiden yhteydessä TDD:n käyttö ei suositeltavaa (Beck 2001).

3.3 Testaustyökalut

Tässä alaluvussa esitellään mitä välineitä testivetoisessa ohjelmistokehityksessä yleensä hyödynnetään kuten xUnit. Nanthaamornphong ja Carver (2015) kyselytutkimukseen osallistui 64 henkilöä, joista 11 esitti testien kirjoittamisen hankaluuksien ylitsepääsemisen avuksi sopivien testaustyökalujen käyttämistä.

Testivetoisen ohjelmistokehityksen testien hallitsemiseksi yleensä hyödynnetään testaustyö-

kaluja. Yleisessä käytössä on eri ohjelmointikielien yksikkötestaustyökalut, xUnit on tällaisten työkalujen perhe. xUnit oli otettu käyttöön yli 30 ohjelmointikielessä jo vuonna 2002 (Beck 2002, s. 119). Käytettävän työkalun valintaan vaikuttaa erityisesti projektissa käytettävä ohjelmointikieli. JUnit työkalua voidaan käyttää Javassa ja PHPUnit työkalua PHP:ssä (https://phpunit.de/index.html). Ne ovat eräitä xUnitin esiintymiä.

4 Huomioita testivetoiseen ohjelmistokehitykseen liittyen

Tässä luvussa käsitellään ensin testivetoiseen ohjelmistokehitykseen liittyviä väärinkäsityksiä. Väärinkäsitykset aiheesta saattavat vaikuttaa ohjelmoijien päätökseen alkaa hyödyntämään tai alkaa etsimään lisätietoa testivetoisesta ohjelmistokehityksestä. Tämä saattaa johdattaa ohjelmoijia olemaan käyttämättä testivetoista ohjelmistokehitystä, kun se voisi olla hyödyksi, tai johdattaa käyttämään, kun se ei toisi tekemiseen lisäarvoa. Väärinkäsitykset saattavat myös hankaloittaa ottamasta parasta mahdollista hyötyä irti testivetoisesta ohjelmistokehityksestä. Väärinkäsitysten läpikäymisen jälkeen pohditaan millainen tulevaisuus testivetoisella ohjelmistokehityksellä mahdollisesti voisi olla.

4.1 Väärinkäsityksiä

Janzen ja Saiedian (2008) huomaamia ohjelmoijien väärinymmärryksiä liittyen testivetoiseen ohjelmistokehitykseen olivat, että automaatiotestaus olisi sama asia kuin testivetoinen ohjelmistokehitys, ja oletus, että testivetoisessa ohjelmistokehityksessä kaikki testit kirjoitettaisiin kerralla ennen koodin tekemistä. Virheelliset mielikuvat voivat aiheuttaa korkeamman kynnyksen muodostusta testivetoisen ohjelmistokehityksen kokeilemiselle ja käytölle. Etenkin lähteissä, jotka ovat helpoimmin saatavilla ja joilla on suuri lukijajoukko, tulisi olla ajantasaista ja totuudenmukaista tietoa testivetoiseen ohjelmistokehitykseen liittyen väärinkäsitysten välttämiseksi.

Testiautomaatiota käsittelevällä englanninkielisellä wikipedia-sivulla väitetään virheellisesti, että ketterässä kehityksessä ja eXtreme programmingissa testiautomaatio tunnetaan testivetoisena ohjelmistokehityksenä tai Test-First ohjelmistokehityksenä[1]. Tämä tieto voidaan todeta vääräksi tutkielmassa aiemmin esitetyn tiedon perusteella. Wikipedia on tietoliikenteen määrältään maailman kymmenentenä[2] ja näin ollen merkittävä tiedon välittäjä. Näin laajasti käytössä olevalla verkkosivulla voidaan olettaa olevan myötävaikutusta väärinkäsi-

1. Alkuperäinen teksti "Test automation, mostly using unit testing, is a key feature of extreme programming and agile software development, where it is known as test-driven development (TDD) or test-first development."(https://en.wikipedia.org/wiki/Test_automation, 2022.2.22)

2. (https://www.alexa.com/topsites, 2022.2.22

tysten levittämiseen.

George ja Williams (2004) väittävät testivetoisen ohjelmistokehityksen heikkoutena olevan ohjelmoijalta vaadittava korkea taitotaso testien kirjoittamista varten kun testiä ollaan tekemässä vaikeasti testattavalle koodille[3]. Korjauksena tälle esitetään, että testien teko ei itsessään vaadi korkeaa taitotasoa ohjelmoijalta, vaan hyvän ja selkeän koodin kirjoittaminen vaatii. Testivetoisessa ohjelmistokehityksessä testi kirjoitetaan ennen koodia ja tämä ohjaa ohjelmoijaa kirjoittamaan selkeää, siisteissä ja pienissä paloissa olevaa koodia. Testivetoisen ohjelmistokehityksen vahvuutena on, että testien teko ennen koodin kirjoittamista johdattaa selkeämmän koodin kirjoittamiseen. Pitkien ja monimutkaisten koodipätkien olisi hankala läpäistä minkäänlaista testiä.

4.2 Testivetoisen ohjelmistokehityksen tulevaisuus

Testivetoinen ohjelmistokehitystä voidaan käyttää omana menetelmänään, kuitenkin se on myös yksi eXtreme Programmingin pääharjoitteista. Molempia menetelmiä on tutkittu huomattava määrä. Dyba ja Dingsoyr (2008) systemaattisessa kirjallisuuskatsauksessa, joka käsittelee ketterään kehitykseen liittyviä empiirisiä tutkimuksia, 33 tutkimuksesta oli eXtreme Programmingia käsitteleviä 25. Oletettavaa on, että testivetoinen ohjelmistokehityksen paikka merkittävänä ketterän kehityksen käytäntönä jatkuu niin itsenäisenä harjoitteena kuin osana eXtreme Programmingia.

3. Alkuperäinen teksti "Skill level. Writing test cases for hard-to-test code requires a high level of experience and determination from programmers."

5 Yhteenveto

Tämän tutkimuksen toivotaan tuovan testivetoisen ohjelmistokehityksen kuvaan selkeyttä ja tietoisuutta, jotta väärinkäsitysten määrä olisi tulevaisuudessa vähäisempi. Näin ollen testivetoista ohjelmistokehitystä voitaisiin tulevaisuudessa myös hyödyntää tehokkaammin.

Testivetoinen ohjelmistokehitys on ohjelmiston suunnittelu- ja analyysimenetelmä, joka on saanut alkunsa osana eXtreme Programmingia. Testivetoinen ohjelmistokehitys etenee sykleissä jotka alkavat yksikkötestin teolla. Kun testi kääntyy tehdään koodi joka läpäisee testin. Syklin lopuksi koodia refaktoroidaan, eli siistitään koodi poistamalla sieltä duplikaatiot ja kaikki muu ylimääräinen sisältö. Jokaisella testien ajokerralla ajetaan kaikki projektiin kuuluvat yksikkötestit. Testivetoisessa ohjelmistokehityksessä hyödynnetään automaatiotestausta. TDD:ssä on suositeltavaa käyttää testausta varten jotakin yksikkötestaustyökalua, esimerkiksi xUnit.

Testivetoiseen ohjelmistokehitykseen liittyy väärinymmärryksiä. Testivetoinen ohjelmistokehitys ei siis ole testausmenetelmä. Se ei myöskään ole sama asia kuin testausautomaatio, eikä se ota kantaa muiden testaustyyppien testien tekoon.

Tämän kirjallisuuskatsauksen heikkoutena on sen laaja-alainen tarkoitus, eli kattavan kokonaiskuvan luominen testivetoisesta ohjelmistokehityksestä ja sen taustoista. Mitä enemmän käytäisiin läpi aiheeseen liittyvää kirjallisuutta sitä tarkempi kuva voitaisiin luoda, mutta sitä enemmän se veisi resursseja. Tämän tutkielman rajoitteena ovat sen resurssien määrät. Tutkielman rajallinen koko, kirjallisuuden saatavuus sekä mahdollisena olevan käytettävän ajan määrä, johtavat siihen että kaikkea testivetoiseen ohjelmistokehitykseen ja sen historiaan liittyvää kirjallisuutta ei ole voitu kartoittaa tässä tutkimuksessa. Tämä voi johtaa tietojen puutteellisuuteen.

Suuri osa löytyneistä tutkimuksista vertasi TDD:n ja TLD:n tuottavuutta ja koodin laatua. Tutkimuksia TDD:n yhteensopivuudesta eri (ketterien) menetelmien kanssa ei löytynyt, joten sitä voitaisiin tutkia. Myös lisää tutkimuksia koskien minkälaisten projektien yhteyteen TDD soveltuisi parhaiten olisi tarvetta.

Lähteet

Aghayi, Emad, Thomas D LaToza, Paurav Surendra ja Seyedmeysam Abolghasemi. 2021. "Crowdsourced Behavior-Driven Development". *The Journal of systems and software* 171.

Beck, Kent. 2000. *Extreme programming explained: embrace change.* Addison-Wesley. ISBN: 0201616416.

———. 2001. "Aim, Fire". *IEEE software* 18 (5): 87–89.

———. 2002. *Test-Driven Development: By Example.* Boston, Mass.: Addison-Wesley. ISBN: 0321146530.

Beck, Kent, ja Cynthia Andres. 2004. *Extreme programming explained: embrace change, Second edition.* Addison-Wesley. ISBN: 0321278658.

Beck, Kent, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning ym. 2001. "Agile Manifesto". Viitattu 6. huhtikuuta 2022. https://agilemanifesto.org/iso/fi/manifesto.html.

Bissi, Wilson, Adolfo Gustavo Serra Seca Neto ja Maria Claudia Figueiredo Pereira Emer. 2016. "The effects of test driven development on internal quality, external quality and productivity: A systematic review". *Information and software technology* 74:45–54. https://doi.org/10.1016/j.infsof.2016.02.004.

Dyba, Tore, ja Torgeir Dingsoyr. 2008. "Empirical studies of agile software development: A systematic review". *Information and software technology* 50 (9): 833–859. https://doi.org/10.1016/j.infsof.2008.01.006.

Fowler, Martin. 2000. *Refactoring: improving the design of existing code.* Addison-Wesley. ISBN: 0201485672.

George, Boby, ja Laurie Williams. 2004. "Information and software technology". *The Journal of systems and software* 46 (5): 337–342. https://doi.org/10.1016/j.infsof.2003.09.011.

Janzen, D.S, ja H Saiedian. 2008. "Does Test-Driven Development Really Improve Software Design Quality?" *IEEE software* 25 (2): 77–84.

Karac, Itir, ja Burak Turhan. 2018. “What Do We (Really) Know about Test-Driven Development?” *IEEE software* 35 (4): 81–85.

Nanthaamornphong, Aziz, ja Jeffrey C Carver. 2015. “Test-Driven Development in scientific software: a survey”. *Software quality journal* 25 (2): 343–372. https://doi.org/10.1007/s11219-015-9292-4.

Nass, Michel, Emil Alégroth ja Robert Feldt. 2021. “Why many challenges with GUI test automation (will) remain”. *Information and software technology* 138. https://doi.org/10.1016/j.infsof.2021.106625.