Tampereen yliopisto

Henri Jussila

TESTIVETOINEN KEHITYS

Ja sen opetus uusille ohjelmoijille

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Tammikuu 2024

TIIVISTELMÄ

Henri Jussila: Testivetoinen kehitys
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Joulukuu 2023

Ohjelmistokehityksen mallit kokivat suuria muutoksia hieman yli 20 vuotta sitten, kun Extreme Programming ja testivetoinen kehitys keksittiin. Työelämässä mallit otettiin nopeasti käyttöön, ja perinteistä vesiputousmallia käytetään nykyään hyvin vähän. Opetus tapahtuu vielä pääosin vesiputousmallia käyttäen, jossa ensin kirjoitetaan koodia, ja testaaminen suoritetaan vasta lopuksi. Työelämän vaatimusten ja opetuksen tarjoamien taitojen välillä suurimmat puutteet ovatkin juuri ketterissä kehitysmalleissa ja testaamisessa. Tässä tutkielmassa tutustutaan testivetoiseen kehitykseen, ja syihin miksi sitä kannattaa käyttää uusien ohjelmointiopiskelijoiden opetuksessa.

Tutkielmassa tarkastellaan erilaisia kehitysmalleja, jotta testivetoista kehitystä voidaan ymmärtää paremmin. Näitä malleja ovat vesiputousmalli, Extreme Programming sekä Test First Development. Testivetoisesta kehityksestä esitellään sen määritelmä, käyttötapa, hyödyt sekä haasteet. Tutkielmassa ei kuitenkaan tarkastella sitä, miltä testivetoisen kehitys näyttää kooditasolla, sillä tarkoitus on keskittyä sen tarjoamiin opetuksellisiin hyötyihin. Työn tavoitteena on arvioida nykyistä ohjelmoinnin opetuskäytäntöä, ja löytää hyötyjä testivetoisen kehityksen opettamisesta uusille ohjelmoijille.

Tutkielma on tehty kirjallisuuskatsauksena. Työssä käydään läpi tutkimuksia, joissa testivetoista kehitystä on otettu mukaan ohjelmointikurssien opetukseen. Tutkimuksissa on havaittu testivetoisen kehityksen parantavan koodin laatua, vähentävän ohjelmointivirheitä sekä tekevän ohjelmointityöstä mielekkäämpää. Opiskelijat, jotka käyttivät testivetoista kehitystä, kertoivat sen auttaneen heitä ymmärtämään kurssin asioita paremmin. Ristiriitaisia tuloksia on saatu ohjelmoijan tehokkuuden suhteen. Joidenkin tutkimusten mukaan testivetoinen kehitys tehostaa ohjelmoijien työskentelyä, koska virheiden korjaamiseen kuluu vähemmän aikaa perinteiseen vesiputousmalliin verrattuna. Toiset tutkimukset ovat taas päätyneet tuloksiin, joissa ei ole havaittu merkittäviä eroja tehokkuudessa. Testivetoisen kehityksen opettamisesta on kuitenkin useita hyötyjä ja se valmistaa ohjelmointiopiskelijoita työelämään.

Avainsanat: Testivetoinen kehitys, TDD, Ketterä kehitys, Vesiputousmalli, Extreme Programming

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

SISÄLLYSLUETTELO

1 Johdanto

Kouluissa ohjelmoinnin opetus tapahtuu pääosin perinteisen vesiputousmallin (Waterfall method) avulla, jossa ensin suoritetaan ohjelman toiminnallisuus ja tämän jälkeen testaaminen. Ohjelman testaaminen suoritetaan usein etukäteen määritetyillä esimerkkisyötteillä, ja kun ohjelma tulostaa halutut syötteet, on testaus valmis. Työelämässä suositaan kuitenkin ketteriä ohjelmistokehitysmenetelmiä vesiputousmallin sijaan. Ketterissä menetelmissä ohjelmointi tapahtuu iteratiivisesti, kun taas vesiputousmallissa ohjelmointi on lineaarista. On selvää, että koulutuksen tarkoitus on valmistaa opiskelija työelämään. Miksi niin suuri osa opetuksesta suoritetaan sitten vesiputousmallin avulla?

Tutkielman aiheena on testivetoinen kehitys (Test-Driven Development = TDD) ja sen opetus uusille ohjelmoijille. Testivetoinen kehitys on ohjelmistokehityksen malli, joka on osa ketterää ohjelmistokehitysmenetelmää nimeltä Extreme Programming (EX) (Nanthaamornphong & Bressan, 2019). TDD:tä voidaan kuitenkin käyttää myös itsenäisesti ja siinä yhdistyy Test-First Development (TFD) sekä refaktorointi (Bakhtiary, Gandomani & Salajegheh, 2020).

TDD:n käytöstä on todettu olevan useita eri hyötyjä niin akateemisessa ympäristössä kuin työelämässä. Hyödyt eivät rajoitu pelkästään itse koodiin, kuten sen laatuun ja lisääntyneeseen modulaarisuuteen, vaan löydettyjä hyötyjä on myös itse ohjelmoijalle sekä työelämässä asiakkaalle, joka tilaa ohjelmiston. Lisäksi työnantajat suosivat työnhaussa opiskelijoita, joilla on kokemusta ketteristä kehitysmalleista sekä testaamisesta.

Tutkimuskysymykseni on: Miksi testivetoista ohjelmistokehitystä kannattaa opettaa uusille ohjelmoijille? Tutkielmani tavoitteena on oppia lisää testivetoisesta ohjelmistokehityksestä ja mitä hyötyjä sen opetuksesta on opiskelijoille tai uusille ohjelmoijille. Suoritan tutkielmani kirjallisuuskatsauksena, jossa tutkin aiemmin tehtyjä tutkimuksia TDD:n opettamisen hyödyistä aloitteleville ohjelmoijille.

Tutkin aihetta yksittäisen ihmisen näkökulmasta, keskittyen opiskelijan näkökulmaan. Aiheenani on opiskelijan hyödyt TDD:n käytöstä, joten uskon tämän näkökulman palvelevan tutkimustani parhaiten. Tiedonhaun olen suorittanut tietokannoista: Andor, Google Scholar, IEEE Xplore ja ProQuest käyttäen hakusanoja test-driven development, TDD, TDD and students, teaching test-driven development ja muita vastaavia yhdistelmiä.

Aloitan kertomalla eri ohjelmistokehityksen malleista. Ensin kerron perinteisestä vesiputousmallista, sitten ketteriin ohjelmistomenetelmiin kuuluvasta Extreme Programmingista ja Test-First Developmentista. Tämän jälkeen aloitan itse TDD:n käsittelyn. Avaan hieman koko käsitettä, ja kerron tarkemmin mitä TDD on ja kuinka se toimii. Käsittelen ensin hyötyjä, joita TDD:n käytöstä on itse ohjelmoijalle ja koodille, sitten muita yleisiä hyötyjä. Tämän jälkeen käsittelen mahdollisia haasteita ja haittoja, joita TDD:n

käytössä voi olla. Lopuksi kerron miten TDD:tä opetetaan ja millaisia eri tutkimustuloksia sen opettamisesta on saatu.

2 Muita ohjelmistokehityksen malleja

Tässä kappaleessa tutustutaan muihin ohjelmistokehityksen malleihin, jotta tutkielman aiheena olevan testivetoisen kehityksen ymmärtäminen olisi helpompaa. Ensimmäisenä käydään läpi perinteistä vesiputousmallia, jota käytetään usein opeuksessa. Tämän jälkeen avataan ketterän kehityksen mallia Extreme Programming, ja lopuksi tutkitaan vielä mitä on Test-First Development, joka on jo hyvin lähellä testivetoista kehitystä.

2.1 Vesiputousmalli

Vesiputousmalli ja sen muodot ovat ohjanneet ohjelmistokehitystä monien vuosikymmenten ajan. Tässä mallissa ohjelmistokehitys etenee lineaarisesti alkaen suunnittelusta, analyysista, mallintamisesta ja ohjelmoinnista, jonka jälkeen seuraa testaus. Mallissa oletetaan, että vaatimukset pysyvät muuttumattomina, ja että kehitysprosessi, mukaan lukien mahdolliset ongelmat, voidaan ennakoida etukäteen. Vesiputousmallissa erillinen laadunvarmistus- tai testausryhmä vastaa testaamisesta ja varmistaa, että tuotetun ohjelmiston laatu vastaa odotuksia. Tätä ohjelmistokehityksen lähestymistapaa kutsutaan myös perinteiseksi- (traditional approach) tai testaa-viimeisenä-lähestymistavaksi (Test-Last Approach). (Bhadauria, Mahapatra & Nerur, 2020)

Vesiputousmallin ongelmia ovat epäselvät vaatimukset ohjelmiston suhteen asiakkaalta, sekä heidän taipumuksensa muuttaa niitä. On myös ollut tapauksia, joissa asiakkaat unohtivat mitä vaatimuksia heillä on ollut. Myös ohjelmoijilla on ollut ongelmia vesiputousmallin kanssa. Kun he luulivat, että työ on melkein valmis, oli siitä todellisuudessa vasta 1/3 valmiina. (Shrivastava, Jaggi, Katoch, Gupta, D., & Gupta, S., 2021)

Valmiin työn hahmottamista vaikeuttaa testien sijoittaminen vasta projektin loppuvaiheeseen, kun koko työ on tehty. Testaaminen voi vastata jopa 50 % ohjelmointiajasta. Syynä tähän on projektin lopussa tapahtuva verifiointiprosessi, jonka aikana löydetään usein suuri määrä vikoja. Vikojen löytäminen projektin loppuvaiheessa johtaa usein aikataulusta myöhästymiseen. (Damm, Lundberg & Olsson, 2005)

2.2 Extreme Programming

Perinteisten ohjelmistomallien ongelmat, joita edellä kuvattiin, johtivat niin asiakkaat kuin ohjelmoijat vaatimaan iteratiivista menetelmää, jossa on lyhyemmät ohjelmointisyklit. Vaatimuksia vastaamaan syntyi Extreme Programming, joka on yksi ketterän kehityksen malleista (Shrivastava ja muut, 2021). Ketterän kehityksen tavoite on päästä eroon

perinteisten ohjelmistokehitysmallien kankeudesta ja edistää nopeaa reagointia muuttuviin käyttäjävaatimuksiin sekä tiukkoihin projektiaikatauluihin (Erickson, Lyytinen & Siau, 2005). XP koostuu joukosta arvoja, periaatteita sekä käytäntöjä, jotka ovat osoittautuneet hyödyllisiksi korkealaatuisen ohjelmiston luomisessa. Nämä käytännöt viedään äärimmäisyyksiin, josta tulee nimi Extreme Programming. (Akhtar, A., Bakhtawar, & Akhtar, S, 2022) XP:ssä ohjelmointi tehdään asiakkaan vaatimusten pohjalta, jonka jälkeen suoritetaan testaaminen koodin toimivuuden varmistamiseksi. Toisin kuin vesiputousmallissa, XP:ssä kaikkia vaatimuksia ei tarvitse saada heti projektin alussa asiakkaalta, vaan niitä voidaan lisätä ohjelmaan kesken projektin (Erickson ja muut, 2005). Tämä on yksi XP:n suurimmista eduista asiakkaalle sekä ohjelmoijille vesiputousmalliin verrattuna. Mallissa korostuu useat syklit pienen ajan sisällä, joiden tarkoitus on ohjelmiston parantaminen askel kerrallaan. Etusijalla on ryhmätyöskentely ohjelmoijien sekä asiakkaiden välillä. (Shrivastava ja muut, 2021)

Vesiputousmallin yksi isoimmista ongelmista liittyi testaamiseen, joka suoritettiin vasta viimeisenä, kun koko ohjelma oli valmis. XP:ssä toiminnallisuus toteutetaan pienissä sykleissä, ja jokaisen syklin jälkeen suoritetaan myös testaaminen, jolloin testattava kokonaisuus pysyy pienempänä. Asiakkaat ovat tiiviisti mukana projektissa ja heidän tulee suorittaa jatkuvaa ominaisuuksien testaamista sitä mukaan, kun uutta toiminnallisuutta valmistuu. Testejä voidaan myös kirjoittaa jo ennen itse toiminnallisuuden kirjoittamista. (Shrivastava ja muut, 2021) Menetelmä, jossa testi kirjoitetaan ennen toiminnallisuuden kirjoittamista, on nimeltään Test-First Development.

2.3 Test-First Development

Test-First Development on ohjelmistokehityksen käytäntö, joka on keskeisessä osassa TDD:ssä sekä XP:ssä. TFD:ssä kirjoitetaan ensin testit halutulle toiminnallisuudelle, jonka jälkeen kirjoitetaan koodia, joka läpäisee kyseiset testit. Sellaista koodia ei tule kirjoittaa, jonka tavoite ei ole läpäistä testejä. Toiminnallisuus katsotaan olevan valmis, kun kaikki testit menevät läpi. Tähän kuuluu juuri kirjoitettu testi sekä kaikki mahdolliset aiemmat testit. Jos ohjelmasta löytyy puutteita tai vikoja, kirjoitetaan ensin testit, jonka jälkeen vasta kirjoitetaan itse koodi, jolla puute tai vika korjataan. Näin ohjelma rakentuu pienissä osissa ja testejä tulee ajettua jatkuvasti. ”If you can’t write a test for what you are about to code, then you should not even be thinking about coding” on yksi TFD:n säännöistä. (Huang & Holcombe, 2009)

TFD:ssä ohjelma rakentuu pienissä osissa, toiminnallisuus kerrallaan. Testien avulla varmistetaan, että lisätty funktionaalisuus toimii, eikä riko jo olemassa olevaa kokonaisuutta. Tämä auttaa tunnistamaan koodissa olevia ongelmia aikaisessa vaiheessa, vähen-

tää debugaukseen käytettävää aikaa ja vaivaa. Ohjelmoinnista tulee tehokkaampaa. Lisäksi modulaarinen rakenne ja laaja testikattavuus helpottavat ohjelman ylläpidettävyyttä. (Huang & Holcombe, 2009)

3 Testivetoinen kehitys

Tässä kappaleessa tutkitaan työn aiheena olevaa testivetoista kehitystä. Käydään ensin läpi TDD:n määritelmä ja kuinka se on yhteydessä aiemmin läpikäytyyn TFD:hen. Tämän jälkeen tutkitaan syklejä, joista TDD muodostuu ja mitä kussakin syklissä tehdään. Lopuksi käydään läpi mitä hyötyjä TDD:n käytöstä on, sekä mitä eri haasteita TDD:n käytölle on löydetty.

3.1 Määritelmä

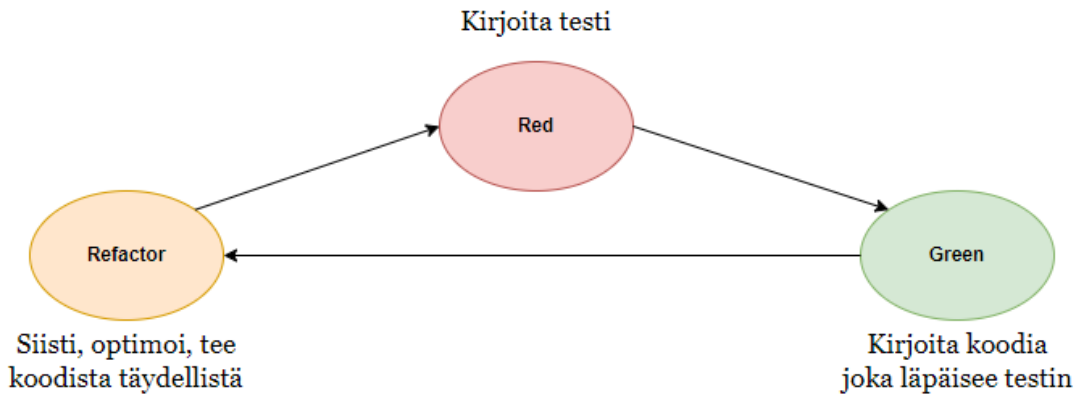
Lyhyesti TDD voidaan määritellä matemaattisella kaavalla TDD = TFD + refaktorointi. Voidaan sanoa, että jos ohjelmistoprojekti suoritetaan pienissä osissa käyttäen TFD:tä ja nämä kaikki osat refaktoroidaan laadun varmistamiseksi, noudatetaan TDD:tä (Huang & Holcombe, 2009). TDD on noussut yhdeksi suosituimmista ketterän kehityksen malleista. Merkittävä muutos perinteisiin kehitysmalleihin verrattuna on TDD:n keskiössä oleva testien kirjoittamien ennen itse toiminnallisuuden kirjoittamista. (Bhadauria ja muut, 2020)

TDD on sykleissä tapahtuvaa ohjelmointia, jossa yksikkötestit ohjaavat pienissä vaiheissa rakentuvaa toiminnallisuutta. Jokainen ohjelmointisykli alkaa yksikkötestin kirjoittamisella sellaiselle toiminnallisuudelle, jota ei ole vielä luotu. Toiminnallisuus kirjoitetaan testin luomisen jälkeen. Sykli päättyy, kun yksikkötesti läpäistään. Refaktoroinnilla on TDD:ssä tärkeä rooli, sillä se auttaa ohjelmoijaa kehittämään koodin sisäistä rakennetta ja mallia (design) ulkoisen toiminnallisuuden pysyessä muuttumattomana. (Baldassarre ja muut, 2021) Refaktoroinnin määritelmä käydään läpi seuraavassa kappaleessa. Uuden toiminnallisuuden katsotaan olevan toimiva vasta, kun se läpäisee myös kaikki aiemmat testit (Huang & Holcombe, 2009).

TDD ei kuitenkaan ole keino testata koodia, vaan ohjelmistokehitysmenetelmä, jolla saadaan aikaan johdonmukaisempaa koodia. Menetelmä opettaa ohjelmoijalle tehokkuutta, modulaarisempaa koodia, laajennettavuutta sekä laadukkaampaa koodia (Bakhtiary ja muut, 2020). Modulaarisuus, laajennettavuus ja laatu ovat kaikki mittareita, joita käytetään, kun koodin laatua arvioidaan.

3.2 Syklit

TDD:ssä ohjelmointi tapahtuu jatkuvissa sykleissä (Red-Green-Refactor). Katsotaan tarkemmin mitä jokainen sykli pitää sisällään.



Kuva 1 Testivetoisen kehityksen syklit (Bajaj, Patel, H. & Patel, J., 2015)

Red

Uusi sykli alkaa aina punaisesta vaiheesta. Kirjoitetaan testi, jolle ei vielä ole toteutusta. Testi ei vielä testaa mitään, joka tarkoittaa, että testi ei mene läpi. Testi näyttää siis ”punaista”. Jos testi jostain syystä menee läpi tarkoittaa se, että se ei testaa uutta funktionaalisuutta. Tässä tapauksessa testiä muutetaan niin että se epäonnistuu. (Staegemann ja muut, 2022)

Green

Vihreässä vaiheessa kirjoitetaan minimaalinen määrä koodia, joka läpäisee testin. Tässä vaiheessa keskitytään ainoastaan koodin toiminnallisuuteen ja kaikki muu voidaan unohtaa, kuten koodin laatu ja hyvät muuttujien nimeämiskäytännöt. (Staegemann ja muut, 2022) Tärkeintä on saada testi läpi, eli näyttämään ”vihreää” mahdollisimman nopeasti.

Refaktorointi

Refaktorointi-vaihe erottaa TDD:n TFD:stä. Refaktoroinnilla tarkoitetaan koodin yleistä siistimistä ja parantamista ilman, että sen toiminnallisuus muuttuu. Tässä vaiheessa siis palataan siistimään ja parantamaan vihreässä vaiheessa kirjoitettua koodia, lisäämättä uutta toiminnallisuutta. Kun refaktorointi on suoritettu, täytyy testien mennä edelleen läpi. Kun kaikki vaiheet on suoritettu, aloitetaan sykli alusta uuden toiminnallisuuden lisäämiseksi (Bajaj ja muut, 2015).

3.3 Hyödyt

TDD:n hyödyistä on tehty lukuisia tutkimuksia niin ohjelmistokehityksen alalla kuin akateemisissa ympäristöissä. Ohjelmiston rakentuessa pienissä osissa on sen ylläpito ja laajentaminen helpompaa. Kattavat testit jokaiselle toiminnallisuudelle parantavat ohjelmiston luotettavuutta ja vähentävät vikojen korjauksiin kuluvaa aikaa. Tämä myös parantaa ohjelmoijan tuottavuutta ja mahdollisesti lyhentää koko ohjelmiston kehitysaikaa. Refaktorointi parantaa koodin laatua. (Sheikh, 2022)

Sheikh (2022) nostaa tutkimuksessaan esille laajamittaisen kirjallisuuskatsauksen, joka tehtiin vuosina 1999–2014 välissä julkaistuille TDD:tä ja Test-Last Developmenttia (TLD) vertaaville tutkimusartikkeleista. Näistä artikkeleista 76 % päätyi lopputulokseen, että TDD parantaa ohjelmiston sisäistä laatua ja 88 % osoitti TDD:n parantavan ohjelmiston ulkoista laatua TLD:hen verrattuna. Vesiputousmalli on yksi esimerkki TLD:stä.

Vaikka tutkimuksia on tehty lukuisia kaikista hyödyistä ei ole saatu yhteneviä tuloksia. Amrit ja Meijberg (2018) huomauttavat, että aiemmat tutkimukset ovat osoittaneet, että koodin laatu paranee TDD:tä käyttämällä, mutta yhtenäisiä tuloksia ei ole saatu ohjelmoijan tehokkuuden suhteen. Testien kirjoittaminen etukäteen vaatii ylimääräistä työtä, mikä johtaa ohjelmoijan tehokkuuden laskuun. Toisaalta, koska TDD parantaa koodin laatua ja johtaa vähäisempään virheiden määrään sekä nopeampaan virheiden korjaamiseen, on se loppupeleissä tehokkaampaa. On siis hieman tulkinnanvaraista, mitä käsite ohjelmoijan tehokkuus tarkoittaa missäkin tilanteessa. Tätä pohditaan tarkemmin työn loppupuolella.

Hyödyt eivät rajoitu pelkästään itse koodiin, vaan hyötyjä on löydetty myös itse ohjelmoijalle. TDD:n on tutkittu vaikuttavan positiivisesti ohjelmointityön mielekkyyteen (Bhadauria ja muut, 2020). Pienempien ominaisuuksien ohjelmointi on usein selkeämpää kuin kerralla ison kokonaisuuden toteuttaminen. Laaja testikattavuus pitää huolen siitä, että ohjelma toimii, vaikka uutta toiminnallisuutta lisätään koko ajan. Ohjelman toteuttaminen pienissä osissa on varmasti myös palkitsevampaa, sillä tunne saavutuksesta syntyy jokaisen syklin päätteeksi.

TDD:stä hyötyvät myös ohjelmistoprojekteissa olevat asiakkaat, sillä sen on todettu kehittävän ohjelmoijien, käyttäjien sekä asiakkaiden välistä ymmärrystä ohjelmiston vaatimusten suhteen (Nanthaamornphong & Bressan, 2019). Vaatimusten on oltava todella selkeitä, jotta oikeanlaiset testit voidaan kirjoittaa. Lisäksi TDD:n osaamisesta voi olla hyötyä työnhaussa. Työnantajat suosivat valmistuneita, joilla on kokemusta ketterän kehityksen menetelmistä. Menetelmien hyödyntäminen ei kuitenkaan ole pelkästään hyödyllistä työnhaun näkökulmasta, vaan se kehittää myös opiskelijan oppimista. (Sheikh, 2022) Tarkastellaan seuraavan kappaleen jälkeen enemmän TDD:n opetuksesta saatavia hyötyjä.

3.4 Haasteet

Vaikka TDD:n käytöstä on todettu useita hyötyjä myös haasteita ja rajoituksia sen käytöstä on löydetty. Käyn tässä vaiheessa läpi ensin merkittävimmät haasteet ja lopuksi listaan muita haasteita, joita tutkimuksissa on noussut esiin.

Staegemann ja muut (2022) käyttivät tutkimuksessaan kymmentä tieteellistä julkaisua, jotka tutkivat TDD:n haasteita. Yksi isoin haaste oli ohjelmoijien kokemattomuus ja tietämättömyys TDD:stä ja sen käytöstä ohjelmointiprojektissa, mikä esti heitä luomasta tehokkaita testejä. On ymmärrettävää, että ilman kokemusta, on mikä tahansa tehtävä haastavaa. Suurimpia puutteita koulutuksen tarjoamien tietojen ja työelämän vaatimusten välillä on juuri ohjelmistokehitysmalleissa ja menetelmissä sekä testaamisessa (Garousi, Giray, Tüzün, Catal & Felderer, 2019). Tämä osoittaa kuinka tärkeää TDD:n opetus kouluissa olisi.

Toinen suuri haaste liittyy TDD:n periaatteeseen, jossa koodin kirjoittaminen alkaa aina alusta. Tämä on kuitenkin harvoin totta, vaan usein projekteissa on mukana legacy koodia. TDD ei ota huomioon legacy koodia, mikä usein tekee sen käytöstä mahdotonta tietyissä projekteissa (Staegemann ja muut, 2019).

Muita usein havaittuja haasteita olivat kasvanut ajankäyttö testien kirjoittamiseen, vaikeus kirjoittaa testejä monimutkaisille funktioille ja graafisille käyttöliittymille, johtoportaan ymmärtämättömyys TDD:stä sekä ohjelmistojen puute testien tekemistä varten (Staegemann ja muut, 2019).

4 Opetus

Tässä kappaleessa käydään tarkemmin läpi ohjelmoinnin opetusta kouluissa. Ensin käydään läpi, miten ohjelmointia perinteisesti opetetaan uusille ohjelmoijille ja mitä vaikutuksia sillä on heidän oppimiseensa. Tämän jälkeen tutkitaan mitä todettuja hyötyjä TDD:n opetuksella on ja millaisia eri tuloksia aiemmista tutkimuksista on saatu.

4.1 Taustaa

“We became painfully aware of the gap between what we were saying to the students about the importance of testing, in contrast to what we were implicitly teaching about testing” (Chow, Komarlu & Conrad, 2021). Näin kuvaa yhdysvaltalaisen UC Santa Barbaran yliopiston ohjelmointikurssia opettava henkilökunta tilannetta, jossa he huomasivat olevansa ohjelmistotestauksen kanssa. Tilanne sopii varmasti myös moneen muuhun kouluun.

Koulutuksen pitäisi valmistaa opiskelija mahdollisimman hyvin työelämään. Ketterän kehityksen menetelmät ovat työelämässä laajalti käytössä, mutta niitä harvemmin opetetaan ohjelmointia opiskeleville. Ketterien mallien sijasta opetus tehdään usein perinteisiä

malleja, kuten vesiputousmallia, käyttäen. Työelämässä ketterät mallit ovat pääosin korvanneet perinteiset mallit (Sheikh, 2022).

Opiskelijoille voi helposti syntyä mielikuva, että testaaminen ei ole oleellinen osa ohjelmointia, jos sitä ei opeteta oikein heti alussa. Useat opiskelijat ajattelevat ohjelmistotestauksen olevan työlästä, tylsää sekä epäolennainen osa ohjelmointitehtävää. Ongelmana on myös opetuksen keskittyminen ohjelman rakentamiseen, ei sen laadun varmistamiseen. (Arcuri 2020) Alalla työskennelleet kuitenkin kuvaavat testaamista täysin päinvastoin. Taakan sijasta he pitävät testaamista vapauttavana. Laaja testikattavuus parantaa ohjelmiston luotettavuutta ja näin refaktorointia tulee tehtyä herkemmin ylläpidettävyyden parantamiseksi. (Chow, Komarlu & Conrad, 2021)

Testaamisen puute tai sen vähäisyys opetuksessa johtaa pahimmassa tapauksessa huonojen ohjelmointitapojen oppimiseen, joista on vaikea päästä myöhemmin eroon. Opiskelijoiden testaamistottumuksissa onkin todettu puutteita. Testaaminen tehdään usein vasta viimeisenä, kun ohjelma on muuten valmis, ja siihen käytetään vain muutamia testejä (Nanthaamornphong & Bressan, 2019). Testaaminen on usein myös hyvin suppeaa. Opiskelijalle syntyy helposti tapa luoda ohjelma, joka on kirjoitettu toimimaan vain tietyillä syötteillä, joita hän pitää todennäköisimpinä. Työelämässä on otettava huomioon myös epätodennäköiset syötteet, jotka voivat mahdollisesti rikkoa ohjelman. (Sheikh, 2022)

4.2 Opetuksen hyödyt

Testivetoisen kehityksen opetusta ohjelmointiopiskelijoille on tutkittu laajasti, ja useita hyötyjä on noussut esille tutkimuksissa. Ohjelmointitehtävän tarkastelu useista eri näkökulmista ja eri ratkaisumenetelmien pohdinta ennen ohjelmoinnin aloittamista on yksi ohjelmoijan tärkeimmistä taidoista. Opiskelijat usein aloittavat ohjelman kirjoittamisen välittömästi, miettimättä tehtävänantoa eri näkökulmista ja kaikkia mahdollisia eri ratkaisuita. Tämä harvoin johtaa laadukkaaseen lopputulokseen. (Sheikh, 2022) Kun toteutettavan toiminnallisuuden testit tehdään ennen itse toiminnallisuutta, on ohjelmoija pakotettu miettimään ratkaisuaan laajemmin.

TDD:n käytön on todettu myös edistävän oppimista ja auttavan opiskelijoita ymmärtämään paremmin kurssin aiheita. Testit tarjoavat välitöntä palautetta ja niitä voidaan ajaa askel kerrallaan debugger-tilassa, jolloin ohjelman toiminnallisuuden ymmärtäminen helpottuu. Testit usein kuvaavat esimerkiksi algoritmin oikeita käyttötilanteita, mikä voi helpottaa sen ymmärtämistä (Arcuri, 2020). Tätä tulkintaa tukee Arcurin (2020) kurssin lopussa järjestämä kysely, jossa tiedusteltiin ohjelmistotestauksesta kurssin aikana. Tuloksista selvisi, että suurin osa vastanneista piti ohjelmistotestausta hyödyllisenä ja he kokivat sen auttaneen heitä ymmärtämään kurssin asioita paremmin. Lisäksi, 81 % kyselyyn

vastanneista piti ohjelmistotestauksen opetusta hyvänä asiana. Aiheen syvällisempi ymmärtäminen tekee siitä myös yleensä mielenkiintoisempaa ja mielekkäämpää opiskelijalle. TDD:llä opettaminen on lähempänä työelämän toimintamallia, mikä voi kasvattaa kiinnostusta ohjelmistokehitystä kohtaan. TDD:llä opetusta saaneet opiskelijat olivat kurssin lopussa kiinnostuneempia ohjelmistokehityksestä kuin ne, joille opetettiin vesiputousmallilla (Nanthaamornphong & Bressan, 2019).

Testivetoisen kehityksen on todettu vaikuttavan positiivisesti tuotetun koodin laatuun. Useissa tutkimuksissa on huomattu, että kun opiskelijat käyttivät TDD:tä oli tuotettu koodi laadukkaampaa ja he tekivät vähemmän ohjelmointivirheitä (Nanthaamornphong & Bressan, 2019). Koodin laadun huomattiin kasvavan lineaarisesti suhteessa kirjoitettujen testien määrään (Sheikh, 2022). Laadukkaampi koodi tarkoittaa usein koodia, joka on modulaarisempaa, muuttujat ovat selkeästi nimetty eikä koodissa esiinny turhaa toistoa. Kun koodi kirjoitetaan testit edellä, on selvää, että myös virheitä tulee vähemmän.

Eri tutkimukset ovat saaneet myös toisistaan hieman poikkeavia tuloksia. Baldassarre ja muut (2021) huomauttavat, että heidän tutkimuksessaan ei ollut eroja koodin laadulla tai ohjelmoijan tehokkuudella TDD:n ja TLD:n välillä, mutta TDD:n käyttö johti laajempiin testikattavuuksiin sekä tarkempaan virheiden huomaamiseen. Myös Romano, Fucci, Baldassarre, Caivano & Scanniello (2019) nostavat esiin, että akateemisessa ympäristössä TDD:n käyttö parantaa koodin laatua tehokkuuden kustannuksella. Tutkimusten erot voivat johtua useista eri syistä, ja niitä käydään tarkemmin läpi seuraavassa kappaleessa.

5 Keskustelu

Testivetoisesta kehityksestä on hyötyä ohjelmoijalle, kirjoitetulle koodille sekä opiskelijalle. Hyödyt yhdistettynä työelämän tarpeisiin ketterien kehitysmallien sekä testaamisen osaamisesta ovat mielestäni riittävät TDD:n tai muun vastaavan mallin sisällyttämiseksi osaksi opetusta. Miksi näin ei sitten kuitenkaan ole?

Moni opiskelija on täysin kokematon ohjelmoinnin suhteen ensimmäisillä ohjelmointikursseilla. Uutena asiana ovat myös eri työkalut kuten terminaali, tulkit ja GIT. Testien kirjoittaminen ennen toiminnallisuutta on luultavasti liian vaikeaa, kun jo pelkän toiminnallisuuden keksiminen on haastavaa. Ohjelmointitaidot eivät siis välttämättä yksinkertaisesti riitä TDD:n opettamiseen ensimmäisillä kursseilla. Olisi mielenkiintoista saada tutkimustuloksia siitä onko toiminnallisuuden kirjoittaminen haastavampaa, kuin sen testin kirjoittaminen aloittelevalle ohjelmoijalle.

On myös ymmärrettävä, että kaikkiin tehtäviin ei TDD:tä kannata käyttää, vaan on lineaarinen vesiputousmalli tehokkaampi ajankäytöllisesti. Lyhyemmät tehtävät veisivät enemmän aikaa, jos niihin pitäisi ensin suunnitella ja kirjoittaa testit ja vasta tämän jälkeen toiminnallisuus. Ensimmäisillä ohjelmointikursseilla on paljon asiaa opeteltavana,

vaikka tehtävät ovatkin melko lyhyitä. Eikö TDD:n pitänytkin parantaa ohjelmoijan tehokkuutta? Tästä eri tutkimukset olivat saaneet ristiriitaisia tuloksia, minkä takia sitä on mielenkiintoista pohtia.

Tutkimuksien väliset erot tehokkuuden suhteen voivat johtua siitä, miten tehokkuuden mittaamista tulkitaan. Pienempi ja yksinkertaisempi ohjelma voi olla nopeampi tehdä perinteisellä vesiputousmallilla, jossa kirjoitetaan ensin toimiva ohjelma ja testit vasta viimeisenä. Suurempaa ohjelmistoa tehtäessä tällä tyylillä on virheiden löytäminen ja korjaaminen usein työläämpää jälkikäteen. Uskon, että ensimmäisten ohjelmointikurssien useimmat tehtävät, jotka ovat pienimuotoisia ohjelmia, ovat tehokkaampi kirjoittaa vesiputousmallilla kuin TDD:llä. Vaikka virheitä tulisikin, ovat ne nopea löytää ja korjata, koska ohjelma on pieni ja yksinkertainen. Hyöty tehokkuuden suhteen luultavasti saavutetaan vasta, kun ohjelma on suurempi.

En lähtisi koko kurssia opettamaan TDD:llä, mutta uskon että sen sisällyttäminen opetukseen olisi hyödyllistä. Tällä tavalla uudet ohjelmoijat saisivat heti alkuvaiheessa kosketuksen testaamiseen ja ketteriin kehitysmenetelmiin. Tarjolla voisi myös olla kursseja, joissa ketterät kehitysmenetelmät sekä testaaminen ovat pääpainona.

6 Yhteenveto

Työelämässä on siirrytty pääosin käyttämään ketterän kehityksen malleja, joissa ohjelmointi tapahtuu iteratiivisesti pienemmissä sykleissä. Testivetoinen kehitys on yksi suosituimmista ketterän kehityksen malleista, ja sen käytöstä on löydetty useita hyötyjä. TDD opettaa ongelman kriittistä pohdintaa useasta eri näkökulmasta ja sen avulla kirjoitetussa ohjelmassa on usein vähemmän virheitä ja enemmän testejä.

Kouluissa ohjelmoinnin opetus tapahtuu pääosin perinteisten kehitysmallien, kuten vesiputousmallin avulla. Näissä malleissa kirjoitetaan ensin ohjelma ja testaaminen suoritetaan viimeisenä. Tällainen työskentely voi johtaa vääristyneeseen kuvaan siitä millaista ohjelmointi tulee olemaan työelämässä. Suurimpia puutteita koulutuksen tarjoamien tietojen ja työelämän vaatimusten välillä on todettu olevan juuri ohjelmistokehitysmalleissa sekä testaamisessa.

Aiemmat tutkimukset ovat todistaneet, että testivetoisen kehityksen opettaminen johtaa parempaan ohjelmiston laatuun ja vähentää ohjelmoijien tekemiä virheitä. Vaikka eri tutkimukset ovat saaneet myös ristiriitaisia tuloksia ohjelmiston laadun ja ohjelmoijan tehokkuuden suhteen, ovat ne aina olleet yksimielisiä siitä, että TDD:n käyttö johtaa tarkempaan virheiden huomaamiseen. Lisäksi TDD:n käyttö auttaa opiskelijoita ymmärtämään kurssin asioita paremmin sekä tekee ohjelmoinnista mielekkäämpää. Opiskelijoilla on myös paremmat työllistymismahdollisuudet, jos heillä on aiempaa kokemusta ketteristä kehitysmalleista sekä testaamisesta.

Tutkimukseni perusteella on selvää, että TDD:llä on useita positiivisia vaikutuksia ohjelmointiopiskelijoiden oppimiseen ja valmistautumiseen työelämään. Uskon, että testivetoisen kehityksen opettaminen ohjelmoinnin johdantokursseilla auttaisi opiskelijoita kehittämään ohjelmointitaitoja, jotka vastaavat työelämän tarpeita. Kaikkea ei tarvitse opettaa TDD:llä, mutta en näe syytä miksi sitä ei voisi opettaa pienissä määrissä jo johdantokursseilla. Opetustarjontaan voisi lisätä jatkokurssin, jossa keskitytään ketteriin kehitysmalleihin sekä testaamiseen, joka kuvastaa työelämää.

Lähdeluettelo

Akhtar, A., Bakhtawar, B., & Akhtar, S. (2022). Extreme Programming Vs Scrum: A Comparison Of Agile Models. *International Journal of Technology, Innovation and Management (IJTIM),* 2(2), 80-96. https://doi.org/10.54489/ijtim.v2i2.77

Amrit, C., & Meijberg, Y. (2018). Effectiveness of Test Driven Development and Continuous Integration - A Case Study. *IT Professional*, 20(1), 1–1. https://doi.org/10.1109/MITP.2017.265104251

Arcuri, A. (2020). Teaching Software Testing in an Algorithms and Data Structures Course. *In 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (pp. 419-424). IEEE. https://doi.org/10.1109/ICSTW50294.2020.00075

Bakhtiary, V., Gandomani, T. J., & Salajegheh, A. (2020). The effectiveness of test-driven development approach on software projects: A multi-case study. *Bulletin of Electrical Engineering and Informatics*, 9(5), 2030-2037. https://doi.org/10.11591/eei.v9i5.2533

Baldassarre, M. T., Caivano, D., Fucci, D., Juristo, N., Romano, S., Scanniello, G., & Turhan, B. (2021). Studying test-driven development and its retainment over a six-month time span. *The Journal of Systems and Software,* 176, 110937–. https://doi.org/10.1016/j.jss.2021.110937

Bajaj, K., Patel, H., & Patel, J. (2015). Evolutionary software development using test driven approach. *In 2015 International Conference and Workshop on Computing and Communication (IEMCON)* (pp. 1-6). IEEE. https://doi.org/10.1109/IEMCON.2015.7344510

Bhadauria, V. S., Mahapatra, R. K., & Nerur, S. P. (2020). Performance outcomes of test-driven development: An experimental investigation. *Journal of the Association for Information Systems,* 21(4), 1045–1071. https://doi.org/10.17705/1jais.00628

Chow, S. P., Komarlu, T., & Conrad, P. T. (2021). Teaching testing with modern technology stacks in Undergraduate Software Engineering Courses. *In Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (pp. 241-247). https://doi.org/10.1145/3430665.3456352

Damm, L. O., Lundberg, L., & Olsson, D. (2005). Introducing test automation and test-driven development: An experience report. *Electronic notes in theoretical computer science*, 116, 3-15. https://doi.org/10.1016/j.entcs.2004.02.090

Erickson, J., Lyytinen, K., & Siau, K. (2005). Agile modeling, agile software development, and extreme programming: the state of research. *Journal of Database Management (JDM)*, 16(4), 88-100.

Garousi, V., Giray, G., Tüzün, E., Catal, C., & Felderer, M. (2019). Aligning software engineering education with industrial needs: A meta-analysis. *Journal of Systems and Software*, *156*, 65-83.

Huang, L., & Holcombe, M. (2009). Empirical investigation towards the effectiveness of Test First programming. *Information and Software Technology*, *51*(1), 182-194.

Nanthaamornphong, A., & Bressan, S. (2019). The empirical study: encouraging students' interest in software development using test-driven development. *Tehnički Glasnik,* 13(4), 267–274. *https://doi.org/10.31803/tg-20191104214708*

Romano, S., Fucci, D., Baldassarre, M. T., Caivano, D., & Scanniello, G. (2019). An empirical assessment on affective reactions of novice developers when applying test-driven development. *In International Conference on Product-Focused Software Process Improvement* (pp. 3-19). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-35333-9_1

Sheikh, W. (2022). Teaching C++ programming using automated unit testing and test-driven development—Design and efficacy study. *Computer Applications in Engineering Education,* 30(3), 821–851. https://doi.org/10.1002/cae.22488

Shrivastava, A., Jaggi, I., Katoch, N., Gupta, D., & Gupta, S. (2021). A systematic review on extreme programming. In *Journal of Physics: Conference Series* (Vol. 1969, No. 1, p. 012046). IOP Publishing. *https://doi.org/10.1088/1742-6596/1969/1/012046*

Staegemann, D., Volk, M., Perera, M., Haertel, C., Pohl, M., Daase, C., & Turowski, K. (2022). A Literature Review on the Challenges of Applying Test-Driven Development in Software Engineering. *Complex Systems Informatics and Modeling Quarterly*, (31), 18-28.